Bruce Eckel

Thinking in Java

Vierte Auflage

30. Juni 2010

α-Version

Deutsche Übersetzung von Ralf Wahner

Inhaltsverzeichnis

Teil I

Vorwort und Einleitung

Kapitel 0

Vorwort

Inhaltsübersicht

[0] Als ich anfing, mich mit Java zu beschäftigen, erwartete ich lediglich eine weitere Programmiersprache. Dies trifft in vieler Hinsicht zu.

[1] Nachdem ich mich einige Zeit intensiv mit Java auseinandergesetzt hatte, begann ich zu verstehen, daß sich die grundlegenden Absichten „hinter“ dieser Sprache von den Zielsetzungen anderer Sprachen unterscheiden, die ich bis dahin gesehen hatte.

[2] Programmieren bedeutet das Zurechtkommen mit Komplexität: Damit ist einerseits die Komplexität des zu lösenden Problems, andererseits aber auch die Komplexität der Maschine gemeint, in Bezug auf deren Eigenschaften und Fähigkeiten das Problem gelöst werden soll. Die meisten unserer Softwareentwicklungsprojekte scheitern an dieser Komplexität. Dennoch hat praktisch keine Programmiersprache, die ich kenne, das Designziel, die Komplexität der Softwareentwicklung und -pflege zu überwinden.[1] Natürlich werden viele Design-Entscheidungen bei Programmiersprachen im Hinblick auf die Komplexität getroffen, aber von einem gewissen Punkt an, wird stets die Berücksichtigung anderer Dinge im Sprachumfang als wichtiger angesehen und diese Dinge sind es, durch welche die Programmierer letztendlich und zwangsläufig an die Grenzen der Sprache stoßen. C++ zum Beispiel, sollte sowohl rückwärtskompatibel zu C (um die Migration für C-Programmierer zu erleichtern) als auch effizient sein. Beides sind nützliche Ziele und haben viel zum Erfolg von C++ beigetragen, bedeuten aber auch zusätzliche Komplexität, die den Abschluß manches Projektes verhindert hat. (Sie können die Schuld natürlich bei den Programmierern und Projektleitern suchen. Wenn aber eine Sprache die Programmierer beim Abfangen von Fehlern unterstützen kann, aus welchem Grund sollte sie darauf verzichten?) Visual Basic (VB), um ein weiteres Beispiel zu nennen, war an Basic gebunden, eine Sprache, deren Design sicherlich nicht auf Erweiterungen ausgerichtet war, so daß die zahlreichen Erweiterungen zu einer wahrhaft unbeherrschbaren Syntax geführt haben. Perl ist rückwärtskompatibel mit `awk`, `sed`, `grep` und anderen Unix-Hilfsprogrammen, die es ersetzen sollte und wird häufig beschuldigt, nur-schreibbaren Quelltext zu hinterlassen (das heißt Sie

[1] Python (*http://www.python.com*) kommt dieser Aufgabe in meinen Augen am nächsten.

können den Quelltext nach einiger Zeit nicht mehr lesen). Anderseits konzentierten sich die Design-Entscheidungen bei C++, VB, Perl und anderen Sprachen *zum Teil* ebenfalls auf das Problem der Komplexität, wodurch diese Sprachen bei der Lösung bestimmter Problemtypen beachtliche Erfolge zu verzeichnen haben.

[3] Ich war, nachdem ich mir ein gewisses Verständnis für Java erarbeitet hatte, am meisten davon beeindruckt, daß die Erleichterung der Komplexität für die Programmierer zu den Design-Anforderungen von Sun Microsystems gehört zu haben schien, beinahe als ob sich die Designer dazu bekannt hätten: „Wir interessieren uns dafür, den Zeitaufwand und Schwierigkeitsgrad für die beziehungsweise bei der Entwicklung eines robusten Quelltextes zu reduzieren.“ In der Anfangszeit führte diese Motivation zu langsamen Programmen (dieser Aspekt hat sich mit der Zeit allerdings gebessert), bewirkte aber in der Tat erstaunlich kürzere Entwicklungszeiten, teilweise nur die Hälfte, wenn nicht sogar noch weniger des Zeitaufwandes zum Schreiben eines gleichwertigen Programms in C++. Dies alleine erspart bereits unglaublich viel Zeit und Geld, aber Java bleibt nicht an diesem Punkt stehen, sondern „verpackt“ viele komplizierte und zunehmend wichtige Dinge wie die Threadprogrammierung und die Netzwerkprogrammierung in Spracheigenschaften und -fähigkeiten beziehungsweise in Bibliotheken, die entsprechende Anforderungen bei Bedarf erheblich vereinfachen. Schließlich nimmt sich Java mit Plattformunabhängigkeit, *dynamic code changes* und Sicherheit einiger wirklich großer Komplexitätsprobleme an, von denen Sie jedes einzelne auf Ihrer persönlichen Skala irgendwo zwischen „Hindernis“ und *show-stopper* einordnen können. Trotz der beobachteten Performanzprobleme ist Java vielsprechend: Java ist im Stande, uns zu erheblich produktiveren Programmierern machen.

[4] Java erhöht die Bandbreite der Kommunikation *zwischen* den Beteiligen in jeder Hinsicht: Bei der Entwicklung von Programmen, in der Teamarbeit, beim Aufbau von Benutzerschnittstellen zur Kommunikation mit dem Anwender, beim Betrieb der Programme auf verschiedenen Rechnerarchitekturen und hinsichtlich der einfachen Entwicklung von Programmen, die über das Internet hinweg kommunizieren.

[5] Nach meiner Auffassung sollten wir die Folgen der Kommunikationsrevolution nicht unbedingt an der Bewegung großer Datenmengen ablesen. Die wahre Revolution besteht vielmehr darin, daß wir alle einfacher miteinander kommunizieren können, sowohl einzeln als auch in Gruppen sowie als gesamter Planet. Es gibt eine Vision, die besagt, daß die nächste Revolution das Zusammenwachsen einer Art globalen Gedächtnisses sein wird, bestehend aus genügend Menschen und einem ausreichenden Grad an Verbundenheit miteinander. Vielleicht ist Java das Werkzeug, um diese Revolution anzufachen, vielleicht auch nicht. Die Aussicht hat mir aber das Gefühl vermittelt, etwas bedeutsames zu tun, indem ich versuche, diese Sprache zu lehren.

0.1 Java SE 5 und SE 6

[6] Die vierte Auflage dieses Buches profitiert sehr von den Verbesserungen der Sprache, die von Sun Microsystems ursprünglich als Version 1.5 des Java Development Kits (JDK 1.5), später JDK 5 oder J2SE 5 und schließlich nach Entfernen der „2“ als Version 5 der Java Standard Edition (SE 5) bezeichnet wurde. Viele Änderungen an der SE 5 sollten die Erfahrungen der Programmierer vervollständigen. Wie Sie sehen werden, ist den Sprachdesignern diese Aufgabe nicht vollständig gelungen. Im allgemeinen haben sie allerdings große Schritte in die richtige Richtung vollzogen.

[7] Eines meiner wichtigsten Ziele in dieser Auflage war, die Verbesserungen in der SE 5/6 vollständig zu erfassen, einzuführen und überall im Buch zu verwenden, das heißt diese Auflage wagt den kühnen Schritt, die SE 5/6 vorauszusetzen. Ein großer Teil der Beispiele läßt sich mit früheren Java-Versionen nicht übersetzen. Ant bricht die Übersetzung mit einer Fehlermeldung ab, wenn Sie

es trotzdem versuchen. Ich bin dennoch der Meinung, daß die Vorteile das Risiko aufwiegen.

[8] Wenn Sie an eine frühere Java-Version gebunden sind, finden Sie die Grundzüge in den früheren Auflagen dieses Buches, die Sie unter der Webadresse *http://www.mindview.net* zum kostenlosen Herunterladen finden. Ich habe mich aus verschiedenen Gründen entschieden, die aktuelle Auflage des Buches nicht in einer kostenlosen elektronischen Version anzubieten, sondern nur die früheren Auflagen.

0.1.1 Java SE 6

[9] Dieses Buch war ein gewaltiges, zeitaufwändiges Projekt. Die Betaversion der SE 6 (Codename „Mustang“) erschien vor der Herausgabe. Einige kleinere Änderungen an der SE 6 konnten verwendet werden, um einige der Beispiele zu verbessern. Die meisten Änderungen an der SE 6 hatten jedoch keine Auswirkungen auf den Inhalt des Buches. Die Änderungen betrafen hauptsächlich Geschwindigkeitsverbesserungen sowie Eigenschaften und Fähigkeiten der Standardbibliothek außerhalb der Reichweite dieses Textes.

[10] Die Quelltextdistribution zu diesem Buch wurde mit einem Release Candidate der SE 6 getestet so daß ich nicht erwartet, daß sich Änderungen auf den Inhalt dieses Buches auswirken. Falls sich bis zur Herausgabe der SE 6 eine wesentliche Änderung ergibt, wird sie in der Quelltextdistribution berücksichtigt, die Sie unter der Webadresse *http://www.mindview.net* zum Herunterladen finden.

[11] Der Einband weist aus, daß dieses Buch für Java SE 5/6 geschrieben wurde. Der Hinweis bedeutet, daß das Buch im Hinblick auf die SE 5 und die sehr deutlichen Änderungen geschrieben wurde, die unter dieser Version in der Sprache eingeführt wurden, aber gleichermaßen für die SE 6 geeignet ist.

0.2 Die vierte Auflage dieses Buches

[12] Die befriedigende Wirkung der Arbeit an einer neuen Auflage eines Buches, kommt durch das „Berichtigen“ der Dinge vor dem Hintergrund dessen zustande, was ich seit dem Erscheinen der letzten Auflage gelernt habe. Diese Einsichten entsprechen häufig der Redensart „Eine Lernerfahrung ist, wenn Sie nicht das bekommen, was Sie erwartet haben.“ Ebenso oft bringt die Arbeit an der nächsten Auflage aber faszinierende neue Ideen hervor und das Gefühl von Verlegenheit wird von der Entdeckungsfreude und dem Vermögen überstrahlt, eine Idee auf eine gelungenere Art und Weise ausdrücken zu können, als beim letzten Mal.

[13] Es ist außerdem eine Herausforderung, die neue Auflage zu einem Buch zu machen, daß die Leser der früheren Auflagen kaufen (und lesen) wollen. Dies zwingt mich, soviel wie möglich zu verbessern, umzuschreiben und neu zu organisieren, damit das Buch zu einer neuen und wertvollen Erfahrung für diese Leser wird, die mit Leib und Seele dabei sind.

0.2.1 Änderungen

[14] Die tradionelle CD der früheren Auflagen gehört nicht mehr zur vierten Auflage. Das Multimediaseminar *Thinking in C*, der wichtigste Inhalt der CD, für Mindview zusammengestellt von Chuck Allison, kann nun als Flashpräsentation heruntergeladen werden. Ziel dieses Seminars ist, Leser vorzubereiten, die noch nicht hinreichend mit der C-Syntax vertraut sind, um dieses Buch verstehen zu können. Zwei Kapitel des Buches geben zwar eine anständige Einführung in die Syntax, mögen

aber für Leser ohne das erforderliche Hintergrundwissen nicht ausreichen. Das Multimediaseminar soll diesen Lesern auf das erforderliche Niveau verhelfen.

[15] Kapitel 22 „Threadprogrammierung“ (in der englischen Ausgabe „Concurrency“, zuvor „Multithreading“) wurde zur Anpassung an die größeren Änderungen in den Threadpackages der Standardbibliothek der SE 5 vollständig neu geschrieben, vermittelt aber nach wie vor die Grundlagen der Kernideen der Threadprogrammierung. Ohne diese Grundlagen ist es schwer, die komplexeren Aspekte der Threadprogrammierung zu verstehen. Ich habe viele Monate an diesem Kapitel gearbeitet und mich in das Gebiet der Threadprogrammierung vertieft. Letztendlich bietet das Kapitel nicht nur eine Einführung in die Grundlagen, sondern wagt sich auch an anspruchsvollere Dinge heran.

[16] Zu jeder signifikaten neuen Eigenschaft oder Fähigkeit der SE 5 gibt es ein neues Kapitel. Die übrigen Neuerungen sind in den bereits vorhandenen Text eingeflochten. Aufgrund meines kontinuierlichen Studiums von Entwurfsmustern, werden überall im Buch auch mehr Entwurfsmuster vorgestellt.

[17] Das Buch hat eine deutliche Neuorganisation durchlaufen, die sich zu einem beträchtlichen Teil durch meine Lehrtätigkeit, kombiniert mit der Einsicht ergeben hat, daß meine Auffassung dessen, was ein „Kapitel“ ist vielleicht einmal neu durchdacht werden könnte. Ich war stets bedenkenlos der Ansicht zugeneigt, daß ein Thema eine gewisse Größe haben sollte, um ein separates Kapitel zu rechtfertigen. Gerade bei Schulungen über Entwurfsmuster habe ich aber beobachtet, daß die Teilnehmer am besten mitkommen, wenn der Vorstellung eines einzelnen Entwurfsmusters unmittelbar eine Übungsaufgabe folgt, auch wenn ich dadurch weniger Sprechzeit habe (ich habe entdeckt, daß diese Vorgehensweise auch für mich als Lehrer angenehmer ist.) Ich habe daher in dieser Auflage versucht, die Kapitel thematisch aufzuteilen und mich nicht um die Länge der Kapitel zu kümmern. Ich glaube, daß das Buch hierdurch besser geworden ist.

[18] Darüber hinaus habe ich die Wichtigkeit des Testens erkannt. Ohne eingebaute Testumgebung mit Tests, die jedesmal ablaufen, wenn Sie eine Anwendung übersetzen, haben Sie keine Chance, zu wissen, ob Sie sich auf Ihren Code verlassen können oder nicht. Um einen solche Prüfmechanismus für dieses Buch zu entwerfen, habe ich eine Testumgebung entwickelt, die die Ausgabe jedes Programms anzeigt und validiert. (Die Testumgebung ist in Python geschrieben und gehört zur Quelltextdistribution zu diesem Buch, die Sie unter der Webadresse *http://www.mindview.net* herunterladen können.) Das Thema Testen wird im Anhang von *http://www.mindview.net/Books/BetterJava* beschrieben. Dort ist alles an grundlegenden Fertigkeiten zusammengefaßt, was aus meiner jetzigen Überzeugung zur Grundausstattung im Werkzeugkasten eines Programmierers gehört.

[19] Außerdem habe ich mich bei jedem einzelnen Beispiel in diesem Buch gefragt, warum ich es auf diese Art und Weise entwickelt habe. In den meisten Fällen habe ich einige Änderungen und Verbesserungen angebracht, einerseits, um der Konsistenz der Beispiele untereinander willen und andererseits, um zu demonstrieren, was aus meiner Sicht die beste Java-Lösung ist (zumindest in den Grenzen eines einführenden Buches). Viele bereits vorhandene Beispiele haben ein erhebliches Redesign und eine Neuimplementierung erfahren. Beispiele wurden entfernt, wenn sie nicht länger sinnvoll waren und neue Beispiele wurden aufgenommen.

[20] Leser haben viele, viele wunderbare Kommentare über die ersten drei Auflagen dieses Buches geschrieben, die mir natürlich gut getan haben. Dennoch gibt es ab und zu auch Beschwerden. Eine periodisch wiederkehrende Beschwerde lautet: „Das Buch ist zu dick“. Es ist in meinen Augen ein von Furchtsamkeit geprägtes Urteil, wenn sich die Kritik tatsächlich nur auf „zu viele Seite“ bezieht. (Man fühlt sich an die Beanstandung des österreichischen Kaisers, Mozarts Werk gegenüber erinnert: „Zu viele Noten!“ Nicht, daß ich mich in irgendeiner Weise mit Mozart vergleichen möchte.) Ich kann nur davon ausgehen, daß eine solche Beschwerde von jemandem stammt, der noch nichts von der

unermeßliche Weite von Java weiß und den Rest der Bücher zu diesem Thema noch nicht gesehen hat. Nichtsdestotroz habe ich in dieser Auflage versucht, veraltete oder zumindest unwesentliche Dinge zu streichen. Im allgemeinen habe ich versucht, jedes Ding zu betrachten, zu entfernen, was nicht länger notwendig war, Änderungen aufzunehmen und soviel wie möglich zu verbessern. Es ist aus meiner Sicht in Ordnung, Teile herauszunehmen, da das ursprüngliche Material nach wie vor in Form der ersten drei Auflagen, zusammen mit den ~~Online-Anhängen~~ zur vierten Auflage unter meiner Webadresse (*http://www.mindview.net*) kostenlos zum Herunterladen zur Verfügung steht.

[21] Ich entschuldige mich bei denen, die den Umfang des Buches nicht ertragen können. Ich habe hart daran gearbeitet, den Umfang in Grenzen zu halten, ob Sie es glauben oder nicht.

0.3 Anmerkung zum Entwurf des Einbandes

[22] Die Gestaltung des Einbands von *Thinking in Java* ist durch die „American Arts and Crafts"-Bewegung inspiriert, die zu Beginn des vorigen Jahrhunderts ins Leben gerufen wurde und ihren Höhepunkt zwischen 1900 und 1920 erreichte. Die Bewegung entstand in England als Reaktion auf die maschinelle Produktion infolge der industriellen Revolution und den reich verzierten Stil der viktorianischen Zeit. Die „American Arts and Crafts"-Bewegung legte Wert auf sparsame Erscheinungsform, die Formen der Natur wie im Jugendstil, das Handwerk und das Ansehen einzelner Handwerker, ohne die Verwendung moderner Hilfsmittel zu meiden. Es gibt zahlreiche Parallelen zu unserer heutigen Situation: Der Übergang in ein neues Jahrhundert, die Entwicklung von den Anfängen des Computerzeitalters ~~to something more refined and meaningful~~ und die Betonung der Kunstfertigkeit der Softwareentwicklung, statt einfach Quelltext zu schreiben.

[23] Ich betrachte Java auf dieselbe Weise: Als einen Versuch, den Programmierer über den Status eines Betriebssystemmechanikers und hin zu einem „Softwarehandwerker" zu erheben.

[24] Der Autor und der Designer des Einbands (sind seit Kindertagen Freunde) empfangen Inspiration von der „American Arts and Crafts"-Bewegung und besitzen beide Möbelstücke, Lampen und weitere Gegenstände die entweder original aus dieser Zeit stammen oder davon inspiriert wurden.

[25] Der Einband dieses Buches soll an einen Setzkasten erinnern, den ein Naturforscher beispielsweise dazu verwendet, seine präparierten Käfer auszustellen. Die Käfer sind Objekte und werden in das Setzkasten-Objekt eingeordnet, das wiederum in das Einband-Objekt eingesetzt wird, wodurch das fundamentale Konzept der Aggregation in der objektorientierten Programmierung veranschaulicht wird. Ein Programmierer kann den Käfern natürlich nicht helfen, aber den Zusammenhang mit Fehlern (*bugs*) erkennen. Die Käfer werden gefangen, vermutlich in einem Probenglas getötet und schließlich in einem kleinen Plexiglaskästchen eingesperrt, wie um die Fähigkeit von Java, Fehler zu finden, anzuzeigen und zu bewältigen (in der Tat eines der leistungsfähigsten Merkmale von Java), stillschweigend anzudeuten.

[26] Das Bild in Wasserfarben auf dem Einband, habe ich für diese Auflage gemalt.

0.4 Danksagung

[27] Mein Dank gilt an erster Stelle den Gefährten, die mit mir zusammengearbeitet haben, um Schulungen durchzuführen, Beratung anzubieten und Unterrichtsprojekte zu entwickeln: Dave Bartlett, Bill Venners, Chuck Allison, Jeremy Meyer und Jamie King. Ich schätze Eure Geduld, während ich fortwährend versuche, das beste Modell zu finden, nach dem unabhängige Leute wie wir zusammenarbeiten können.

[28] In letzter Zeit stehe ich, unzweifelhaft aufgrund des Internets, mit einer überraschend großen Anzahl von Menschen in Kontakt, die mich, in der Regel von zu Hause aus, bei meinen Unternehmungen unterstützen. Früher hätte ich ziemlich viel Bürofläche mieten müssen, um all diese Kollegen unterzubringen, aber durch Internet, FedEx und Telefon kann ich ohne zusätzliche Kosten von ihrer Hilfe profitieren. Sie alle haben mir bei meinen Lernversuchen, to play well with others, sehr geholfen und ich hoffe, auch weiterhin lernen zu können, meine Arbeit mit Hilfe der Anstrengungen anderer Menschen zu verbessern. Paula Steuer hat unschätzbar Wertvolles geleistet, indem sie sich meiner planlosen Geschäftspraktiken angenommen und sie vernünftig gemacht hat (Danke, daß Sie mir einen Stoß gegeben haben, wenn ich keine Lust hatte etwas zu tun, Paula). Jonathan Wilcox hat die Struktur meines Unternehmens untersucht, jeden Stein umgedreht, that might hide scorpions, und uns durch den Prozeß geschleift hat, alles juristisch untadelig zu machen. Danke für Ihre Sorgfalt und Beharrlichkeit. Sharlynn Cobaugh ist zu einer Expertin in Sachen sound processing worden, hat einen beträchtlichen Anteil of creating the multimedia training experiences und eine Reihe weiterer Probleme in Angriff genommen. Die Leute von Amaio in Prag sind bei einigen Projekten für mich eingesprungen. Daniel Will-Harris war meine ursprüngliche Inspiration für das Arbeiten im Internet und he is of course fundamental to all my graphic design solutions.

[29] Gerald Weinberg ist im Laufe der Jahre durch seine Konferenzen und Workshops zu meinem inoffiziellen Betreuer und Mentor geworden, wofür ich ihm sehr dankbar bin.

[30] Ervin Varga war bei technischen Korrekturen an der vierten Auflage besonders hilfsbereit. Obwohl auch andere Kollegen bei verschiedenen Kapiteln und Beispielen geholfen haben, war Ervin mein wichtigster technischer Lektor für dieses Buch und hat auch die Aufgabe übernommen, den „Solution Guide“ für die vierte Auflage neu zu schreiben. Ervin hat Fehler entdeckt und Verbesserungsvorschläge gemacht, die das Buch unschätzbar bereichert haben. Seine Gründlichkeit und Aufmerksamkeit in Einzelheiten sind faszinierend und er ist mit Abstand der beste technische Lektor, den ich jemals hatte. Danke, Ervin.

[31] Mein Weblog auf Bill Venners *http://www.artima.com* war eine Quelle der Unterstützung, wenn ich needed to bounce ideas around. Ich danke den Lesern, die mir durch ihre Kommentare geholfen haben, Konzepte zu klären, darunter James Watson, Howard Lovatt, Michael Barker und weitere, insbesondere diejenigen, die mir beim Thema „Generische Typen“ geholfen haben.

[32] Danke an Mark Welsh für seine kontinuierliche Unterstützung.

[33] Evan Cofsky unterstützt mich nach wie vor, indem er aus dem hohlen Bauch heraus sämtliche verborgenen Einzelheiten über Konfiguration, Betrieb und Pflege Linux-basierter Webserver weiß, den Mindview-Server abstimmt und für seine Sicherheit sorgt.

[34] Ein besonderer Dank an meinen neuen Freund, den Kaffee, der bei diesem Projekt für einen beinahe grenzenlosen Enthusiasmus gesorgt hat. Das „Camp4 Coffee“ in Crested Butte (Colorado) ist zum Lieblingstreff der Leute geworden, die bei Mindview Schulungen besuchen und bietet bei Schulungen den besten Catering-Service an, den ich je hatte. Danke an meinen Kumpel Al Smith, daß er das „Camp4 Coffee“ eröffnet und zu einem so großartigen Lokal und interessanten und unterhaltsamen Teil des Lebens in Crested Butte gemacht hat. Danke auch an die Barristas im „Camp4 Coffee“ dafür, daß sie so frohgemut sparsame Getränkeportionen austeilen.

[35] Danke an die Leute von Prentice Hall dafür, daß sie mir stets geben, was ich möchte, alle meine Spezialwünsche berücksichtigen und keine Mühe scheuen, damit für mich alles glatt läuft.

[36] Einige Hilfsmittel haben sich während meiner Entwicklungsphase als außerordentlich wertvoll erwiesen und ich bin ihren Schöpfern jedesmal sehr dankbar, wenn ich sie verwende. Cygwin (*http://www.cygwin.com*) hat mir die Lösung zahlloser Probleme ermöglicht, mit denen Windows nicht umgehen kann oder will und ich fühle mich Cygwin täglich mehr verbunden (hätte ich Cyg-

win nur schon fünfzehn Jahre früher gehabt, als mein Hirn noch fest mit GNU-Emacs verdrahtet war). Die Entwicklungsumgebung Eclipse von IBM (*http://www.eclipse.org*) ist ein wahrhaft wundervolles Geschenk für die Entwicklergemeinde und ich stelle große Erwartungen an ihre weitere Entwicklung. (Wie hat es IBM auf die Höhe der Zeit geschafft? Ich muß eine Meldung versäumt haben.) IntelliJ Idea von JetBrains erfindet fortwährend neue Wege für Entwicklungswerkzeuge.

[37] Ich habe bei diesem Buch begonnen, Enterprise Architect von Sparxsystems zu nutzen und es ist schnell zum UML-Werkzeug meiner Wahl geworden. Der Jalopy-Quelltextformatierer von Marco Hunsicker (*http://www.triemax.com*) hat sich bei vielen Gelegenheiten als nützlich erwiesen und Marco war sehr hilfsbereit bei der Konfiguration des Programms an meine Bedürfnisse. Slava Pestovs JEdit (*http://www.jedit.org*) hat sich mit seinen Plugins gelegentlich als nützlich erwiesen und ist ein vernünftiger Anfängereditor für Schulungen.

[38] Und natürlich, wenn ich nicht schon überall sonst darauf hinweise, arbeitet ich, um Probleme zu lösen, stets mit Python (*http://www.python.com*), dem geistigen Kind meines Kumpels Guido van Rossum und seiner Bande alberner Genies, mit denen ich einige schöne Tage bei *sprinting* verbracht habe (Tim Peters, *I've now framed that mouse you borrowed, officially named the "TimBotMouse"*). Ihr solltet Euch gesündere Plätze zum Mittagessen aussuchen. (Ich danke auch der gesamte Python-Gemeinde, ein unglaublicher Haufen.)

[39] Viele Leser haben Fehler gefunden und Korrekturvorschläge geschickt. Ich stehe in ihrer aller Schuld, aber besonderer Dank gilt für die erste Auflage: Kevin Raulerson (hat viele größere Fehler entdeckt), Bob Resendes (einfach unglaublich), John Pinto, Joe Dante, Joe Sharp (alle drei sagenhaft), David Combs (viele Korrekturen zur Grammatik und Klarheit), Dr. Robert Stephenson, John Cook, Franklin Chen, Zev Griner, David Karr, Leander A. Stroschein, Steve Clark, Charles A. Lee, Austin Maher, Dennis P. Roth, Roque Oliveira, Douglas Dunn, Dejan Ristic, Neil Galarneau, David B. Malkovsky, Steve Wilkinson und viele andere. Prof. Ir. Marc Meurrens hat sich sehr darum bemüht, die elektronische Version der ersten Auflage in Europa zu veröffentlichen und verfügbar zu machen.

[40] Ich danke allen, die mir geholfen haben, die Anwendungsbeispiele zur Swing-Bibliothek in der zweiten Auflage neu zu schreiben: Jon Shvarts, Thomas Kirsch, Rahim Adatia, Rajesh Jain, Ravi Manthena, Banu Rajamani, Jens Brandt, Nitin Shivaram, Malcolm Davis und alle übrigen, die mich unterstützt haben.

[41] Chris Grindstaff war beim Entwickeln von Abschnitt 23.14 „Das Standard Widget Toolkit (SWT)“ in der vierten Auflage sehr hilfbereit. Sean Neville hat den ersten Entwurf von Abschnitt 23.13 „Webbrowserbasierte Flex-Clients“ für mich geschrieben.

[42] Kraig Brockschmidt und Gen Kiyooka sind zwei geschickte Techniker, die meine Freunde geworden sind. Beide sind ebenso einflußreich wie ungewöhnlich, indem sie Yoga und andere Formen spiritueller Erfahrung praktizieren, die ich sowohl als inspirativ als auch als lehrreich empfinde.

[43] Jedesmal wenn ich glaube, die Threadprogrammierung verstanden zu haben, öffnet sich eine weitere Tür und es gilt, einen neuen Berg zu besteigen. Danke an Brian Goetz für seine Hilfe bei allen Hürden im neuen Kapitel über Threadprogrammierung und dafür, daß er alle Fehler gefunden hat (hoffentlich).

[44] Es war nicht besonders überraschend für mich, daß mir das Verständnis von Delphi beim Verstehen von Java geholfen hat, da beide Sprachen viele Konzepte und Entwurfs-Entscheidungen gemeinsam haben. Meine Delphi-Freunde haben mir geholfen, Einsicht in diese ausgezeichnete Programmiersprache zu nehmen: Marco Cantu (wieder ein Italiener; von Latein durchdrungen zu sein, begünstigt vielleicht das Geschick für Programmiersprachen), Neil Rubenking (der sich mit Yoga, vegetarischer Ernährung und Zen beschäftigt hatte, bis er Computer entdeckte) und natürlich Zack

Urlocker (der ursprüngliche Produktmanager von Delphi) ein langjähriger Freund, mit dem ich die Welt bereist habe. Wir alle stehen in der Schuld von Anders Hejlsberg und seines Glanzes, der sich fortwährend mit C# plagt (einer wesentlichen Inspiration der SE 5, wie Sie in diesem Buch lernen werden).

[45] Die Erkenntnisse und die Unterstützung meines Freundes Richard Hale Shaw (ebenso wie Kims) haben mir sehr geholfen. Richard und ich haben monatelang zusammen Schulungen veranstaltet und versucht, die bestmögliche Lernerfahrung für die Teilnehmer zu erarbeiten.

[46] Die Gestaltung des Buches, des Einbandes und das Photo auf dem Einband stammen von meinem Freund Daniel Will-Harris, renommierter Autor und Designer (*http://www.will-harris.com*), der in der Mittelstufe mit *rub-on letters* gespielt hat, während er auf die Erfindung des Computers und des Desktop-Publishings gewartet und sich über mich beschwert hat, wenn ich murmelnd über meinen Rechenaufgaben saß. Die druckfertigen Seiten stammen allerdings von mir, das heißt für die Druckfehler bin ich verantwortlich. Ich habe Microsoft Word XP für Windows verwendet, um das Buch zu schreiben und druckfertige Seiten per Adobe Acrobat erzeugt. Das Buch wurde direkt aus den von Acrobat generierten PDF-Dateien gedruckt. In Anerkennung des elektronischen Zeitalters war ich in Übersee, als ich die Schlußversion der ersten und zweiten Auflage schrieb. Die erste Auflage wurde von Kapstad in Südafrika aus gesendet, die zweite Auflage aus Prag. Die dritte und vierte Auflage stammte aus Crested Butte (Colorado). Der Textkörper ist in *Georgia* gesetzt, die Überschriften in *Verdana*. Der Zeichensatz auf dem Einband ist *ITC Rennie Mackintosh*.

[47] Ein besonderer Dank gilt all meinen Lehrern und Studenten (die ebenfalls meine Lehrer sind).

[48] Molly, unsere Katze war häufig in meinem Arbeitszimmer, während ich an dieser Auflage schrieb und hat auf mich auf ihre freundliche und schnurrende Weise unterstützt.

[49] Das Ensemble unterstützender Freund umfaßt, ist aber keinesfalls begrenzt auf: Patty Gast (eine einzigartige Masseuring), Andrew Binstock, Steve Sinofsky, JD Hildebrandt, Tom Keffer, Brian McElhinney, Brinkley Barr, Bill Gates beim „Midnight Engineering Magazine“, Larry Constantine and Lucy Lockwood, Gene Wang, Dave Mayer, David Intersimone, Chris and Laura Strand, die Almquists, Brad Jerbic, Marilyn Cvitanic, Mark Mabry, die Familien Robbins, Moelters und McMillans, Michael Wilk, Dave Stoner, die Cranstons, Larry Fogg, Mike Sequeira, Gary Entsminger, Kevin und Sonda Donovan, Joe Lordi, Dave und Brenda Bartlett, Patti Gast, Blake, Annette und Jade, die Rentschlers, die Sudeks, Dick, Patty, and Lee Eckel, Lynn und Todd mit ihren Familien. Und natürlich meine Eltern.

Kapitel 1

Einleitung

Inhaltsübersicht

[0] „He gave man speech, and speech created thought, Which is the measure of the Universe“
Percy Bysshe Shelley, Prometheus Unbound

Übersetzt etwa[1]: „Dem menschlichen Geschlecht gab er die Sprache Und aus der Sprache rang sich der Gedanke, Er, der das Maß des Universums ist!“

[1] „Human beings ... are very much at the mercy of the particular language which has become the medium of expression for their society. It is quite an illusion to imagine that one adjusts to reality essentially without the use of language and that language is merely an incidental means of solving specific problems of communication and reflection. The fact of the matter is that the "real world" is to a large extent unconsciously built up on the language habits of the group.“
(The Status of Linguistics as a Science, 1929, Edward Sapir)

Übersetzt etwa: „[Lücke: Übersetzung fehlt noch.] “

[2] Java ist, wie eine menschliche Sprache, ein Mittel, um Konzepte auszudrücken. Bei Erfolg ist dieses Ausdrucksmedium deutlich einfacher und viel flexibler als seine Alternativen, wenn die Probleme größer und komplexer werden.

[3] Sie können Java nicht als eine Ansammlung von Eigenschaften und Fähigkeiten betrachten, da einige in Isolation nicht sinnvoll sind. Sie können die Summe der Teile nur dann nutzen, wenn Sie sich über Design Gedanken machen, statt nur über das alleinige Programmieren. Auf diese Weise geartetes Verständnis von Java setzt voraus, daß Sie the problems with the language and with

[1] Anmerkung des Übersetzers: Quelle: Shelley, Percy Bysshe: Der entfesselte Prometheus. Wien 1876, S. 50-58 (*http://www.zeno.org/Literatur/M/Shelley,+Percy+Bysshe*).

programming im allgemeinen verstehen. Dieses Buch diskutiert Programmierprobleme, warum sie Probleme sind und welchen Ansatz Java gewählt hat, um sie zu lösen. Die Auswahl von Eigenschaften und Fähigkeiten, die ich in den einzelnen Kapiteln erkläre, fußt auf meiner Perspektive der Frage, wie ein bestimmter Problemtyp mit Java gelöst wird. Auf diese Weise hoffe ich, Sie Stück für Stück auf den Punkt zubewegen zu können, an dem die für Java spezifische Denkweise zu ihrer „Muttersprache“ wird.

[4] Ich setze, über das gesamte Buch hinweg, als Ihre innere Haltung voraus, daß Sie im Kopf ein Modell aufbauen wollen, das Ihnen gestattet, ein tiefgreifendes Verständnis der Sprache zu entwickeln, so daß Sie ein Puzzleteil, wenn Sie es finden, in Ihr Modell einfügen und sich die Antwort selbst herleiten können.

1.1 Voraussetzungen

[5] Dieses Buch setzt voraus, daß Sie in einem gewissen Umfang bereits mit dem Programmieren vertraut sind: Sie verstehen ein Programm als eine Ansammlung von Anweisungen, die Idee der Subroutine, Funktion beziehungsweise des Makros, Verzweigungsanweisungen wie `if`, Schleifenkonstruktionen mit `while` und so weiter. Es gibt viele Möglichkeiten, diese Dinge zu lernen, etwa das Programmieren in einer Makrosprache oder die Arbeit mit einem Werkzeug wie Perl. Wenn Sie beim Programmieren den Punkt erreicht haben, an dem Sie sich mit diesen Grundideen der Programmierung sicher fühlen, sind Sie in Lage dieses Buch durchzuarbeiten. Natürlich ist das Buch für C-Programmierer leichter, umso mehr für C++-Programmierer, aber schließen Sie sich nicht aus, wenn Sie keine Erfahrung mit diesen Sprachen haben. Seien Sie aber bereit, hart zu arbeiten. Das Multimediaseminar *Thinking in C* wird Ihnen außerdem helfen, die zum Erlernen von Java benötigten Grundlagen zügig zu verinnerlichen. Dennoch stelle ich die Konzepte der objektorientierten Programmierung (OOP) und die grundlegenden Schleifen- und Verzweigungsanweisungen von Java vor.

[6] Gelegentliche Verweise auf Eigenschaften oder Fähigkeiten von C oder C++ sind nicht als Kommentare für Eingeweihte gedacht, sondern sollen allen Programmierern ermöglichen, Java aus der Perspektive dieser beiden Sprachen zu betrachten, deren Nachfolger Java schließlich ist. Ich habe versucht, diese Verweise einfach zu gestalten und alles zu erklären, womit ein Programmierer ohne Erfahrung in C/C++ nach meiner Einschätzung nicht vertraut ist.

1.2 Erlernen der Sprache Java

[7] Etwa um dieselbe Zeit, als mein erstes Buch *Using C++* bei Osborne/McGraw-Hill (1989) erschien, habe ich begonnen, C++ zu unterrichten. Das Unterrichten von Programmierideen ist mein Metier geworden und ich habe seit 1987 bei meinen Zuhörern überall auf der Welt nickende Köpfe, leere Gesichter und beunruhigte Mienen gesehen. Als ich meine ersten unternehmensinternen Schulungen mit kleineren Teilnehmerzahlen abhielt, fiel mir während der Übungsaufgaben etwas auf. Selbst die Anwesenden, die zuvor gelächelt und mit dem Kopf genickt hatten, waren über vieles verwirrt. Durch die Organisation und den Vorsitz des C++-Zuges bei der „Software Development Conference“ über mehrere Jahre hinweg (später Organisation und den Vorsitz des Java-Zuges), fand ich heraus, daß ich und die anderen Redner für den typischen Teilnehmerkreis tendentiell zuviele Themen zu schnell behandelten. Letztendlich würde ich, sowohl durch die Bandbreite der Zuhörer als auch durch die Art meiner Präsentation, einen Teil des Auditoriums „abhängen“. Vielleicht verlange ich zuviel, aber da ich zu den Leuten gehöre, die gegen traditionelles Vortrage resistent sind (und

ich glaube, daß diese Resistenz bei den meisten Menschen vom Stumpfsinn verursacht wird), wollte ich versuchen, alle bei der Stange zu halten.

[8] Eine Zeit lang arbeitete ich an mehreren verschiedenen Präsentationen in fairly short order. Ich lernte schließlich durch Versuch und Wiederholung (eine Vorgehensweise, die auch beim Design von Programmen gut funktioniert). Letztendlich entwickelte ich einen Kurs aus allem, was ich aus meiner Lehrerfahrung gelernt hatte. Meine Firma Mindview Inc. bietet diesen Kurs mittlerweile öffentlich oder betriebsintern als Schulung *Thinking in Java* an. Dies ist unsere Haupteinführungsschulung, die die Grundlage für unsere fortgeschrittenen Schulungen schafft. Sie finden die Einzelheiten unter der Webadresse *http://www.mindview.net*. (Die Einführungsschulung ist auch als *Hands-On Java* CD-Rom erhältlich. Informationen dazu unter derselben Webadresse.

[9] Die Rückmeldungen aus jeder einzelnen Schulung helfen mir, das Material zu überarbeiten und mich neu zu konzentieren, bis ich denke, daß es als Unterrichtsmedium gut funktioniert. Aber Dieses Buch ist mehr als Schulungsnotizen. Ich habe versucht, soviel Information wie möglich auf den Seiten unterzubringen und so zu strukturieren, daß Sie in das nächste Gebiet hineingezogen werden. Das Buch ist vor allem dazu entworfen, dem einzelgängerischen Leser zu dienen, der sich mit einer neuen Programmiersprache auseinandersetzt.

1.3 Ziele

[10] Der Entwurf dieses Buches dient demselben Ziel, wie der Entwurf meines vorigen Buches *Thinking in C++*: Der Art und Weise wie Menschen eine Programmiersprache lernen. Wenn ich mir ein Kapitel dieses Buches vorstelle, überlege ich mir, was eine gute Lektion in einer Schulung ausmacht. Die Rückmeldungen der Schulungsteilnehmer helfen mir, zu begreifen, welches die schwierigen Dinge sind, die der Beleuchtung bedürfen. Durch das Präsentieren des Lehrstoffes bei Gebieten, die meinen Ehrgeiz anstacheln und mich dazu treiben, zuviele Eigenschaften und Fähigkeiten auf einmal zu behandeln, habe ich erkannt, daß Sie, wenn Sie viele neue Dinge einarbeiten möchten, diese auch alle erklären müssen und dies trägt leicht zur Verwirrung der Lernenden bei.

[11] In jedem Kapitel versuche ich, eine Eigenschaft oder Fähigkeit oder einige wenige zusammenhängende Eigenschaften oder Fähigkeiten darzustellen, ohne Konzepte vorauszusetzen, die noch nicht eingeführt wurden. Auf diese Weise können Sie jeden Baustein im Kontext Ihrer gegenwärtigen Kenntnisse verinnerlichen, bevor Sie fortfahren.

[12] Meine Ziele in diesem Buch sind:

1. Beim Vorstellen des Stoffes stets nur einen Schritt auf einmal zu gehen, so daß Sie jeden Gedanken einfach nachvollziehen können, bevor Sie weiterarbeiten. Sorgfalt bei der Reihenfolge, in der die Eigenschaften und Fähigkeiten vorgestellt werden, so daß Sie mit einer Sache vertraut gemacht werden, bevor Sie ihre Anwendung sehen. Dies ist natürlich nicht immer möglich. In diesen Fällen gibt es eine kurze einführende Beschreibung.

2. So einfache und kurze Beispiele wie möglich zu verwenden. Das hindert mich zwar gelegentlich daran, Probleme aus dem „richtigen Leben“ in Angriff zu nehmen, ich habe allerdings festgestellt, daß sich Anfänger in der Regel wohler fühlen, wenn sie jedes Detail eines Beispiels verstehen, statt sich von den Entfaltungsmöglichkeiten des gelösten Problems beeindrucken zu lassen. Es gibt außerdem eine harte Grenze hinsichtlich der Menge an Quelltext, die die Teilnehmer einer Schulung aufnehmen können. Ich zweifele nicht daran, daß ich für die Verwendung von „Spielzeugbeispielen“ Kritik werde einstecken müssen, bin aber bereit, diese Kritik, zugunsten des pädagogisch sinnvollen Konzeptes hinzunehmen.

3. Ihnen zu vermitteln, was aus meiner Sicht für Sie wichtig ist, um die Sprache zu verstehen, statt allem was ich weiß. Ich glaube, daß es eine Hierarchie der Informationen bezüglich ihrer Wichtigkeit gibt und daß es einige Tatsachen gibt, die 95% der Programmierer niemals zu wissen brauchen, Einzelheiten, die die Leute nur durcheinanderbringen und ihre Wahrnehmung von der Komplexität der Sprache verdeutlichen. Wenn Sie, um ein Beispiel aus C zu wählen, die Tabelle über den Operatorvorrang auswendig lernen (ich habe das nie getan), können Sie raffinierte Programme schreiben. Wenn Sie aber selbst bereits über Ihre Verwendung einiger Operatoren nachdenken müssen, verwirren Sie auch den Leser oder Wartungsprogrammierer des Quelltextes. Vergessen Sie also die Regeln über Operatorvorrang und verwenden Sie Klammern, wenn eine Situation unklar ist.
4. Jeden Abschnitt so zu bündeln, daß die Sprechzeit und die Zeit zwischen den Übungsaufgaben gering ist. Dadurch bleiben nicht nur die Köpfe der Teilnehmer praktischer Übungen frischer und konzentierter bei der Sache, sondern dem Leser wird auch ein stärkeres Gefühl vermittelt, etwas erreicht zu haben.
5. Sie mit soliden Grundlagen auszustatten, damit Sie die Dinge gut genug verstehen können, um sich schwierigeren Aufgaben und Büchern zuwenden zu können.

1.4 Unterrichten nach diesem Buch

[13] Die erste Auflage dieses Buches ist aus einer einwöchigen Schulung hervorgegangen, in den Kindertagen von Java genügend Zeit, um die Sprache abzudecken. Als Java wuchs und fortwährend immer mehr Eigenschaften, Fähigkeiten und Bibliotheken beinhaltete, versuchte ich hartnäckig, dennoch alles in einer Woche zu unterrichten. Irgendwann forderte mich ein Kunde auf, „nur die Grundlagen“ zu lehren. Als ich dieser Aufforderung folgte, entdeckte ich, daß der Versuch, alles in eine einzige Woche zu „stopfen“, für mich, ebenso wie für die Schulungsteilnehmer, zu anstrengend geworden war. Java war keine „einfache“ Sprache mehr, die sich in einer Woche unterrichten ließ.

[14] Diese Erfahrung und Erkenntnis haben mich zu einer Neustrukturierung dieses Buches veranlaßt. Das Buch ist als Unterbau einer zweiwöchigen Schulung beziehungsweise eines zweisemestrigen Kurse gedacht. Der Einführungsteil endet mit Kapitel 13 „Fehlerbehandlung mit Ausnahmen“. Eventuell möchten Sie die Grundlagen mit einer Einführung in JDBC, Servlets und JavaServer Pages ergänzen. Dies ergibt zusammen einen Grundlagenkurs und ist der Inhalt der CD *Hands-On Java*. Der Rest des Buches umfaßt einen Fortgeschrittenenkurs und ist der Inhalt der CD *Intermediate Thinking in Java*. Beide CDs sind unter der Webadresse *http://www.mindview.net* erhältlich.

[15] Informationen über professor/support erhalten Sie bei Prentice-Hall (*www.prenhallprofessional.com*).

1.5 Die API-Dokumentation des Java Development Kits

[16] Sun Microsystems liefert die Programmiersprache Java und ihre Bibliotheken mit Dokumentation in elektronischer Form aus, die Sie mit Hilfe eines Webbrowsers lesen können (unter der Webadresse *http://java.sun.com* zum kostenlosen Herunterladen verfügbar). Viele Bücher zu Java haben diese Dokumentation kopiert, das heißt entweder haben Sie sie bereits oder Sie können sie herunterladen. Dieses Buch wiederholt diese Dokumentation nicht, wenn es nicht notwendig ist, da Sie die Beschreibung einer Klasse in der Regel schneller über den Webbrowser finden, als Sie sie in einem Buch nachschlagen können (außerdem ist die Online-Dokumentation wahrscheinlich aktueller). Das Buch verweist einfach auf die JDK- beziehungsweise API-Dokumentation und beinhaltet

nur dann eine zusätzliche Beschreibung zu einer Klasse, wenn es erforderlich ist, die Dokumentation zu ergänzen, um ein bestimmtes Beispiel verstehen zu können.

1.6 Übungsaufgaben

[17] Ich habe entdeckt, daß einfache Übungsaufgaben außergewöhnlich nützlich sind, um das Verständnis der Schulungsteilnehmer zu vervollständigen. Die Kapitel enthalten daher zahlreiche Übungsaufgaben.

[18] Die meisten Übungsaufgaben sind leicht genug um sie mit vernünftigem Zeitaufwand auch bei Schulungen unter der Aufsicht des Dozenten bearbeiten zu können, der darauf achtet, daß alle Teilnehmer mitkommen. Einige Aufgaben sind etwas schwieriger, aber keine Aufgabe ist wirklich kompliziert.

[19] Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

1.7 Erforderliche Grundlagen für Java

[20] Das Multimediaseminar *Thinking in C* ist ein zusätzlicher Bonus bei dieser Auflage und bietet eine Einführung in die Syntax, Operatoren und Funktionen von C, auf denen die Java-Syntax aufbaut. Bei den früheren Auflagen wurden diese Inhalte über die dem Buch beiliegende CD *Foundations for Java* vermittelt. Heute können Sie das Multimediaseminar kostenlos herunterladen.

[21] Ursprünglich hatte ich Chuck Allison beauftragt, *Thinking in C* als eigenständiges Produkt aufzusetzen. Die beständige Erfahrung, daß die Schulungsteilnehmer nicht über ausreichende Kenntnisse der C-Syntax verfügten, hat mich aber bewogen, das Multimediaseminar zur zweiten Auflage von *Thinking in C++* sowie zur zweiten und dritten Auflage von *Thinking in Java* hinzuzugeben. Diese Kandidaten scheinen zu denken: „Ich bin ein schlauer Programmierer und möchte C++ oder Java lernen, aber kein C. Also lasse ich C aus und fange direkt mit C++ oder Java an.“ In der Schulung angekommen, dämmert diesen Leuten langsam, daß die Voraussetzung, die Syntax von C zu verstehen, einen guten Grund hat.

[22] Die Technologien haben sich geändert und es war sinnvoller geworden, *Thinking in C* als Flash-Präsentation zum Herunterladen anzubieten, als auf CD mitzuliefern. Durch Anbieten des Multimediasemiars zum Herunterladen, kann ich sicherstellen, daß jeder mit angemessener Vorbereitung beginnt.

[23] Das Multimediaseminar *Thinking in C* macht das Buch für einen weiteren Leserkreis attraktiv. Obwohl Kapitel 3 „Alles ist Objekt“ und Kapitel 4 „Operatoren“ die von C stammenden Grundzüge von Java abdeckt, ist das Multimediaseminar eine behutsamere Einführung und setzt noch weniger Programmiererfahrung voraus, als das Buch.

1.8 Die Quelltextdistribution zu diesem Buch

[24] Der Quelltext sämtlicher Beispiele in diesem Buch („Quelltextdistribution“) ist als Freeware unter Urheberrecht unter der Webadresse *http://www.mindview.net* in Form einer einzigen ZIP-Datei erhältlich. Dies ist die offizielle Seite zum Herunterladen der Quelltextdistribution, damit

sicher sein können, die aktuelle Version zu bekommen. Der Quelltext darf bei Schulungen und zu sonstigen Lehrzwecken verteilt werden.

[25] Der Zweck des Urheberrechts besteht hauptsächlich darin, zu gewährleisten, daß der Quelltext korrekt zitiert wird sowie um zu verhindern, daß Sie den Quelltext ohne Erlaubnis in gedruckter Form veröffentlichen. (Solange die Quelle zitiert wird, ist die Verwendung von Beispielen aus dem Buch in den meisten Medien in der Regel kein Problem.)

[26–27] Jede Quelltextdatei enthält einen Verweis auf die folgende urheberrechtliche Erklärung:

```
//:! Copyright.txt
This computer source code is Copyright (c)2006 MindView, Inc.
All Rights Reserved.

Permission to use, copy, modify, and distribute this
computer source code (Source Code) and its documentation
without fee and without a written agreement for the
purposes set forth below is hereby granted, provided that
the above copyright notice, this paragraph and the
following five numbered paragraphs appear in all copies.

1. Permission is granted to compile the Source Code and to
include the compiled code, in executable format only, in
personal and commercial software programs.

2. Permission is granted to use the Source Code without
modification in classroom situations, including in
presentation materials, provided that the book ''Thinking in
Java'' is cited as the origin.

3. Permission to incorporate the Source Code into printed
media may be obtained by contacting:

MindView, Inc. 5343 Valle Vista La Mesa, California 91941
Wayne@MindView.net

4. The Source Code and documentation are copyrighted by
MindView, Inc. The Source code is provided without express
or implied warranty of any kind, including any implied
warranty of merchantability, fitness for a particular
purpose or non-infringement. MindView, Inc. does not
warrant that the operation of any program that includes the
Source Code will be uninterrupted or error-free. MindView,
Inc. makes no representation about the suitability of the
Source Code or of any software that includes the Source
Code for any purpose. The entire risk as to the quality
and performance of any program that includes the Source
Code is with the user of the Source Code. The user
understands that the Source Code was developed for research
and instructional purposes and is advised not to rely
exclusively for any reason on the Source Code or any
program that includes the Source Code. Should the Source
Code or any resulting software prove defective, the user
assumes the cost of all necessary servicing, repair, or
correction.

5. IN NO EVENT SHALL MINDVIEW, INC., OR ITS PUBLISHER BE
LIABLE TO ANY PARTY UNDER ANY LEGAL THEORY FOR DIRECT,
INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES,
INCLUDING LOST PROFITS, BUSINESS INTERRUPTION, LOSS OF
BUSINESS INFORMATION, OR ANY OTHER PECUNIARY LOSS, OR FOR
```

```
PERSONAL INJURIES, ARISING OUT OF THE USE OF THIS SOURCE
CODE AND ITS DOCUMENTATION, OR ARISING OUT OF THE INABILITY
TO USE ANY RESULTING PROGRAM, EVEN IF MINDVIEW, INC., OR
ITS PUBLISHER HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH
DAMAGE. MINDVIEW, INC. SPECIFICALLY DISCLAIMS ANY
WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR
PURPOSE. THE SOURCE CODE AND DOCUMENTATION PROVIDED
HEREUNDER IS ON AN ''AS IS'' BASIS, WITHOUT ANY ACCOMPANYING
SERVICES FROM MINDVIEW, INC., AND MINDVIEW, INC. HAS NO
OBLIGATIONS TO PROVIDE MAINTENANCE, SUPPORT, UPDATES,
ENHANCEMENTS, OR MODIFICATIONS.

Please note that MindView, Inc. maintains a Web site which
is the sole distribution point for electronic copies of the
Source Code, http://www.MindView.net (and official mirror
sites), where it is freely available under the terms stated
above.

If you think you've found an error in the Source Code,
please submit a correction using the feedback system that
you will find at http://www.MindView.net.
///:~
```

Sie können den Quelltext bei Ihren Projekten und in Schulungen verwenden (insbesondere in Ihren Schulungsunterlagen und Präsentationen), solange der in jeder Datei befindliche Hinweis auf das Urheberrecht erhalten bleibt.

1.8.1 Formatierungsrichtlinien

[28] Im Fließtext der deutschen Ausgabe dieses Buches sind Bezeichner, das heißt die Namen von Methoden, Feldern, lokalen Variablen und Klassen) in `Schreibmaschinenschrift` gesetzt. Auch die meisten Schlüsselwörter sind in `Schreibmaschinenschrift` gesetzt, ausgenommen Schlüsselwörter die so häufig verwendet werden, daß der Schreibmaschinensatz lästig wird, zum Beispiel `class`.

[29] Die Quelltextbeispiele in diesem Buch sind nach einem bestimmten Schema formatiert. Die Formatierung folgt soweit als möglich der Richtlinie, die Sun Microsystems bei nahezu allen Quelltextbeispielen vorgibt (siehe *http://java.sun.com/docs/codeconv*) und die von den meisten Java-Entwicklungsumgebungen unterstützt wird. Wenn Sie auch meine anderen Arbeiten lesen, werden Sie feststellen, daß die Formatierungsrichtlinie von Sun Microsystems mit meiner übereinstimmt. Das ist erfreulich für mich, obwohl ich (soweit ich weiß) nichts damit zu tun habe. Die Formatierung von Quelltext ist ein gutes Thema für stundenlange Diskussionen. Ich beschränke mich daher auf die Anmerkung, daß ich mit meinen Beispielen keine Stilrichtung als korrekt diktieren möchte. Ich habe meine eigenen Gründe für meine Formatierung. Da Java eine formatfreie Sprache ist, können Sie bei Ihrem gewohnten Stil bleiben. Eine Lösung des Formatierungsproblems ist, ein Hilfsprogramm wie Jalopy (*http://www.triemax.com*) zu verwenden, das mir bei der Arbeit an diesem Buch geholfen hat, um die Formatierung an Ihre Wünsche anzupassen.

[30] Alle Quelltextbeispiele im Buch wurden mit einem automatischen System getestet, sollten sich also ohne Fehlermeldungen des Compilers übersetzen lassen.

[31] Dieses Buch konzentiert sich auf und wurde getestet mit Version 5/6 der Java Standard Edition (SE 5/6). Wenn Sie etwas über eine frühere Java-Version lernen müssen, was nicht in dieser Auflage behandelt wird, finden Sie die erste bis dritte Auflage dieses Buches unter der Webadresse *http://www.mindview.net* kostenlos zum Herunterladen.

1.9 Druckfehler

[32] Gleichgültig, wie viele Hilfsmittel ein Autor verwendet, um Fehler zu finden, einige schleichen sich immer ein und springen dem „frischen“ Leser ins Auge. Wenn Sie etwas für falsch halten, folgen Sie bitte dem Verweis zu diesem Buch auf *http://www.mindview.net*[2] und schicken Sie den Fehler mit einem Korrekturvorschlag ein. Ihre Hilfe ist willkommen.

[2] Anmerkung des Übersetzers: Hinweise zu Druck- oder Übersetzungsfehlern senden Sie bitte an *ralf.wahner@web.de*.

Teil II

Einführung in die objektorientierte Programmierung

Kapitel 2

Einführung in die objektorientierte Programmierung

Inhaltsübersicht

[0] „We cut nature up, organize it into concepts, and ascribe significances as we do, largely because we are parties to an agreement that holds throughout our speech community and is codified in the patterns of our language … we cannot talk at all except by subscribing to the organization and classification of data which the agreement decrees.“ Benjamin Lee Whorf (1897-1941)

Übersetzt etwa: „Unser Verhalten, die Natur in ihre Bestandteile zu zerlegen, nach Konzepten und zugemessener Wichtigkeit zu ordnen, ist zu einem großen Teil dadurch begründet, daß wir eine Vereinbarung befolgen, die überall in unserer Sprachgemeinschaft Gültigkeit hat und in den Mustern unserer Sprache verschlüsselt ist. Wir können uns nicht mitteilen, außer wenn wir uns der Strukturierung und Klassifikation von Informationen im Sinne dieser Vereinbarung anschließen.

[1] Die Computerrevolution ging von einer Maschine aus. Die Entstehung unsere Programmiersprachen spiegelt daher solche Maschinen wider.

[2] Computer sind aber nicht in erster Linie Maschinen, sondern Hilfsmittel, um den Verstand zu erweitern („bicycles for the mind“, wie Steve Jobs gerne sagt) sowie ein ausdrucksfähiges Medium. Diese Hilfsmittel nehmen immer mehr die Gestalt von Teilen unseres ~~minds~~ statt von Maschinen an und ähneln Ausdrucksformen wie Schreiben, Malerei, Bildhauerei und Filme machen. Die objektorientierte Programmierung ist ein Teil dieser Bewegung mit dem Ziel, den Computer aus Ausdrucksmedium zu nutzen.

[3] Dieses Kapitel stellt Ihnen die grundlegenden Konzepte der objektorientierten Programmierung vor, zusammen mit einem Überblick über die Entwicklungsverfahren. Das Buch und auch dieses Kapitel setzen voraus, daß Sie Programmiererfahrung haben, nicht notwendig in C. Wenn Sie den Eindruck haben, daß Sie noch Vorbereitung brauchen, um dieses Buch zu bewältigen, sollten Sie das Multimediaseminar *Thinking in C* durchgehen.

[4] Dieses Kapitel besteht aus Hintergrund- und Zusatzinformationen. Viele fühlen sich beim Erlernen der objektorientierten Programmierung unbehaglich, wenn sie nicht zuerst einen Überblick verinnerlicht haben. Aus diesem Grund wird auf den folgenden Seiten eine Reihe von Konzepten vorgestellt, damit Sie einen soliden Überblick über die objektorientierte Programmierung bekommen. Andere dagegen, fühlen sich mit dem Überblick über die Konzepte nicht wohl, ehe sie einen Teil der Mechanik gesehen haben, bleiben ohne einige Quelltextbeispiele stecken und verpassen den Lernerfolg. Wenn Sie zur zweiten Gruppe gehören und darauf erpicht sind, sich mit den Einzelheiten der Sprache auseinanderzusetzen, überblättern Sie dieses Kapitel einfach. Das Auslassen dieses Kapitels wird Sie nicht daran hindern, Programme zu schreiben oder die Sprache zu erlernen. Vielleicht kehren Sie aber irgendwann einmal zu diesem Kapitel zurück, um Ihre Kenntnis zu vervollständigen und zu verstehen, warum Objekte wichtig sind und welche Rolle sie beim Design eines Programmes spielen.

2.1 Die Entwicklung des Abstraktionsvermögens

[5] Jede Programmiersprache repräsentiert Abstraktion. Man könnte sagen, daß die Komplexität der lösbaren Probleme direkt mit der Ausrichtung und Güte der Abstraktionsmöglichkeiten einer Programmiersprache zusammenhängt. Die „Ausrichtung“ betrifft den zu abstrahierenden Gegenstand. Die Sprache Assembler stellt zum Beispiel nur eine geringfügige Abstraktion der unterliegenden Maschine dar. Viele der folgenden, sogenannten „imperativen Sprachen“ wie Fortran, Basic und C sind wiederum Abstraktionen von Assembler. Sie sind zwar, verglichen mit Assembler, erhebliche Verbesserungen, die primäre Ausrichtung ihrer Abstraktion verlangt aber nach wie vor, daß Sie in Begriffen bezüglich der Struktur des Computers denken, statt bezüglich der Struktur des Problems, das Sie lösen wollen. Der Programmierer muß die Verbindung zwischen dem Modell der Maschine (im „Lösungsraum“ (*solution space*), das heißt dem Ort an dem die Lösung implementiert wird, zum Beispiel am Computer) und dem Modell des eigentlichen Problems herstellen (im „Problemraum“ (*problem space*), das heißt dem Ort in dem das Problem existiert, zum Beispiel ein Unternehmen). Der Aufwand, um diese Abbildung zu bewerkstelligen und die Tatsache, daß dieser Aufwand außerhalb der Möglichkeiten einer Programmiersprache liegt, ist eine der Ursachen dafür, daß die Entwicklung von Programmen schwierig und ihre Pflege teuer ist. Eine Nebenwirkung hiervon ist die Entstehung einer ganzen Industrie um „Entwicklungsverfahren“.

[6] Die Alternative zur Modellierung der Maschine ist die Modellierung des zu lösenden Problems. Frühe Sprachen wie Lisp und APL gaben bestimmte Sichtweisen vor, in diesem Fall „alle Probleme sind letztendlich Listen“ beziehungsweise „jedes Problem ist algorithmisch“. Prolog behandelt sämt-

liche Probleme in Form von Entscheidungsketten. Es gibt Sprachen zur Programmierung anhand von Beschränkungen (Constraintprogrammierung) und Sprachen in denen Sie ausschließlich durch Bedienung graphischer Symbole programmieren können. (Die letzteren Sprachen haben sich aber als zu restriktiv erwiesen.) Jeder dieser Ansätze mag für eine bestimmte Klasse von Problemen eine gute Lösung bieten, wird aber unpraktisch, sobald Sie diesen Einsatzbereich verlassen.

[7] Der objektorientierte Ansatz geht in der Hinsicht einen Schritt weiter, als er dem Programmierer Hilfsmittel zur Verfügung stellt, um Elemente im Problemraum darzustellen. Dieser Ansatz ist allgemein genug, um den Programmierer nicht auf einen bestimmten Problemtyp einzuschränken. Wir bezeichnen die Elemente des Problemraums und ihre Repräsentanten im Lösungsraum als „Objekte“. (Sie brauchen allerdings auch noch weitere Objekte, die keine Entsprechung im Problemraum haben.) Die Idee besteht darin, daß sich das Programm an die Gegebenheiten des Problems anpassen kann, indem neue Objekttypen angelegt werden können. Wenn Sie den Quelltext lesen, der die Lösung beschreibt, lesen Sie zugleich Worte, die auch das Problem beschreiben. Dies ist eine flexiblere und mächtigere Sprachabstraktion, als alles zuvor.[1] Die objektorientierte Programmierung gestattet Ihnen also, das Problem in seinen eigenen Begriffen zu beschreiben, statt in den Begrifflichkeiten des Computers, auf dem die Lösung läuft. Es gibt nach wie vor eine Verbindung zum Computer: Jedes Objekt sieht ein bißchen wie ein kleiner Computer aus, das heißt es hat einen Zustand und Operationen, die Sie ausführen können. Die Analogie zu Objekten der realen Welt ist aber nicht schlecht, denn auch diese haben Eigenschaften und Verhalten.

[8] Alan Kay beschreibt fünf grundlegende Eigenschaften von Smalltalk, der ersten erfolgreichen objektorientierten Sprache und einer der Sprachen auf denen Java aufbaut. Die Eigenschaften repräsentieren einen reinen objektorientierten Ansatz:

- *Alles ist Objekt.* Sie können sich ein Objekt als eine „schlaue Variable“ vorstellen: Es speichert Informationen, aber Sie können auch „Anfragen“ an das Objekt richten, damit es Operationen auf sich selbst ausführt. Theoretisch können Sie jede konzeptionelle Komponente des zu lösenden Objektes (Hunde, Gebäude, Dienste und so weiter) in Ihrem Programm durch ein Objekt darstellen.

- *Ein Programm besteht aus mehreren Objekten, die einander durch Versenden von Benachrichtigungen mitteilen, was zu tun ist.* Sie richten eine Anfrage an ein Objekt, indem Sie diesem Objekt „eine Benachrichtigung senden“. Sie können sich eine Benachrichtigung anschaulich als Ersuchen vorstellen, eine Methode aufzurufen, die zu einem bestimmten Objekt gehört.

- *Jedes Objekt verfügt über eigenen Arbeitsspeicher, bestehend aus anderen Objekten.* Sie legen mit anderen Worten einen neuen Objekttyp dadurch an, daß Sie verschiedene existierende Objekte zu einem Päckchen verschnüren. Sie setzen also Komplexität in Ihr Programm ein, die aber „hinter“ der Einfachheit von Objekten verborgen ist.

- *Jedes Objekt hat einen Typ.* Jedes Objekt ist nach dem Sprachgebrauch in der objektorientierten Programmierung eine *Instanz* einer Klasse, wobei „Klasse“ gleichbedeutend mit „Typ“ ist. Die wichtigste kennzeichnende Eigenschaft einer Klasse ist, welche Benachrichtigungen Sie an die Klasse senden können.

- *Alle Objekte eines bestimmten Typs können dieselben Benachrichtigungen empfangen.* Dies ist eine gewichtige Feststellung, wie Sie später sehen werden. Da ein Objekt vom Typ „Kreis“ auch ein Objekt vom Typ „geometrische Figur“ ist, akzeptiert ein „Kreis“-Objekt garantiert auch die Benachrichtigungen des Typs „geometrische Figur“. Sie können Ihren Quelltext also so

[1] Einige Sprachdesigner vertreten die Auffassung, daß die objektorientierte Programmierung von sich aus nicht geeignet ist, um alle Programmierprobleme auf einfache Weise zu lösen und befürworten stattdessen die Kombination verschiedener Ansätze zu *Mehrparadigmensprachen*. Siehe zum Beispiel Timothy Budd: *Multiparadigm Programming in Leda*, Addison Wesley (1995).

schreiben, daß sich die Anweisungen auf den Typ „geometrische Figur“ beziehen und automatisch jedes Objekt behandeln, das zur Beschreibung dieses Typs paßt. Diese *Substituierbarkeit* (*substitutability*) ist eines der mächtigen Konzepte in der objektorientierten Programmierung.

[9] Grady Booch hat eine viel knappere Beschreibung des Konzepts „Objekt“ gegeben:

Ein Objekt hat Zustand, Verhalten und eine Identität.

Ein Objekt kann intere Daten (die seinen Zustand bestimmen) und Methoden haben (die sein Verhalten ausmachen) und läßt sich in eindeutiger Weise von jedem anderen Objekt unterscheiden oder, um dieser Beschreibung ein begreifbare Form zu geben: Jedes Objekt hat eine eindeutige Adresse im Arbeitsspeicher.[2]

2.2 Die Schnittstelle eines Objektes

[10] Aristoteles war wahrscheinlich der erste Mensch, der sich einer sorgfältigen Untersuchung des Typkonzeptes gewidmet hat, da er von der „Klasse der Fische“ und der „Klasse der Vögel“ sprach. Der Ansatz, daß alle Objekte, obwohl individuell eindeutig, Teil einer Klasse von Objekten mit gemeinsamen Eigenschaften und gemeinsamem Verhalten sind, wurde in der ersten objektorientierten Programmiersprache (Simula-67) in Form des fundamentalen Schlüsselwortes `class` implementiert, welches die Definition eines neuen Typs innerhalb eines Programms einleitet.

[11] Die Sprache Simula wurde, wie ihr Name andeutet, zur Programmierung von Simulationen entwickelt, zum Beispiel des klassischen Bankschalterproblems. Dabei treten mehrere Schalter, Kunden, Konten, Transaktionen und Geldeinheiten auf, das heißt viele „Objekte“. Objekte, die bis auf ihren Zustand während der Programmausführung identisch sind, werden zu „Klassen von Objekten“ gruppiert. Hierher stammt das Schlüsselwort `class`. Das Definieren abstrakter Datentypen (Klassen) ist ein fundamentales Konzept der objektorientierten Programmierung. Abstrakte Datentypen funktionieren weitestgehend wie die eingebauten Typen: Sie können eine „Variable“ eines solchen Typs deklarieren (im Sprachgebrauch der objektorientierten Programmierung ein *Objekt* oder eine *Instanz*) und diese „Variable“ bedienen (*dem Objekt eine Benachrichtigung senden* oder *eine Anfrage an das Objekt richten*; das Objekt interpretiert diese Benachrichtigung beziehungsweise Anfrage selbst). Die Komponenten (Elemente) jeder Klasse haben gemeinsame Eigenschaften: Jedes Konto hat einen Kontostand, jeder Schalter akzeptiert Einzahlungen und so weiter. Gleichzeitig hat jede Komponente aber einen eigenen Zustand: Jedes Konto hat einen anderen Kontostand, jeder Schalter hat einen eigenen Namen. Somit läßt sich jeder Schalter, jeder Kunde, jedes Konto, jede Transaktion und so weiter als eindeutige Entität im Programm darstellen. Diese Entität ist das Objekt und jedes Objekt gehört einer bestimmten Klasse an, die seine Eigenschaften und sein Verhalten definiert.

[12] Obwohl wir in der objektorienten Programmierung eigentlich neue Daten*typen* anlegen, verwenden nahezu alle objektorientierten Programmiersprachen das Schlüsselwort `class`. Wenn Sie das Wort „Typ“ sehen, denken Sie an „Klasse“ und umgekehrt.[3]

[13] Da eine Klasse eine Menge von Objekten mit identischen Eigenschaften (Datenkomponenten) und identischem Verhalten (Funktionalität) beschreibt, ist eine Klasse tatsächlich ein Datentyp, denn beispielsweise auch eine Fließkommazahl hat Eigenschaften und Verhalten. Der Unterschied besteht darin, daß der Programmierer eine Klasse definiert, um eine Lösung für ein Problem aufzulegen, während ein existierender Datentyp entworfen wurde, um eine Einheit im Arbeitsspeicher

[2] Diese Charakterisierung ist eigentlich zu restriktiv, da Objekte auf unterschiedlichen Rechnern und in verschiedenen Adressräumen existieren und sogar auf einer Festplatte dauerhaft gespeichert werden können. In diesen Fällen muß die Identität des Objektes anhand eines anderen Kriteriums als seiner Speicheradresse festgestellt werden.

[3] Die Begriffe „Klasse“ und „Typ“ werden gelegentlich unterschieden, wobei die Schnittstelle als „Typ“ bezeichnet wird, eine bestimmte Implementierung einer Schnittstelle dagegen als „Klasse“.

eines Rechners darzustellen. Sie erweitern die Programmiersprache, indem Sie spezielle, an Ihren Bedarf angepaßte Datentypen anlegen. Die Programmiersprache akzeptiert die neuen Klassen und widmet sich ihnen mit derselben Sorgfalt und Typprüfung wie bei den eingebauten Typen.

[14] Der objektorientierte Ansatz ist keineswegs auf das Entwickeln von Simulationen beschränkt. Unabhängig davon, ob Sie der Sichtweise zustimmen, daß jedes Programm eine Simulation des eigentlichen Systems ist, ist die objektorientierte Programmierung in der Lage, eine erhebliche Menge von Problemen auf einfache Lösungen zu reduzieren.

[15] Ist eine Klasse einmal entwickelt und getestet, können Sie beliebig viele Objekte davon erzeugen und diese bedienen, als seien sie Elemente des zu lösenden Problems. In der Tat besteht eine Herausforderung in der objektorientierten Programmierung darin, eine 1:1-Abbildung zwischen den Elementen des Problemraums und den Objekten des Lösungsraums zu erwirken.

[16] Wie können Sie nun ein Objekt veranlassen, etwas nützliches zu tun? Sie brauchen eine Möglichkeit, bei einem Objekt eine Tätigkeit anzufordern, etwa eine Transaktion zu beenden, etwas auf dem Bildschirm zu zeichnen oder einen Schalter umzulegen. Ein Objekt kann nur bestimmte Anfragen verarbeiten. Die zulässigen Anfragen an ein Objekt sind in seiner *Schnittstelle* definiert und der Typ legt einen Namen für die Schnittstelle fest. Das folgende Beispiel repräsentiert eine Glühlampe:[4]


```
Light lt = new Light();
lt.on();
```

Die Schnittstelle definiert die Anfragen, die Sie an ein bestimmtes Objekt richten können. Allerdings müssen auch Anweisungen existieren, um eine solche Anfrage verarbeiten zu können. Die *Implementierung* besteht aus diesen Anweisungen, zusammen mit den verborgenen Daten. Aus der Perspektive der prozeduralen Programmierung keine komplizierte Angelegenheit. Ein Typ verknüpft jede mögliche Anfrage mit einer Methode und wenn bei einem Objekt eine bestimmte Anfrage eintrifft, wird diese Methode aufgerufen. Dieser Vorgang wird im allgemeinen durch zwei Sprechweisen zusammengefaßt: „Senden einer Benachrichtung an ein Objekt“ (Anfrage) und „das Objekt stellt selbst fest, wie diese Benachrichtigung behandelt wird“ (das Objekt verarbeitet Anweisungen).

[17] Der Name des Typs/der Klasse im Glühlampenbeispiel ist `Light`, der Name des einzelnen `Light`-Objektes ist `lt`. Die zulässigen Anfragen an ein `Light`-Objekt sind An- und Aussschalten (`on()`/`off()`) sowie Lichtstärke heller (`brighten()`) und dunkler (`dim()`) regeln. Sie erzeugen ein `Light`-Objekt, indem Sie eine „Referenzvariable“ (`lt`) dafür deklarieren, und per `new` ein neues Objekt dieses Typs anfordern. Sie senden einem Objekt eine Benachrichtigung, indem Sie den Namen der „Referenzvariablen“ des Objektes wählen, und über einen Punkt mit der Benachrichtung verknüpfen. Das ist aus der Perspektive des Clientprogrammierers, der mit einer vordefinierten Klasse arbeitet, praktisch alles im Hinblick auf die Programmierung mit Objekten.

[18] Das obige Diagramm befolgt das Format der Unified Modeling Language (UML). Eine Klasse wird als Rahmen mit drei Bereichen dargestellt, wobei der oberste Bereich den Klassennamen

[4] Anmerkung des Übersetzers: Die farbige Abbildung stammt aus der deutschen Wikipedia, Artikel „Glühlampe“.

enthält, der mittlere Bereich die *Felder* der Klasse und der untere Bereich die *Methoden* (die Funktionen des Objektes, welche die an das Objekt gesendeten Benachrichtigungen enthalten). Häufig enthält das Diagramm nur den Klassennamen und die öffentlichen Methoden, nicht aber den mittleren Bereich, wie im obigen Beispiel. Wenn Sie sich nur für den Klassennamen interessieren, kann auch der Methodenbereich fortgelassen werden.

2.3 Objekte sind Anbieter von Diensten

[19] Wenn Sie das Design eines Programmes entwickeln oder zu verstehen versuchen, stellen Sie sich Objekte am besten als „Anbieter von Diensten“ vor. Ein Programm bietet dem Benutzer Dienste an und bildet seine Funktionalität mit Hilfe der von anderen Objekten angebotenen Dienste. Ihr Ziel besteht darin, Klassen zu entwickeln, beziehungsweise in existierenden Bibliotheken zu finden, welche die idealen Dienste zur Lösung Ihres Problems anbieten.

[20] Fangen Sie mit der Frage an, welche Klassen Ihr Problem auf der Stelle lösen würden, wenn Sie sie „aus dem Hut zaubern“ könnten. Stellen Sie sich zum Beispiel ein Buchführungsprogramm vor. Einige Klassen stellen vordefinierte Buchungsposten dar, andere führen Berechnungen aus und wieder eine andere Klasse druckt Bankanweisungen und Rechnungen auf einem beliebigen Druckermodell aus. Ein Teil dieser Klassen ist eventuell schon vorhanden. Welche Eigenschaften und Fähigkeiten müssen die noch nicht existierenden Klassen haben? Welche Dienste müssen die Objekte der noch nicht vorhandenen Klassen zur Verfügung stellen und welche Klassen sind wiederum notwendig, damit die ersteren ihre Aufgabe erfüllen können? Wenn Sie Ihr Problem auf diese Art und Weise untersuchen, erreichen Sie letztendlich dem Punkt, an dem sich zeigt, daß die benötigte Klasse entweder einfach genug ist, um selbst programmiert zu werden oder voraussichtlich bereits eine entsprechende Klasse existiert. Dies ist eine vernünftige Vorgehensweise, um ein Problem in mehrere Klassen zu zerlegen.

[21] Ein Objekt als Anbieter von Diensten zu betrachten, hat noch einen weiteren Vorteil: Die Kohäsion der Klasse des Objektes wird verbessert. Starke Kohäsion ist eine fundamentale Qualität des Designs einer Software und bedeutet, daß die verschiedenen Aspekte einer Softwarekomponente (beispielsweise einer Klasse, einer Methode oder einer Bibliothek von Klassen) „zusammenpassen“. Programmierer neigen beim Design einer Klasse dazu, sie mit zuviel Funktionalität zu überfrachten. Sie könnten sich beispielsweise entscheiden, daß das Modul zum Drucken von Bankanweisungen das Formatieren und Ausdrucken komplett implementiert. Ihnen leuchtet sicher ein, daß dieser Funktionsumfang für eine Klasse zuviel ist und besser auf mehrere Klassen aufgeteilt wird. Eine Klasse könnte einen Katalog aller benötigten Vorlagen für Bankanweisungen darstellen, deren Objekt(e) nach Informationen darüber abgefragt werden können, wie ein Bankanweisung ausgedruckt wird. Eine zweite Klasse könnte eine generische Druckerschnittstelle repräsentieren, die sämtliche Daten über alle lieferbaren Drucker beinhaltet (aber keine Informationen über Buchführung; eine solche Klassen sollte besser käuflich erworben statt selbst entwickelt werden). Eine dritte Klasse könnte die Dienste der beiden ersteren Klassen kombinieren, um das eigentliche Problem zu lösen. Jede Klasse bietet somit eine kohäsive Menge von Diensten an. In einem guten objektorientierten Design verrichtet jede Klasse eine bestimmte Aufgabe und nichts sonst. Auf diese Weise zeigen sich nicht nur Klassen, die eventuell von einem Drittanbieter gekauft werden können (die Druckerschnittstelle), sondern auch Klassen, die eventuell an einer anderen Stelle wiederverwendet werden können (der Vorlagenkatalog).

[22] Die Behandlung von Klasse beziehungsweise Objekten als Anbieter von Diensten ist ein sehr wirksames vereinfachendes Hilfsmittel und zahlt sich nicht nur im Designprozeß aus, sondern auch wenn sich jemand anders mit Ihrem Quelltext auseindersetzt, um ihn zu verstehen oder wiederzuverwenden.

2.4 Die verborgene Implementierung

[23] Es ist sinnvoll, die Programmierer in zwei Gruppen einzuteilen, nämlich die *Autoren* von Klassen, die neue Datentypen entwickeln und die *Clientprogrammierer*,[5] das heißt die Konsumenten von Klassen, welche die Datentypen in Ihren Anwendungen nutzen. Die Clientprogrammierer haben das Ziel, einen Werkzeugenkasten voller Klassen anzusammeln, die sie beim Entwickeln von Anwendungen einsetzen können. Das Ziel der Autoren besteht darin, eine Klasse zu schreiben, welche nur die vom Clientprogrammierer benötigten Komponenten exponiert und alles übrige verbirgt. Warum? Weil die Clientprogrammierer keinen Zugriff auf den verborgenen Teil haben, so daß der Autor Änderungen am verborgenen Teil vornehmen kann, ohne sich um Auswirkungen für die Clientprogrammierer kümmern zu müssen. Der verborgene Teil der Implementierung besteht in der Regel aus dem empfindlichen Innenleben der Klasse beziehungsweise des Objektes und kann von einem unvorsichtigen oder schlecht informierten Clientprogrammierer leicht unbrauchbar gemacht werden. Das Verbergen der Implementierung reduziert die Programmierfehler.

[24] In jeder Beziehung sind Grenzen wichtig, die von allen Beteiligten respektiert werden. Beim Anlegen einer Bibliothek entsteht eine Beziehung zu den Clientprogrammierern, die die Bibliothek nutzen. Die Anwender einer Bibliothek sind selbst Programmierer und verwenden die Bibliothek, um eine Anwendung oder eine noch umfangreicherere Bibliothek zu entwickeln. Sind alle Komponenten einer Klasse für jedermann zugänglich, so können die Clientprogrammierer mit der Klasse tun was sie wollen und es gibt keine Möglichkeit, die Einhaltung von Regeln zu erzwingen. Sie können zwar von den Clientprogrammierern verlangen, daß sie einige Komponenten Ihrer Klasse meiden, haben aber ohne Zugriffskontrolle keine Möglichkeit, um unerwünschte Zugriffe zu verhindern. Die Komponenten Ihrer Klasse sind ohne Zugriffskontrolle völlig ungeschützt.

[25] Das erste Argument für Zugriffskontrolle lautet also, die Clientprogrammierer am Zugriff auf Komponenten zu hindern, die sich nicht berühren sollen. Diese sind Bestandteile, die für die internen Operationen der Klasse benötigt werden, aber nicht zur Schnittstelle der Klasse gehören, welche die Clientprogrammierer bedienen. Die Zugriffskontrolle ist eigentlich eine Hilfestellung für die Clientprogrammierer, da sie mühelos erkennen können, welche Komponenten wichtig sind und welche sie ignorieren können.

[26] Das zweite Argument für Zugriffskontrolle lautet, daß der Autor der Bibliothek Änderungen an der internen Funktionsweise einer Klasse vornehmen kann, ohne sich über Auswirkungen für die Clientprogrammierer sorgen zu müssen. Beispielsweise könnten Sie eine bestimmte Klasse zunächst in einer vereinfachten Form schreiben, nach einiger Zeit aber feststellen, daß eine Änderung das Verhalten der Objekte beschleunigen würde. Sind Schnittstelle und Implementierung klar voneinander getrennt und geschützt, so können Sie eine solche Änderung leicht ausführen.

[27] Java hat drei Schlüsselworte, um Zugriffsbeschränkungen in einer Klasse zu deklarieren: `public`, `private` und `protected`. Diese *Zugriffsmodifikatoren* deklarieren, wer die anschließende Deklaration oder Definition verwenden darf. Eine mit dem `public`-Modifikator deklarierte Komponente steht jedermann zur Verfügung. Das Schlüsselwort `private` bedeutet dagegen, daß niemand außer Ihnen, dem Autor der Klasse, Zugriff auf diese Komponente erhält und zwar nur im Inneren von Methoden derselben Klasse. Der `private`-Modifikator ist eine gemauerte Wand zwischen dem Autor und den Clientprogrammierern. Ein Zugriffsversuch auf eine private Komponente wird zur Übersetzungszeit als Fehler gemeldet. Der `protected`-Modifikator wirkt wie `private`, aber mit der Ausnahme, daß eine abgeleitete Klasse Zugriff auf die als `protected` deklarierten Felder ihrer Basisklasse erhält, nicht aber auf die privaten Felder. Das Konzept der Ableitung wird in Abschnitt 2.6 vorgestellt.

[28] Java definiert außerdem einen „Standardzugriff“, der stets wirksam ist, wenn Sie keinen der

[5] Ich bin meinem Freund Scott Meyers für diese Bezeichnung zu Dank verpflichtet.

drei zuvor beschriebenen Zugriffsmodifikatoren einsetzen. Der Standardzugriff wird üblicherweise als *Packagezugriff* bezeichnet, da die entsprechenden Komponenten für Klassen im selben Package erreichbar sind, nicht aber für Klassen außerhalb des Packages. Komponenten unter Packagezugriff verhalten sich, Klassen außerhalb des Packages gegenüber, wie private Komponenten.

2.5 Wiederverwendung einer Implementierug

[29] Eine fertig entwickelte und getestete Klasse ist (im Idealfall) ein wiederverwendbares Stück Quelltext. Allerdings hat sich gezeigt, daß Wiederverwendbarkeit nicht annähernd so leicht zu erreichen ist, als allgemein angenommen wird. Vielmehr ist Erfahrung und Einsicht erforderlich, um eine wiederverwendbare Klasse zu entwickeln. Ist es Ihnen aber gelungen, eine solche Klasse zu schreiben, so „schreit“ sie geradezu danach, wiederverwendet zu werden. Die Wiederverwendung von Quelltext ist einer der größten Vorteile der objektorientierten Programmierung.

[30] Die einfachste Art, eine Klasse wiederzuverwenden besteht darin, ein Objekt der Klasse direkt zu verwenden, aber Sie können auch ein Objekt dieser Klasse in eine andere Klasse einsetzen, also ein sogenanntes *Komponentenobjekt* anlegen. Ihre neue Klasse kann beliebig viele Objekte beliebiger Klasse in beliebiger Kombination enthalten, um die gewünschte Funktionalität zu implementieren. Da Sie die neue Klasse aus Objekten existierender Klassen zusammensetzen („komponieren“), wird dieses Konzept als **Komposition** bezeichnet (bei dynamischer Komposition spricht man von **Aggregation**). Die Komposition wird häufig auch als „hat ein“-Beziehung bezeichnet, zum Beispiel „ein Auto hat einen Motor“:

(Die ausgefüllte Raute in diesem UML-Diagramm zeigt eine Komposition an, genauer „ein Auto hat einen Motor“. Ich verwende in der Regel eine einfachere Notation, bestehend aus einer Linie ohne Raute, um eine Assoziation anzuzeigen.[6])

[31] Der Kompositionsansatz erlaubt viel Flexibilität. Die Komponentenobjekte Ihrer neuen Klasse sind typischerweise privat und somit für die Clientprogrammierer, die Ihre Klasse verwenden nicht erreichbar. Dadurch können Sie die Komponentenobjekte modifizieren, ohne vorhandene Quelltexte von Clientprogrammierern zu stören. Sie können die Komponentenobjekte sogar zur Laufzeit austauschen, das Verhalten Ihres Programms also dynamisch ändern. Der im nächsten Abschnitt beschriebene Ableitungsansatz hat diese Flexibilität nicht, da der Compiler per Ableitung definierten Klassen bereits zur Übersetzungszeit Beschränkungen auferlegen muß.

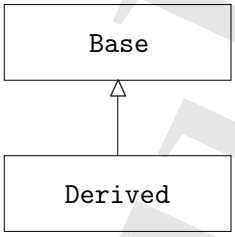
[32] Der Ableitungsmechanismus ist ein sehr wichtiges Konzept der objektorientierten Programmierung und wird entsprechend häufig hervorgehoben. Unerfahreren Programmierern kann dadurch vermittelt werden, daß die Ableitung so oft als möglich gebraucht werden sollte. Ungeschicktes und übermäßig kompliziertes Design ist die Folge davon. Sie sollten stattdessen bei neuen Klassen zunächst die einfachere und flexiblere Komposition wählen. Ihr Design wird dadurch sauberer. Wenn Sie etwas mehr Erfahrung gesammelt haben, zeigt sich einigermaßen offensichtlich, wo Ableitung benötigt wird.

[6] Dies ist in der Regel für die meisten Diagramme detailliert genug und Sie brauchen nicht präzise anzugeben, ob Sie eine Aggretation oder eine Komposition verwenden.

2.6 Ableitung (Vererbung)

[33] Das Konzept der Klasse ist ein komfortables Werkzeug. Es gestattet Ihnen, Daten und Funktionalität konzeptionell zusammenzufassen, so daß Sie ein passendes Konzept im Problemraum darstellen können und nicht gezwungen sind, die Ausdrucksweise des unterliegenden Rechners zu verwenden. Diese Konzepte werden in der Programmiersprache mit Hilfe des Schlüsselwortes `class` als fundamentale Einheiten ausgewiesen.

[34] Angesichts des Aufwandes beim Entwickeln einer Klasse ist es bedauerlich, bei einer anderen, ebenfalls neuen Klasse mit ähnlicher Funktionalität wieder von vorne anfangen zu müssen. Ist es nicht sinnvoller, ein „Duplikat“ der existierenden Klasse durch Ergänzungen und Änderungen anzupassen? Genau diesen Effekt leistet die Ableitung (Vererbung), wobei sich Ergänzungen und Änderung an der ursprünglichen Klasse (*Basisklasse*, *Oberklasse*, *übergeordnete Klasse* oder auch *Elternklasse*) allerdings auch immer in der „duplizierten“ Klasse (*abgeleitete Klasse*, *Unterklasse*, *untergeordnete Klasse* oder auch *Kindklasse*) niederschlagen:



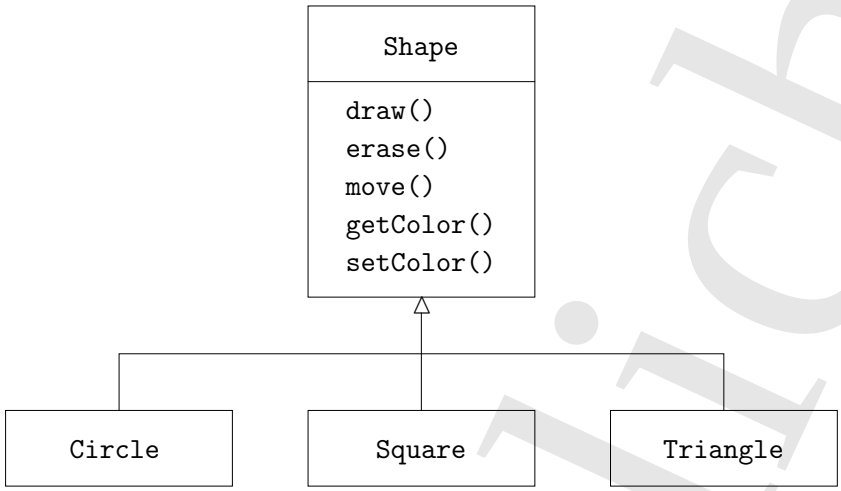
(In diesem UML-Diagramm zeigt der Pfeil von der abgeleiteten Klasse zur Basisklasse. Wie Sie sehen werden, gibt es häufig mehr als eine abgeleitete Klasse.)

[35] Die Definition eines Typs beinhaltet mehr, als die bloße Beschreibung der Grenzen einer Anzahl von Objekten, nämlich auch die Beziehungen zu anderen Typen. Zwei Typen können Eigenschaften und Verhalten gemeinsam haben, wobei der eine Typ aber auch über mehr Eigenschaften verfügen, zusätzliche Benachrichtigungen verarbeiten oder dieselben Benachrichtungen anders verarbeiten kann, als der andere. Der Ableitungsmechanismus drückt diese Ähnlichkeit zwischen Typen mit Hilfe des Konzeptes des Basistyps und des davon abgeleiteten Typs aus. Sie definieren einen Basistyp, um die Kerneigenschaften und -fähigkeiten bestimmter Objekte Ihres Programms darzustellen. Aus dem Basistyp leiten Sie andere Typen ab, um die verschiedenen Ausdrucksmöglichkeiten der Kerneigenschaften und -fähigkeiten zu realisieren.

[37] Eine Wiederverwertungsanlage zum Beispiel, sortiert Abfall. Der Basistyp könnte `Trash` heißen, wobei jedes Stück Abfall ein Gewicht, einen Wert und so weiter hat und geschreddert, geschmolzen oder zerlegt werden kann. Aus diesem Basistyp könnten nun weitere Abfallsorten abgeleitet werden, die eventuell zusätzliche Eigenschaften (etwa die Farbe von Altglas) oder Verhalten haben können (Aluminium kann zerkleinert werden, Stahl ist magnetisch). Ein Teil des Verhaltens könnte von der Abfallsorte abhängen, etwa der Wert von Altpapier von Papiersorte und Zustand. Das Ableitungskonzept gestattet Ihnen, eine Hierarchie von Typen anzulegen, die das zu lösende Problem mit Hilfe dieser Typen ausdrückt.

[38] Ein anderes Beispiel ist die klassische Hierarchie geometrischer Figuren, die etwa bei CAD-Systemen (Computer Aided Design) oder in der Spieleprogrammierung verwendet wird. Der Basistyp der Hierachie ist `Shape` und schreibt jeder geometrischen Figur eine Größe, eine Farbe, eine Position und so weiter zu. Jede geometrische Figur kann gezeichnet, gelöscht, bewegt, gefärbt werden und so weiter. Vom Basistyp `Shape` werden nun spefizische Typen von Figuren abgeleitet, zum Beispiel `Circle`, `Square`, `Triangle` und so weiter, die jeweils zusätzliche Eigenschaften und Verhaltensweisen haben können. Einige Figuren können beispielsweise an einer Achse gespiegelt werden.

Ein Teil des Verhaltens ist spezifisch für den Typ der Figur, zum Beispiel die Berechnung der Fläche. Die Typhierarchie verkörpert sowohl Gemeinsamkeiten als auch Abweichungen zwischen den verschiedenen geometrischen Figuren:

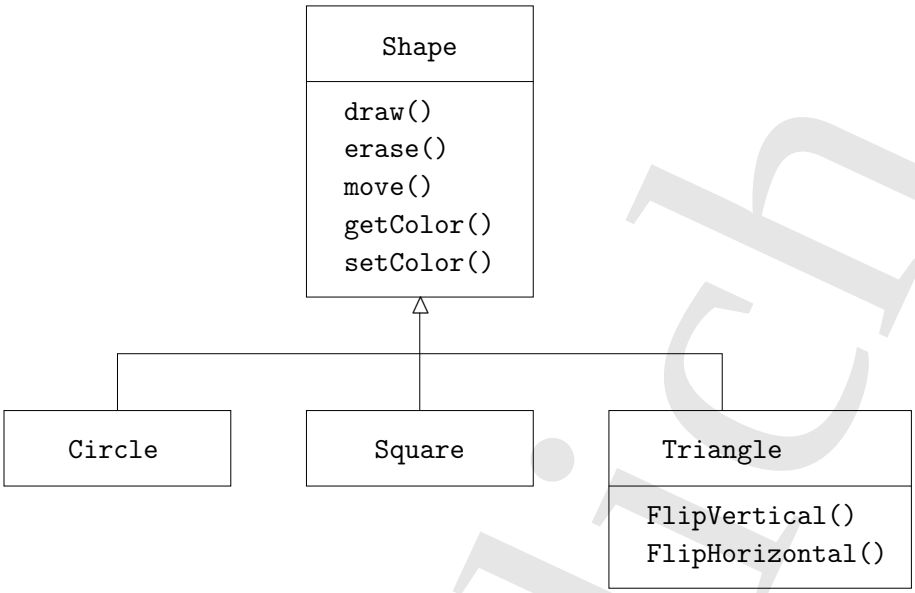


[39] Die Beschreibung der Lösung mit den Begriffen des Problems ist sehr nützlich, da kein intermediäres Modell benötigt wird, um die Beschreibung des Problems auf die Beschreibung der Lösung abzubilden. In der objektorientierten Programmierung ist die Typhierarchie das primäre Modell, so daß Sie von der Beschreibung des Problems in der Realität direkt zur Beschreibung des Systems in Form des Quelltextes übergehen. Tatsächlich besteht eine der Schwierigkeiten beim objektorientierten Design darin, daß man zu leicht vom Anfang zum Ende kommt. Ein für die Suche nach komplexen Lösungen konditionierter Verstand ist angesichts dieser Einfachheit unter Umständen zunächst verblüfft.

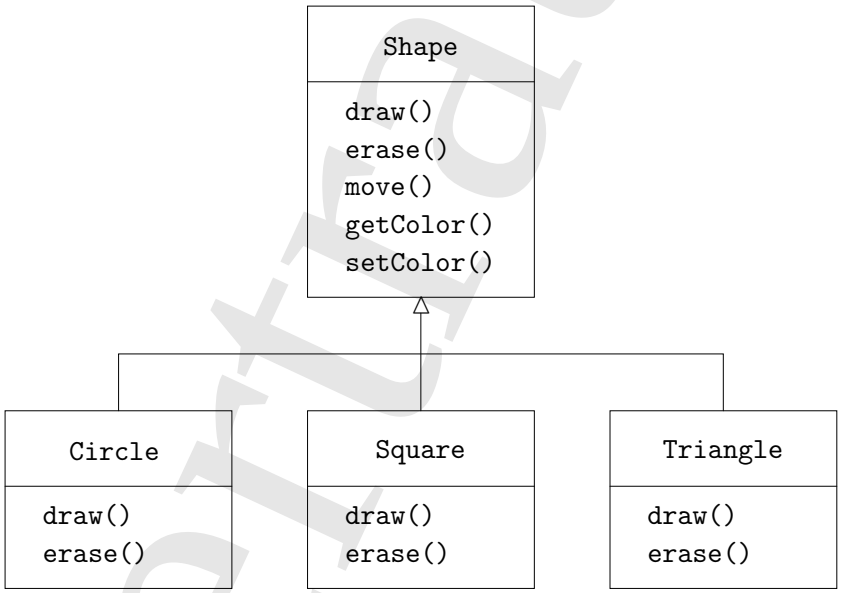
[40] Wenn Sie einen existierenden Typ ableiten, definieren Sie einen neuen Typ. Dieser neue Typ beinhaltet nicht nur sämtliche Komponenten des existierenden Typs, wobei die privaten Komponenten allerdings verborgen sind und nicht erreicht werden können, sondern (wichtiger noch) dupliziert die Schnittstelle der Basisklasse. Somit können Sie alle Benachrichtigungen an ein Objekt der Basisklasse auch an jedes Objekt einer von dieser abgeleiteten Klasse senden. Da sich der Typ einer Klasse aus den Benachrichtigungen ergibt, die an ein Objekt dieser Klasse gesendet werden können, ist die abgeleitete Klasse in diesem Sinne vom selben Typ als die Basisklasse. Im vorigen Beispiel ist die Klasse `Circle` vom Typ `Shape`. Diese ableitungsbedingte Typäquivalenz ist eine der fundamentalen Schnittstellen zum Verständnis der Bedeutung der objektorientierten Programmierung.

[41] Da Basisklasse und abgeleitete Klasse dieselbe fundamentale Schnittstelle haben, muß jeweils eine *individuelle* Implementierung, das heißt typspezifische Anweisungen existieren, die beim Empfangen einer Benachrichtigung verarbeitet werden. Wenn Sie eine Klasse nur ableiten und sonst nichts tun, gehen die Methoden der Basisklasse an die abgeleitete Klasse über. Ein Objekt der abgeleiteten Klasse hat in diesem Fall nicht nur dieselbe Schnittstelle wie ein Objekt der Basisklasse, sondern auch identisches Verhalten. Dieser Fall ist nicht besonders interessant.

[42] Es gibt zwei Möglichkeiten, um die neue abgeleitete Klasse von der Basisklasse unterscheidbar zu machen. Sie können einerseits neue Methoden in der abgeleiteten Klassen definieren, die allerdings nicht zur Schnittstelle der Basisklasse gehören. In diesem Fall stellt die Basisklasse nicht das gesamte gewünschte Verhalten zur Verfügung, so daß Sie zusätzliche Methoden anlegt haben. Diese einfache und primitive Anwendung des Ableitungsmechanismus' ist gelegentlich die perfekte Lösung für ein Problem. Denken Sie in diesem Fall aber sorgfältig darüber nach, ob die Basisklasse nicht ebenfalls die zusätzlichen Methoden benötigt. Dieser Erkenntnis- und Iterationsprozeß hinsichtlich des Designs geschieht in der objektorientierten Programmierung regelmäßig.



[43] Die Ableitung einer Basisklasse *kann*, muß aber nicht notwendig das Erweitern der Schnittstelle der abgeleiteten Klasse durch zusätzliche Methoden beinhalten (gerade bei Java, wo die Ableitung über das Schlüsselwort `extends` definiert wird). Die andere Möglichkeit, um die abgeleitete Klasse von der Basisklasse unterscheidbar zu machen besteht darin, daß Verhalten einer in der Basisklasse definierten Methode durch sogenanntes *Überschreiben* zu ändern.



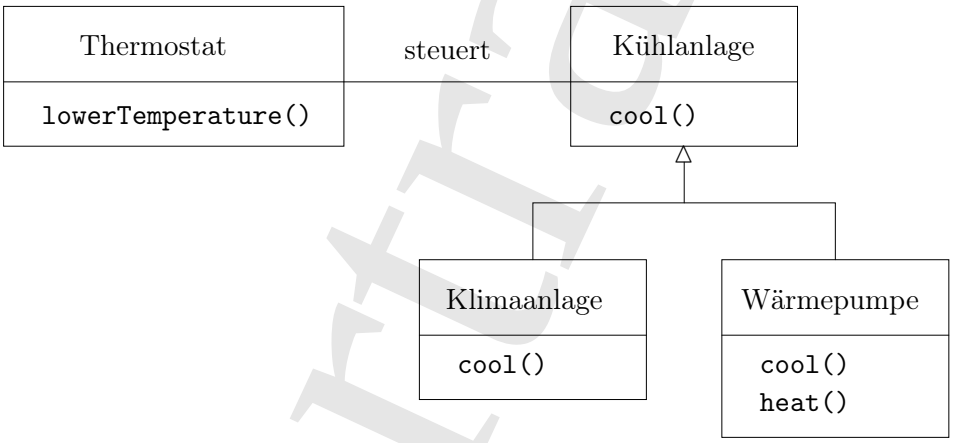
Sie überschreiben eine Methode, indem Sie einfach eine neue Definition in der abgeleiteten Klasse anlegen. Das Überschreiben einer Methode bedeutet, daß Sie eine Methode aus der Schnittstelle der Basisklasse mit einem typspezifischen Verhalten ausstatten.

2.6.1 „Ist ein“-Beziehung und „ähnelt einem“-Beziehung

[44] Es besteht eine gewisse Auseinandersetzung hinsichtlich der Frage, ob unter der Ableitung nur Methoden der Basisklasse überschrieben und keine zusätzlichen Methoden definiert werden sollten. Dies würde bedeuten, daß der Typ der abgeleiteten Klasse *exakt* dem Typ der Basisklasse entspricht,

da beide exakt dieselbe Schnittstelle haben. Demzufolge können Sie ein Objekt der Basisklasse exakt durch ein Objekt der abgeleiteten Klasse ersetzen. Sie können sich eine *reine Substitution* vorstellen und die Ersetzbarkeit wird häufig als *Substitutionsprinzip* bezeichnet. Dies ist eigentlich der ideale Umgang mit dem Ableitungsmechanismus. Das Verhältnis zwischen Basisklasse und abgeleiteter Klasse wird häufig als „ist ein"-Beziehung bezeichnet, denn „ein Kreis (`Circle`) ist eine geometrische Figur (`Shape`)". Die Frage, ob eine „ist ein"-Beziehung möglich und sinnvoll ist, liefert eine Entscheidungshilfe dafür, ob die Ableitung von einer existierenden Klasse angebracht ist.

[45] Manchmal müssen Sie der Schnittstelle einer abgeleiteten Klasse neue Elemente hinzufügen, die Schnittstelle also erweitern. Der abgeleitete Typ kann nach wie vor anstelle des Basistyps eingesetzt werden, aber die Substitution ist nicht mehr perfekt, da die zusätzlichen Methoden über die Schnittstelle des Basistyps nicht mehr erreichbar sind. Ich verwende in diesem Fall die Bezeichnung *„ähnelt einem"-Beziehung*. Der neue Typ hat zwar die Schnittstelle des alten Typs, definiert aber auch weitere Methoden, so daß die Schnittstellen nicht wirklich als exakt identisch bezeichnet werden können. Betrachten Sie eine Klimaanlage als Beispiel. Angenommen, Ihr Haus ist so verdrahtet, daß Sie eine Kühlanlage steuern können, das heißt Sie haben eine Schnittstelle, um eine Kühlanlage steuern zu können. Stellen Sie sich vor, daß die Klimaanlage kaputt geht und Sie stattdessen eine Wärmepumpe einsetzen, die sowohl die Funktion einer Heizung als auch die einer Kühlanlage übernehmen kann. Die Wärmepumpe ähnelt einer Klimaanlage, hat aber zusätzliche Funktionen. Da das Steuersystem in Ihrem Haus nur eine Kühlanlage bedienen kann, ist die Kommunikation zwischen Steuersystem und der Wärmepumpe auf den kühlenden Anteil der neuen Maschine beschränkt. Die Schnittstelle der neuen Maschine wurde erweitert, aber das vorhandene Steuersystem kennt nur die ursprüngliche Schnittstelle:



[46] Ein Blick auf das Design zeigt, daß die Basisklasse zur Steuerung der Kühlanlage nicht allgemein genug ist. Der Name der Basisklasse sollte wiedergeben, daß sie die Temperatur regelt, also auch die Heizung bedienen kann und das Substitutionsprinzip wäre wieder erfüllt. Das Diagramm ist ein Beispiel dafür, was einem Design in der Wirklichkeit widerfahren kann.

[47] Angesichts des Substitutionsprinzips stellt sich leicht der Eindruck ein, daß dieser Ansatz (reine Substitution) ein einzig richtige Weg ist. Es ist selbstverständlich schön, wenn sich Ihr Design in dieser Weise ergibt. Es gibt allerdings Situationen, in denen sich genauso klar herausstellt, daß Sie die Schnittstelle der abgeleiteten Klasse um zusätzliche Methoden erweitern müssen. Bei sorgfältiger Betrachtung sollten sich beide Fälle einigermaßen offensichtlich zeigen.

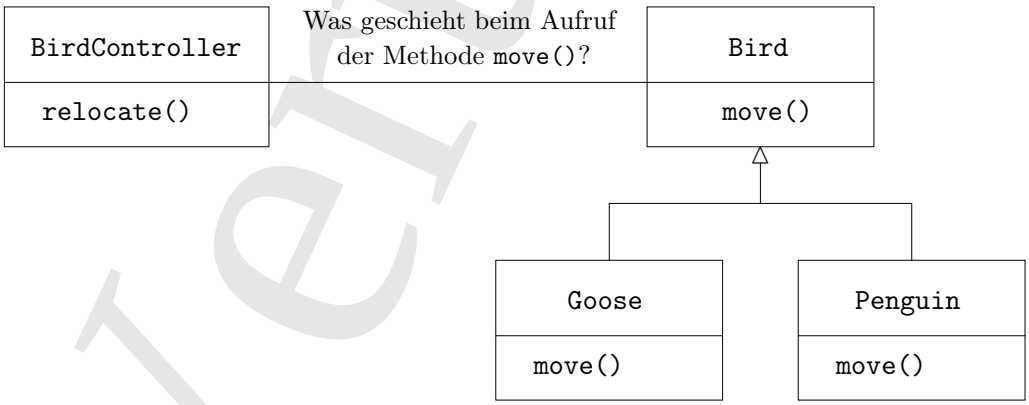
2.7 Austauschbarkeit von Objekten durch Polymorphie

[48] Bei der Arbeit mit Typhierarchien kommt es häufig vor, daß Sie ein Objekt nicht bezüglich seines spezifischen Typs, sondern bezüglich seines Basistyps behandeln möchten. Auf diese Weise können Sie Anweisungen anlegen, die nicht von einem spezifischen abgeleiteten Typ abhängen. Ein Methode, die Typen aus der Hierarchie geometrischer Figuren im vorigen Abschnitt aufruft, würde beispielsweise generische Objekte vom Typ `Shape` verarbeiten und nicht beachten, ob es sich dabei um `Circle`, `Square`, `Triangle`, oder einen Figurentyp handelt, der eventuell noch nicht definiert ist. Jede Figur kann gezeichnet, gelöscht und bewegt werden, so daß die Methode einfach eine Benachrichtigung an die Schnittstelle der Klasse `Shape` sendet.

[49] Eine solche Methode bleibt unberührt, wenn neue Typen von geometrischen Figuren definiert werden. Das Hinzufügen neuer Typen ist die häufigste Art und Weise, um ein objektorientiertes Programm an die Verarbeitung neuer Situation anzupassen. Sie können zum Beispiel einen neuen Typ `Pentagon` von `Shape` ableiten, ohne eine Methode ändern zu müssen, die ein Argument vom Typ `Shape` erwartet. Diese Befähigung, ein Design mühelos durch Ableitung eines neuen Typs zu erweitern, ist eine der grundsätzlichen Möglichkeiten zur Kapselung von Änderungen. Designs werden auf diese Weise erheblich verbessert, während zugleich die Kosten für die Pflege der Software reduziert werden.

[50] Der Versuch, ein Objekt einer abgeleiteten Klasse wie ein Objekt der generischen Basisklasse zu behandeln (Kreise wie Figuren, Zweiräder wie Fahrzeuge, Kormorane wie Vögel und so weiter), führt zu einem Problem: Veranlaßt eine Methode eine generische geometrische Figur sich zu zeichnen, eine generisches Fahrzeug sich zu bewegen oder einen generischen Vogel zu fliegen, so kann der Compiler zur Übersetzungszeit nicht „wissen“, welches Stück Quelltext ausgeführt wird. Genau dieser Aspekt ist der Kerngedanke des Polymorphiekonzeptes: Der Programmierer will überhaupt nicht wissen, welches Stück Quelltext verarbeitet wird. Die `draw()`-Methode kann gleichmaßen auf einem `Circle`-, `Square`- oder `Triangle`-Objekt aufgerufen werden und das Objekt verarbeitet die seinem spefizischen Typ entsprechenden Anweisungen.

[51] Wenn Sie sich nicht darum kümmern müssen, welches Stück Quelltext verarbeitet wird, können diese Anweisungen bei einem neu hinzugefügten Untertyp individuell definiert werden, ohne die aufrufende Methode verändern zu müssen. Der Compiler kann folglich nicht im Einzelnen „wissen“, welches Stück Quelltext ausgeführt wird. Wie verhält sich der Compiler in dieser Situation?



Die Klasse `BirdController` in diesem Diagramm arbeitet mit Objekten des generischen Typs `Bird` und „weiß“ nicht, welcher eigentliche Typ vorliegt. Dies ist für die Klasse `BirdController` eine komfortable Lösung, da keine weiteren Anweisungen benötigt werden, um den exakten Untertyp von `Bird` zu bestimmen oder dessen Verhalten anzusprechen. Wie geht es also vor sich, daß sich beim Aufrufen der `move()`-Methode und gleichzeitigem Nichtbeachten des eigentlichen Untertyps

von `Bird`, dennoch das richtige Verhalten zeigt (eine Gans watschelt, fliegt oder schwimmt, ein Pinguin watschelt oder schwimmt)?

[52] Die Antwort führt zu einer fundamentalen überraschenden Wendung in der objektorientierten Programmierung: Der Compiler kann keinen Methodenaufruf im traditionellen Sinne veranlassen. Ein von einem Compiler in einer nicht objektorientierten Sprache veranlaßter Funktionsaufruf basiert auf *statischer Bindung* (früher Bindung). Dieser Begriff ist Ihnen unter Umständen noch nie begegnet, da Sie niemals einen anderen Mechanismus in Betracht zu ziehen brauchten. Bei der statischen Bindung verknüpft der Compiler einen Funktionsnamen mit einem Funktionsaufruf, der zur Laufzeit in die absolute Speicheradresse des auszuführenden Codes aufgelöst wird. In der objektorientierten Programmierung kann das Programm die Speicheradresse nicht vor der Laufzeit bestimmen, so daß ein anderes Schema benötigt wird, um einem generischen Objekt eine Benachrichtigung zu senden.

[53] Die objektorientierten Sprachen lösen das Problem mit Hilfe der *dynamischen Bindung* (späten Bindung). Wenn Sie einem Objekt eine Benachrichtigung senden, wird der auszuführende Code erst zur Laufzeit bestimmt. Der Compiler gewährleistet, daß die aufgerufene Methode definiert ist und unterzieht Argumente und Rückgabewert einer Typprüfung, hat aber keine Informationen über den exakten auszuführenden Code.

[54] Java verwendet bei der dynamischen Bindung ein spezielles Codestück anstelle des absoluten Methodenaufrufs. Dieser Code berechnet die Speicheradresse des Methodenkörpers anhand von Informationen, die im Objekt gespeichert sind (dieser Prozeß wird in Kapitel 9 detailliert beschrieben). Somit kann sich jedes Objekt individuell verhalten, gemäß dem Inhalt dieses speziellen Codestücks. Wenn Sie einem Objekt eine Benachrichtigung senden, kümmert sich das Objekt also eigentlich selbsttätig darum, die Benachrichtung auszuwerten.

[55] Bei manchen Sprachen müssen Sie explizit deklarieren, daß Sie bei einer Methode die Flexibilität der dynamischen Bindung nutzen wollen (C++ verwendet hierfür das Schlüssel `virtual`). Bei diesen Sprachen werden Methodenaufrufe im Standardverhalten nicht dynamisch gebunden. Bei Java ist die dynamische Bindung dagegen Standardverhalten und Sie brauchen nicht an zusätzliche Schlüsselworte zu denken, um polymorphes Verhalten zu nutzen.

[56] Wir kehren nochmals zum Figurenbeispiel zurück. Das Diagramm der Klassenhierarchie (alle Klassen basieren auf derselben einheitlichen Schnittstelle) wurde im vorigen Abschnitt angegeben. Zur Demonstration des Polymorphieeffektes schreiben wir ein einzelnes Stück Quelltext, das die spezifischen Einzelheiten des Typs nicht beachtet und nur mit der Schnittstelle der Basisklasse kommuniziert. Diese Anweisungen sind von typspezifischen Informationen entkoppelt und daher leichter zu entwicklen und zu verstehen. Wird die Klassenhierarchie um einen neuen Typ erweitert, zum Beispiel `Hexagon`, so funktioniert das einzelne Stück Quelltext ebensogut mit dem neuen Typ, als mit den zuvor vorhandenen Typen. Das Programm ist also *erweiterbar*.

[57] Die Methode (Sie lernen bald, eine Methode zu definieren)

```
void doSomething(Shape shape) {
    shape.erase();
    // ...
    shape.draw();
}
```

kommuniziert mit einem Argument vom Typ `Shape`, ist also unabhängig vom spezifischen Typ des gezeichneten und später wieder gelöschten Objektes. Wird die Methode `doSomething()` in einem anderen Teil des Programms aufgerufen:

```
Circle circle = new Circle();
Triangle triangle = new Triangle();
```

```
Line line = new Line();
doSomething(circle);
doSomething(triangle);
doSomething(line);
```

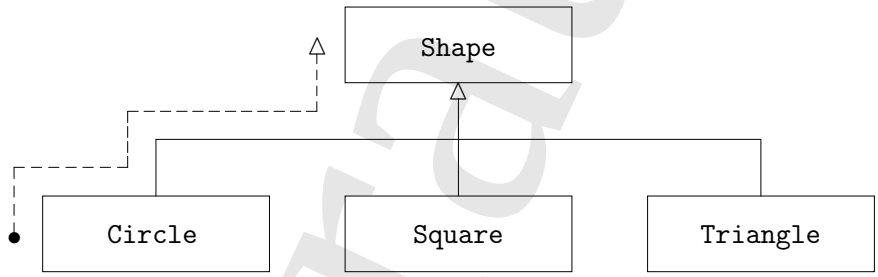
so bewirken die Methodenaufrufe unabhängig vom eigentlichen Typ des Objektes automatisch das korrekte Verhalten.

[58] Das ist ein erstaunlicher Kunstgriff. In der Zeile

```
doSomething(circle);
```

wird einer Methode, die ein Argument vom Typ `Shape` erwartet, ein Argument vom Typ `Circle` übergeben. Da ein `Circle`-Objekt auch dem Basistyp `Shape` angehört, kann es von der `doSomething()`-Methode wie ein `Shape`-Objekt behandelt werden. Ein `Circle`-Objekt akzeptiert also jede Benachrichtigung an die Schnittstelle der Klasse `Shape`. Somit ist die Übergabe eines Argumentes vom Typ `Circle` an eine Methode, die ein Argument vom Typ `Shape` erwartet, eine vollkommen sichere und logische Handlung.

[59] Die Behandlung eines Objektes (genauer, einer Referenz auf ein Objekt) eines abgeleiteten Typs als wäre es vom Basistyp wird als *aufwärtsgerichtete Typumwandlung* bezeichnet. Der Begriff „Umwandlung“ ist im Sinne von „umformen“ oder „in eine Form gießen“ zu verstehen und die Richtung „aufwärts“ bezieht sich auf die typische Darstellungsweise von Ableitungsdiagrammen, bei welcher der Basistyp oben steht und die abgeleiteten Klasse nach unten „wachsen“. Eine Typumwandlung in Richtung des Basistyps entspricht somit einer Bewegung nach oben im Ableitungsdiagramm:



Ein objektorientiertes Programm beinhaltet stets irgendwo eine aufwärtsgerichtete Typumwandlung, die den Programmierer davon entlastet, den exakten Typ eines verarbeiteten Argumentes kennen zu müssen, zum Beispiel in der Methode `doSomething()`:

```
shape.erase();
// ...
shape.draw();
```

Beachten Sie, daß die Methode den eigentlichen Typ Ihres Argumentes nicht explizit ermittelt und ihr Verhalten vom Ergebnis dieser Untersuchung abhängig macht. Eine Methode, die ihr Argument auf jeden möglichen Untertyp von `Shape` prüft, ist unübersichtlich und muß jedesmal überarbeitet werden, wenn Sie eine neue Klasse von `Shape` ableiten. Hier genügt die Information, daß das übergebene Objekt dem Typ `Shape` angehört. Sie wissen, daß sich das Objekt zeichnen und löschen kann und überlassen die Einzelheiten dem Objekt selbst.

[60] Das Beeindruckende an den Anweisungen in der `doSomething()`-Methode ist, daß irgendwie das Richtige geschieht. Das Aufrufen der `draw()`-Methode eines `Circle`-Objektes bewirkt selbstverständlich die Verarbeitung anderer Anweisungen, als das Aufrufen der `draw()`-Methode eines `Square`- oder `Line`-Objektes. Das Senden einer `draw()`-Benachrichtigung an die anonyme `Shape`-Schnittstelle bewirkt aber anhand des eigentlichen Untertyps von `Shape` das korrekte Verhalten. Dieser Effekt ist verblüffend, da der Compiler, wie zuvor bereits erwähnt, beim Übersetzen der

Methode `doSomething()` nicht genau „wissen“ kann, mit welchem Argumenttyp die Methode tatsächlich arbeiten wird. In der Regel würden Sie beim Aufrufen der `erase()`- und `draw()`-Methoden letztlich die Versionen aus der Basisklasse `Shape` erwarten, nicht aber die Versionen der spezifischen Unterklassen `Circle`, `Square` oder `Line`. Dennoch geschieht aufgrund des Polymorphieeffektes das Richtige. Compiler und Laufzeitumgebung kümmern sich um die Einzelheiten. Sie müssen an dieser Stelle nicht mehr wissen, als daß der Polymorphieeffekt existiert und wichtiger noch, wie Sie ihn im Design einsetzen können. Wenn Sie einem Objekt eine Benachrichtigung senden, tut das Objekt das Richtige, selbst wenn eine aufwärtsgerichtete Typumwandlung im Spiel ist.

2.8 Die universelle Basisklasse Object

[61] Ein seit der Einführung von C++ besonders markanter Aspekt der objektorientierten Programmierung ist die Frage, ob alle Klassen letztendlich von einer einzigen Basisklasse abgeleitet werden sollten. Bei Java und nahezu allen anderen objektorientierten Sprachen außer C++ lautet die Antwort „Ja“. Die ultimative Basisklasse von Java heißt schlicht und einfach `Object`. Es zeigt sich, daß eine auf einer einzigen Wurzelklasse aufgebaute Klassenhierarchie viele Vorteile hat.

[62] Alle Klassen einer Hierarchie über einer einzigen Wurzelklasse haben eine gemeinsame Schnittstelle und gehören somit demselben fundamentalen Typ an. Die Alternative (bei C++) besteht darin, daß nicht jede Klasse von einer gemeinsamen Basisklasse abstammt. Im Hinblick auf Rückwärtskompatibilität paßt dieser Ansatz besser zu C und kann als weniger restriktiv betrachtet werden. Wenn Sie andererseits in vollem Umfang objektorientiert programmieren wollen, müssen Sie eine eigene Hierarchie aufbauen, um den in anderen objektorientierten Programmiersprachen eingebauten Komfort zur Verfügung zu haben. Außerdem stützt sich jede neu eingebundene Klassenbibliothek auf eine andere inkompatible Schnittstelle. Die Anbindung der neuen Schnittstelle an Ihr Design erfordert Arbeitsaufwand und eventuell Mehrfachvererbung. Lohnt sich diese Anstrengung relativ zur zusätzlichen Flexibilität bei C++? Der Aufwand rentiert sich, wenn Sie die Flexibilität tatsächlich brauchen, etwa weil Sie bereits viel Entwicklungsarbeit in C investiert haben. Wenn Sie allerdings am Reißbrett beginnen, kann eine Alternative wie Java häufig produktiver sein.

[63] Jede Klasse einer Hierarchie über einer einzigen Wurzelklasse kann garantiert mit einer gewissen Mindestfunktionalität ausgestattet werden. Sie wissen, daß bestimmte grundlegende Operationen auf jedem Objekt Ihres Programms ausführbar sind. Alle Objekte können einfach im dynamischen Speicher erzeugt werden und die Übergabe von Argumenten ist erheblich vereinfacht.

[64] Einer Klassenhierarchie über ein einzigen Wurzelklasse erleichtert die Implementierung einer automatischen Speicherbereinigung erheblich. Die automatische Speicherbereinigung ist eine fundamentale Verbesserung von Java im Vergleich mit C++. Da jedes Objekt garantiert Informationen zu seinem Typ beinhaltet, stehen Sie niemals vor der Situation, den Typ eines Objektes nicht bestimmen zu können. Dies ist bei systemnahen Operationen wie der Ausnahmebehandlung besonders wichtig und ermöglicht mehr Flexibilität in der Programmierung.

2.9 Die Containerklassen der Standardbibliothek

[65] Im allgemeinen wissen Sie nicht, wieviele Objekte Sie brauchen oder wie lange die Objekte benötigt werden, um ein bestimmtes Problem zu lösen. Ebenso wissen Sie nicht, wie Sie die Objekte speichern sollen. Wie könnten Sie wissen, wieviel Arbeitsspeicher Sie anfordern müssen, wenn diese Information erst zu Laufzeit verfügbar ist?

[66] Die Lösung der meisten Probleme beim objektorientierten Design scheint leichtfertig: Sie definieren eine neue Klasse, deren Objekte weitere Objekte (Elemente) referenzieren. Natürlich können Sie dasselbe auch mit Hilfe eines Arrays bewerkstelligen, das bei den meisten Sprachen zur Verfügung steht. Ein Objekt der neuen Klasse, im allgemeinen als *Containerklasse* bezeichnet expandiert allerdings bei Bedarf automatisch, um die übergebenen Elemente aufnehmen zu können. (Die Containerklassen werden auch als Kollektionsklassen oder kürzer *Kollektionen* bezeichnet, wobei die Standardbibliothek von Java den Begriff „Kollektion“ in einem anderen Sinne verwendet. Ich habe mich in diesem Buch für den Begriff „Container“ entschieden.) Sie brauchen also nicht zu wissen, wieviele Elemente Sie in einen Container einsetzen werden. Sie erzeugen das Containerobjekt einfach und überlassen ihm die Einzelheiten.

[67] Eine gute objektorientierte Programmiersprache stellt eine Reihe verschiedener Containerklassen zur Verfügung, bei C++ etwa, ist dies der häufig als *Standard Template Library* (STL) bezeichnete Teil der Standardbibliothek. Smalltalk hat eine sehr umfangreiche Containerbibliothek. Auch die Standardbibliothek von Java beinhaltet zahlreiche Containerklassen. Bei den Bibliotheken einiger Sprachen genügen eine oder zwei generische Containerklassen, um alle Bedürfnisse abzudecken. Bei anderen Sprachen, darunter auch Java, enthält die Containerbibliothek dagegen verschiedene Klassen für die unterschiedlichsten Anforderungen: Es gibt verschiedene Klassen für Listen (zur Aufnahme von Referenzen), Schlüssel/Wert-Paare (*assoziative Arrays* zur Verknüpfung von Objekten mit anderen Objekten), Mengen (enthalten höchstens ein Exemplar jedes Elementes) sowie weitere Datenstrukturen wie Warteschlangen, Bäume, Stapelspeicher und so weiter.

[68] Aus der Perspektive des Designs genügt ein abfrag- und änderbarer Container, um Ihr Problem zu lösen. Würde ein einzelner Containertyp alle Anforderungen erfassen, so bestünde kein Bedarf für unterschiedliche Typen. Wahlmöglichkeiten zwischen verschiedenen Containertypen sind aber aus zwei Gründen wichtig. Erstens haben die einzelnen Containerklassen verschiedene Schnittstellen und unterschiedliches externes Verhalten. Ein Stapelspeicher hat eine andere Schnittstelle und ein anderes Verhalten als eine Warteschlange, die sich wiederum von einer Menge oder Liste unterscheidet. Eine dieser Containerklassen gestattet unter Umständen eine flexiblere Lösung Ihres Problems als eine andere. Zweitens sind verschiedene Container bei bestimmten Operationen verschieden effizient. Es gibt beispielsweise zwei grundlegende Containerklassen für Listen: `ArrayList` und `LinkedList`. Beide Klassen repräsentieren eine einfache Folge von Elementen und haben identische Schnittstellen sowie identisches externes Verhalten. Bestimmte Operationen verursachen allerdings signifikant verschiedene Kosten. Das Abfragen oder Ändern eines Elementes an einer beliebigen Position ist beim Containertyp `ArrayList` eine Operation mit konstantem Zeitaufwand, das heißt die benötigte Zeit ist stets gleich, unabhängig von der gewählten Position. Das Durchlaufen der gesamten Liste bis zu einem beliebig gewählten Element ist dagegen beim Containertyp `LinkedList` teuer, und dauert umso länger, je näher sich das Element am Listenende befindet. Andererseits ist das Einsetzen eines Elemente in die Mitte der Liste bei `LinkedList` günstiger als bei `ArrayList`. Diese und weitere Operationen sind bei verschiedenen Containertypen je nach unterliegener Datenstruktur der Folge verschieden effizient. Vielleicht beginnen Sie mit einem Container vom Typ `LinkedList` und entscheiden sich später, bei der Arbeit an der Performanz des Programmes für `ArrayList`. Die Abstraktion in Gestalt des Interfaces *`List`* gestattet das einfache Austauschen des Containertyps bei minimalen Auswirkungen auf Ihren Quelltext.

2.9.1 Parametrisierte Typen (Generische Typen)

[69] Vor Version 5 der Java Standard Edition (SE 5) enthielten Container nur Elemente der universellen Basisklasse `Object`. Da die gesamte Klassenhierarchie von Java auf der einzigen Wurzelklasse `Object` aufgebaut ist, konnte ein Container, der Elemente vom Typ `Object` enthielt, Objekte jedes

beliebigen Typs[7] aufnehmen. Die Containerklassen waren dadurch einfach wiederzuverwenden.

[70] Ein solcher Container wurde genutzt indem Sie einfach Objektreferenzen einsetzten und später wieder abfragten. Da der Container aber Elemente vom Typ `Object` speicherte, wurde eine, dem Container übergebene Objektreferenz aufwärts in `Object` umgewandelt, wobei der eigentliche Typ verlorenging. Eine zu einem späteren Zeitpunkt abgefragte Referenz hatte den Typ `Object` und nicht mehr ihren ursprünglichen Typ. Wie konnte aber der eigentliche Typ, des dem Container übergebenen Elementes wieder hergestellt werden?

[71] Hierfür war eine weitere Typumwandlung notwendig, diesesmal aber nicht aufwärts bezüglich der Ableitungshierarchie, also hin zu einem allgemeineren Typ, sondern abwärts, hinzu einem spezielleren Typ. Eine solche Umwandlung wird als *abwärtsgerichtete Typumwandlung* bezeichnet. Bei einer aufwärtsgerichteten Typumwandlung ist bekannt, daß beispielsweise ein `Circle`-Objekt auch dem Typ `Shape` angehört, die Umwandlung also typsicher ist. Im Gegensatz dazu ist keinesfalls sicher, daß eine Referenz vom Typ `Object` auf ein Objekt vom Typ `Circle` oder `Shape` verweist. Eine abwärtsgerichtete Typumwandlung ist nur sicher, wenn Sie genau wissen, mit welchem Objekttyp Sie umgehen.

[72] Die Gefahr einer abwärtsgerichteten Typumwandlung mit falschem Zieltyp war aber dadurch begrenzt, daß Sie gegebenenfalls zur Laufzeit eine Fehlermeldung mit einer kurzen Beschreibung erhielten, eine sogenannte *Ausnahme*. Beim Abfragen einer Referenz aus einem Container war stets eine exakte Typinformation erforderlich, um die abwärtsgerichtete Typumwandlung korrekt ausführen zu können.

[73] Abwärtsgerichtete Typumwandlungen erfordern stets zusätzliche Programmlaufzeit und zusätzliche Bemühungen von seiten des Programmierers. Wäre es nicht sinnvoll, eine Containerklasse so zu entwickeln, daß die Typinformation der Elemente erhalten bleibt und die Notwendigkeit der abwärtsgerichteten Typumwandlung samt eventueller Fehler auszuschalten? Die Lösung ist der Mechanismus des parametrisierten Typs. Ein *parametrisierter Typ* oder auch *generischer Typ* ist eine Klasse, die der Compiler automatisch an die Verwendung eines bestimmten *Parametertyps* anpassen kann. Der Compiler kann beispielsweise eine parameterisierte Containerklasse so anpassen, daß das erzeugte Containerobjekt nur Elemente des Parametertyps `Shape` akzeptiert und wieder herausgibt.

[74] Die Aufnahme parametrisierter Typen in den Sprachumfang, der sogenannten *Java Generics*, war eine der großen Änderungen der SE 5. Sie erkennen die Verwendung eines parametrisierten Typs an den eckigen Klammern, zwischen denen der Parametertyp notiert wird. Die folgende Zeile erzeugt beispielsweise einen Container vom Typ `ArrayList` für Elemente vom Typ `Shape`:

```
ArrayList<Shape> shapes = new ArrayList<Shape>();
```

Viele Komponenten der Standardbibliothek von Java wurden überarbeitet, um die Vorteile durch parametrisierten Typen nutzen zu können. Wie Sie sehen werden, haben die parametrisierten Typen Auswirkungen auf viele Beispiele in diesem Buch.

2.10 Erzeugung und Lebensdauer von Objekten

[75] Die Art und Weise der Objekterzeugung und -zerstörung ist beim Umgang mit Objekten ein entscheidendes Thema. Jedes Objekt braucht Resourcen, um existieren zu können, vor allem Arbeitsspeicher. Ein nicht mehr benötigtes Objekt muß aufgeräumt werden, damit die beanspruchten Resourcen zur Wiederverwendung freigegeben werden können. Ein einfachen Fällen ist die Frage,

[7] Java-Container können keine Elemente primitiven Typs enthalten, wobei diese Einschränkung durch das Autoboxing ab der SE 5 praktisch verschwindet. Dieser Aspekt wird später in diesem Buch in allen Einzelheiten diskutiert.

wie ein Objekt aufgeräumt werden sollte, nicht besonders spannend: Das Objekt wird erzeugt, solange als erforderlich verwendet und sollte anschließend zerstört werden. Es ist aber nicht schwierig, in eine kompliziertere Situation zu geraten.

[76] Stellen Sie zum Beispiel vor, Sie entwickeln ein Design für die Verwaltung der Flugzeuge an einem Flughafen. (Dasselbe Modell funktioniert auch beim Verwalten der Kisten in einem Lager, bei einem Videoverleih oder einer Pension für Haustiere.) Die Aufgabe wirkt auf den ersten Blick einfach: Sie legen einen Container für die Flugzeuge an, erzeugen für jede Maschine beim Eintritt in den Kontrollbereich ein neues Objekt und deponieren es in diesem Container. Verläßt eine Maschine den Kontrollbereich, so wird das entsprechende Objekt aufgeräumt.

[77] Eventuell gibt es aber noch ein zweites System, das Informationen über die Maschinen aufzeichnet, beispielsweise Daten die keiner sofortigen Aufmerksamkeit durch die Hauptsteuerung bedürfen, wie ein Verzeichnis der Flugpläne der kleineren Maschinen, die den Flugplatz verlassen. In diesem Fall existiert ein zweiter Container für kleine Maschinen und jedes Mal, wenn Sie ein neues Objekt für ein in den Kontrollbereich eintretendes Flugzeug erzeugen, tragen Sie das Objekt auch in den zweiten Container ein, wenn es keine kleine Maschine ist. Ein Hintergrundprozeß führt eine Operation auf den gespeicherten Objekten durch, wenn die Anwendung unter geringer Auslastung läuft.

[78] Nun ist die Situation etwas komplizierter: Wie können Sie feststellen, wann eines der Objekte aufgeräumt werden kann? Während Sie die Verarbeitung eines Objektes beendet haben, kann ein anderer Teil des Systems das Objekt noch brauchen. Dasselbe Problem ergibt sich in zahlreichen anderen Situationen und ist in Programmiersprachen wie C++, in denen ein Objekt nach seiner Verwendung explizit zerstört werden muß, eine komplexe Aufgabe.

[79] Wo befinden sich die Daten eines Objektes und wie wird die Lebensdauer eines Objektes kontrolliert? Aus der Perspektive von C++ hat die Kontrolle über die Effizienz höchste Priorität. Der Programmierer steht somit vor einer Entscheidung: Um maximale Verarbeitungsgeschwindigkeit zu erreichen, können der Bereich des Arbeitsspeichers und die Lebensdauer des Objektes programmatisch bestimmt werden, indem die Objekte auf dem Aufrufstapel (gelegentlich als *automatische Variablen* oder *scoped variables* bezeichnet) oder im statischen Bereich des Speichers deponiert werden. Dadurch erhält die Geschwindigkeit des Allokierens und der Freigabe von Arbeitsspeicher Priorität und diese Steuerbarkeit kann sich in gewissen Situationen als sehr wertvoll erweisen. Andererseits opfern Sie mit diesem Ansatz aber Flexibilität, da der exakte Speicherbedarf, die Lebensdauer und der Objekttyp bereits während der Entwicklung des Programms bekannt sein müssen. Bei allgemeineren Problemen, wie einer CAD-Anwendung (Computer Aided Design), der Verwaltung eines Lagers oder der Flugzeuge am Flughafen, bedeutet eine solche Vorgabe eine zu starke Einschränkung.

[80] Die andere Möglichkeit besteht darin, Objekte im dynamischen Speicher zu erzeugen Bei diesem Verfahren wissen Sie erst zur Laufzeit, wieviele Objekte Sie brauchen, welche Lebensdauer sie haben oder welcher exakte Objekttyp vorliegt. Diese Informationen werden spontan ermittelt, während das Programm läuft. Wenn Sie ein neues Objekt brauchen, erzeugen Sie es an der Stelle, an der Sie es benötigen, im dynamischen Speicher. Aufgrund der dynamischen Speicherverwaltung zur Laufzeit, kann das Allokieren von Arbeitsspeicher im dynamischen Speicher deutlichlich länger dauern, als das Erzeugen von Speicher auf dem Aufrufstapel. Häufig genügt zum Erzeugen von Arbeitsspeicher auf dem Aufrufstapel eine einzige Assembleranweisung, um den Stapelzeiger nach unten zu bewegen und eine weitere, um ihn wieder zurückzusetzen. Der Zeitaufwand, um Speicher im dynamischen Speicher zu allokieren, hängt vom Design des entsprechenden Mechanismus' ab.

[81] Der dynamische Ansatz setzt die im allgemeinen vernünftige Annahme voraus, daß Objekte tendentiell kompliziert sind, so daß der zusätzliche Aufwand durch die Suche nach und die Freigabe

von Arbeitsspeicher beim Erzeugen eines Objektes nicht wesentlich ins Gewicht fällt. Außerdem ist die größere Flexibilität wichtig, to solve the general programming problem.

[82] Java allokiert Arbeitsspeicher ausschließlich dynamisch.[8] Jedesmal, wenn Sie ein Objekt erzeugen, verwenden Sie den `new`-Operator.

[83] Die Lebensdauer eines Objektes ist ein Thema für sich. Bei Sprachen, die das Erzeugen von Objekten auf dem Aufrufstapel erlauben, bestimmt der Compiler die Lebensdauer und kann das Objekt automatisch zerstören. Erzeugen Sie dagegen ein Objekt im dynamischen Speicher, so hat der Compiler keine Informationen über dessen Lebensdauer. Bei einer Sprache wie C++ müssen Sie den Zeitpunkt der Zerstörung eines Objektes programmatisch bestimmen, wobei Speicherlecks entstehen können, wenn Sie einen Fehler machen (ein häufiges Problem bei Programmen in C++). Java verfügt über eine *automatische Speicherbereinigung*, die selbsttätig erkennt, wenn ein Objekt nicht mehr verwendet wird und dessen Zerstörung einleitet. Eine automatische Speicherbereinigung ist ein komfortabler Mechanismus, da sie die erforderlichen Anweisungen und die Anzahl der Dinge reduziert, die Sie im Auge behalten müssen. Die Speicherbereinigung liefert, wichtiger noch, ein erheblich höheres Niveau an Sicherheit vor dem heimtückischen Problem von Speicherlecks, das viele Projekte in C++ in die Knie gezwungen hat.

[84] Die automatische Speicherbereinigung von Java hat entwurfsbedingt die Aufgabe, sich dem Problem der Freigabe von Arbeitsspeicher anzunehmen (auch wenn andere Aspekte beim Aufräumen eines Objektes hiervon unberücksichtigt bleiben). Die Speicherbereinigung „weiß“, wann ein Objekt nicht länger benötigt wird und gibt den von diesem Objekt beanspruchten Arbeitsspeicher automatisch frei. Die automatische Speicherbereinigung macht, zusammen mit der Tatsache, daß alle Objekte von einer einzigen Wurzelklasse abgeleitet sind und daß Sie Objekt nur auf eine einzige Weise erzeugen können, nämlich im dynamischen Speicher, die Programmierung in Java viel einfacher in C++. Sie müssen viel weniger Entscheidungen treffen und Hürden überwinden.

2.11 Ausnahmebehandlung

[85] Seit den Anfängen der ersten Programmiersprachen ist die Behandlung von Fehlern eine besonders schwierige Aufgabe. Daher wird dieser Aspekt bei vielen Sprachen einfach nicht beachtet und an die Designer der Bibliotheken weitergegeben. Diese wiederum warten mit halbfertigen Maßnahmen auf, die zwar in vielen Situationen funktionieren, sich aber mühelos umgehen lassen, zum Beispiel in dem man sie einfach ignoriert. Ein wesentliches Problem der meisten Fehlerbehandlungsansätze besteht darin, daß sie sich auf die Wachsamkeit des Programmierers verlassen müssen, sich an eine festgesetzte Vereinbarung zu halten, die nicht von der Sprache erzwungen wird. Läßt die Wachsamkeit des Programmierers nach, zum Beispiel wenn Eile geboten ist, so werden diese Konventionen leicht vergessen.

[86] Die Ausnahmebehandlung verbindet die Fehlerbehandlung direkt mit der Programmiersprache und gelegentlich sogar mit dem Betriebssystem. Eine Ausnahme ist ein Objekt, welches von der Fehlerquelle „ausgeworfen“ (oder „hervorgerufen“) wird und von einem passenden Ausnahmebehandler „abgefangen“ werden kann, der zur Behandlung dieses Fehlertyps geschrieben wurde. Die Ausnahmebehandlung definiert gewissermaßen einen alternativen, parallelen Ausführungspfad, der beim Auftreten eines Fehlers eingeschlagen wird. Da die Ausnahmebehandlung einen separaten Ausführungspfad wählt, wirkt sie sich nicht störend auf die Anweisungen des normalen Ausführungspfades aus. Die Trennung der Ausführungspfade erleichtert das Schreiben des Quelltextes, da Sie nicht permanent gezwungen sind, auf Fehler zu prüfen. Eine Ausnahme unterscheidet sich in einer wesentlichen Hinsicht von einem zurückgegebenen Fehlerwert oder einem gesetzten Flag, mit dem eine

[8] Mit Ausnahme der primitiven Typen, über die Sie später mehr lernen.

Methode eine Fehlersituation anzeigt. Fehlerwerte und Flags können ignoriert werden, nicht aber eine Ausnahme. Eine Ausnahme wird somit garantiert irgendwo behandelt oder gemeldet. Darüber hinaus bieten Ausnahmen eine Möglichkeit, um nach dem Auftreten eines Fehlers wieder einen funktionsfähigen Programmzustand herzustellen. Statt das Programm einfach zu beenden, besteht häufig die Möglichkeit die Dinge zu korrigieren und die Verarbeitung wieder aufzunehmen, wodurch das Programm robuster wird.

[87] Die Ausnahmebehandlung von Java ragt über andere Programmiersprachen hinaus, weil das Konzept von Anfang an zum Sprachumfang gehörte und seine Anwendung erzwungen wird. Es ist die einzig akzeptable Möglichkeit, um Fehler zu berichten. Wenn Sie Ausnahmen nicht typgerecht behandeln, gibt der Compiler eine Fehlermeldung aus. Diese garantierte Konsistenz kann die Fehlerbehandlung erheblich erleichtern.

[88] Beachten Sie, daß die Ausnahmebehandlung kein Konzept im Bereich der objektorientierten Programmierung ist, obwohl Ausnahmen bei den meisten objektorientierten Sprachen durch Objekte repräsentiert werden. Das Konzept der Ausnahmebehandlung existierte bereits vor der objektorientierten Programmierung.

2.12 Threadprogrammierung

[89] Die gleichzeitige Verarbeitung mehr als einer Aufgabe ist ein fundamentales Konzept in der Computerprogrammierung. Bei vielen Programmierproblemen ist es erforderlich, das Programm während seiner Verarbeitung anzuhalten, ein bestimmtes Problem zu lösen und anschließend zur Verarbeitung des Hauptprozesses zurückzukehren. Es gibt zahlreiche Ansätze zur Lösung dieses Problems. Anfangs schrieben Programmierer mit systemnahen Kenntnissen über die unterliegende Architektur Unterbrechungsroutinen (Interrupthandler) und das Anhalten des Hauptprozesses wurde per Hardwareinterrupt ausgelöst. Dieser Ansatz funktionierte zwar gut, war aber kompliziert und nicht portabel, so daß die Übertragung eines Programmes auf eine neue Architektur langsam und kostspielig war.

[90] Hin und wieder sind Interrupts zur Behandlung zeitkritischer Aufgaben notwendig. Es gibt aber auch eine umfangreiche Klasse von Problemen bei denen Sie lediglich versuchen, das Problem in separat lauffähige Teile zu zerlegen, um die Reaktionsfähigkeit des Programms zu verbessern. Die innerhalb eines Programms separat lauffähigen Teile heißen *Threads* beziehungsweise *Aufgaben* und das allgemeine Konzept *Threadprogrammierung*. Graphische Benutzerschnittstellen sind ein gängiges Beispiel für Threadprogrammierung. Mit Hilfe von Threads kann der Benutzer eine Schaltfläche betätigen und eine schnelle Antwort bekommen, statt gezwungermaßen zu warten, bis das Programm seine aktuelle Tätigkeit beendet hat.

[91] Aufgaben sind in der Regel nur eine Möglichkeit, um Zeit bei einem einzelnen Prozessor zu kontingentieren. Unterstützt das Betriebssystem aber mehrere Prozessoren, so kann jede Aufgabe einem anderen Prozessor zugewiesen werden und es können tatsächlich mehrere Aufgabe gleichzeitig verarbeitet werden. Eine komfortable Eigenschaft der Threadprogrammierung auf Sprachebene ist, daß sich der Programmierer nicht darum kümmern muß, ob mehrere Prozessoren oder nur ein Prozessor vorhanden ist. Das Programm ist logisch in Aufgaben unterteilt und wenn der Rechner mehr als einen Prozessor besitzt, wird das Programm schneller verarbeitet, ohne daß spezielle Anpassungen benötigt werden.

[92] All dies erweckt den Eindruck, daß die Threadprogrammierung eine recht einfache Angelegenheit ist. Die Sache hat allerdings einen Haken: Die gemeinsame Nutzung von Resourcen. Erwartet mehr als eine Aufgabe Zugriff auf dieselbe Resource, so stehen Sie vor einem Problem. Beispielsweise können nicht zwei Prozesse zugleich Daten an einen Drucker senden. Die Lösung des Problems

besteht darin, eine gemeinsam verwendete Resource wie den Drucker während ihrer Verwendung zu sperren. Eine Aufgabe sperrt eine Resource, verrichtet ihre Arbeit und hebt die Sperre anschließend wieder auf, damit die Resource wieder verfügbar wird.

[93] Die Threadunterstützung ist bei Java in die Sprache eingebaut. Die SE 5 hat die Standardbibliothek im Hinblick auf Threadunterstützung erheblich erweitert.

2.13 Java und das Internet

[94] Wenn Java, im Grunde genommen, nur eine weitere Programmiersprache ist, so ist die Frage berechtigt, warum Java als so wichtig und revolutionärer Schritt in der Programmierung von Computern angepriesen wird. Die Antwort ist nicht unmittelbar offensichtlich, wenn Ihr Hintergrund in der traditionellen Programmierung liegt. Java ist zwar auch beim Lösen traditioneller Programmierprobleme nützlich, spielt aber eine wichtige Rolle bei Problemen im Kontext des Internets.

2.13.1 Was ist das Internet?

[95] Das Internet wirkt auf den ersten Blick und bei allem Gerede über „surfen“, „Präsenz“ und „Homepages“ geheimnisvoll. Es hilft, einen Schritt zur Seite zu treten, um festzustellen, worum es tatsächlich geht. Sie müssen hierzu aber Client/Server-Systeme verstehen, ein weiteres Konzept voller verwirrender Eigenschaften und Fähigkeiten.

2.13.1.1 Das Client/Server-Konzept

[96] Die Grundidee des Client/Server-Konzeptes ist die zentrale Speicherung von Informationen beliebiger Art, in der Regel in einer Datenbank, die bei Bedarf an bestimmte Personen oder Rechner verteilt werden. Der Schlüssel zum Client/Server-Konzept besteht in der zentralisierten Informationsspeicherung, so daß Änderungen des Datenbestandes an nur einer zentralen Stelle vorgenommen werden müssen und von dort an die Konsumenten der Daten verbreitet werden. Der Datenspeicher, die Software zur Verteilung dieser Daten und der beziehungsweise die Rechner auf denen Datenspeicher und Software installiert sind, werden zusammengenommen als *Server* bezeichnet. Die auf dem Rechner des Konsumenten installierte Software, die mit dem Server kommuniziert, Informationen abfragt, verarbeitet und dem Konsumenten anzeigt, heißt *Client*.

[97] Das grundlegende Client/Server-Konzept ist also nicht besonders kompliziert. Die Probleme beginnen, wenn ein einzelner Server mehrere Clients zu gleich bedienen muß. Im allgemeinen ist ein Datenbankmanagementsystem beteiligt und ein Datenbankdesigner balanciert die Struktur der Daten in Form von Tabellen optimal aus. Client/Server-Systeme gestatten dem Client häufig, über den Server neue Daten zu erfassen. Dabei müssen Sie gewährleisten, daß die von einem Client neu übergebenen Daten nicht von einem anderen Client überschrieben werden oder neu erfaßte Daten bei der Übertragung in die Datenbank nicht verloren gehen (Transaktionsverarbeitung). Ändert sich die Clientsoftware, so muß sie neu übersetzt, auf Fehler geprüft und auf den Clientrechnern installiert werden. Dieser Vorgang ist komplizierter und aufwändiger als Sie vielleicht annehmen, wobei die Unterstützung verschiedener Rechnerarchitekturen und Betriebssysteme besonders problematisch ist. Schließlich gibt es noch den äußerst wichtigen Aspekt der Performanz: Hunderte von Clients können jederzeit Anfragen an den Server senden, so daß sich kleine Verzögerungen entscheidend auswirken können. Die Programmierer arbeiten hart daran, die Verarbeitung von Teilaufgaben auszulagern,

um die Wartezeit zu minimieren. Teilaufgaben werden häufig an den Clientrechner übertragen, gelegentlich aber auch, mittels sogenannter *Middleware* (zur Verbesserung der Wartbarkeit), an andere serverseitige Rechner.

[98] Die einfache Idee der Informationsverteilung per Client/Server-Konzept hat derart viele Komplexitätsebenen, daß das Gesamtproblem hoffnungslos undurchdringlich scheint. Dennoch ist der Ansatz sehr wichtig: Das Client/Server-Konzept bestreitet etwa die Hälfte aller Programmierarbeiten und durchdringt alle Bereiche von der Aufnahme von Bestellungen und Kreditkartentransaktionen bis hin zur Verteilung beliebiger Daten, etwa an der Börse, in der Wissenschaft und im öffentlichen Dienst. In der Vergangenheit wurden individuelle Lösungen für individuelle Probleme gesucht und jedesmal eine neue Lösung erfunden. Diese Lösungen waren schwierig zu entwickeln und anzuwenden und der Anwender mußte sich bei jedem Programm mit einer neuen Schnittstelle vertraut machen. Das Client/Server-Problem *an sich* bedurfte einer neu durchdachten Lösung.

2.13.1.2 Das Internet als gewaltiger Server

[99] Das Internet ist eigentlich ein gigantisches Client/Server-System. Schlimmer noch, koexistieren sämtliche Server und Clients zugleich in einem einzigen Netzwerk. Allerdings brauchen Sie sich über diesen Punkt keine Gedanken zu machen, da Sie stets nur mit einem einzigen Server Verbindung aufnehmen und kommunizieren (auch wenn Ihre Anfragen auf der Suche nach den richtigen Server eventuell um die ganze Welt laufen).

[100] Anfangs war eine solche Anfrage ein Einwegvorgang. Sie richteten eine Anfrage an einen Server und dieser übergab Ihnen eine Datei, die von einem Programm auf Ihrem lokalen Rechner (das heißt einem Client) interpretiert wurde, beispielsweise vom Webbrowser formatiert. Es verging nur wenig Zeit, bis die Anwender mehr Funktionalität verlangten, als nur Dateien von einem Server herunterzuladen. Die Anwender wollten komplette Client/Server-Funktionalität, so daß ein Client auch Informationen an Server zurück übertragen konnte, zum Beispiel um serverseitige Datenbankanfragen zu senden, serverseitig neue Informationen anzulegen oder Aufträge zu erteilen (wodurch wiederum spezielle Sicherheitsvorkehrungen notwendig wurden). Diese Änderungen können in der Entwicklung des Internets nachvollzogen werden.

[101] Der Webbrowser war ein großer Schritt vorwärts: Das Konzept, daß ein Stück Information ohne Änderungen auf einem beliebigen Rechner darstellt werden konnte. Die ursprünglichen Webbrowser waren allerdings recht schlicht und gingen angesichts der an sie gestellten Anforderungen schnell in die Knie. Sie waren nicht besonders interaktiv und neigten dazu, Server und Internet zu verstopfen, da bei jedem Schritt der eine Berechnung erforderte, Informationen zur Verarbeitung an den Server zurückgesendet werden mußten. Es konnten Sekunden oder gar Minuten vergehen, nur um festzustellen, daß Ihnen in der Anfrage ein Tippfehler unterlaufen war. Als reine Anzeigekomponente war der Webbrowser nicht in der Lage auch nur die einfachsten Berechnungen selbst auszuführen. (Andererseits bot ein solcher Webbrowser auch Sicherheit, da er keine Programme auf dem lokalen Clientrechner ausführen konnte, die eventuell Fehler oder Viren enthielten.)

[102] Zahlreiche Ansätze wurden zur Lösung dieses Problems verfolgt. Zunächst wurden standardisierte Graphikformate weiterentwickelt, um bessere Animationen und Videos innerhalb des Webbrowsers zu ermöglichen. Die verbleibenden Probleme können nur durch *clientseitige Programmierung* gelöst werden, das heißt unter Einbeziehung der Fähigkeit, Programme auf dem Clientrechner vom Webrowser aus starten zu können.

2.13.2 Clientseitige Programmierung

[103] Das ursprüngliche Server/Webbrowser-Design des Internets gestattet zwar interaktive Inhalte, aber die Interaktivität wurde ausschließlich über den Server bewerkstelligt. Der Server erzeugte statische Seiten, die der clientseitige Webbrowser nur interpretierte und anzeigte. Die *Hypertext Markup Language* (HTML) bietet schlichte Möglichkeiten zur Datenerfassung: Texteingabefelder, Ankreuzfelder, Radiobuttons, Listen und Dropdown-Listen sowie Schaltflächen, um die Einträge eines Formulars zurückzusetzen beziehungsweise die erfaßten Daten an den Server zurückzusenden. Die mit der zurückgesendeten Anfrage übertragenen Daten teilen dem CGI-Protokoll mit, was zu tun ist. Die häufigste Aktion ist das Aufrufen eines serverseitigen Programms, welches in einem, typischer *cgi-bin* genannten Verzeichnis liegt. (Wenn Sie beim Versenden eines Formulars auf die Adresszeile Ihres Webbrowsers achten, können Sie hin und wieder die Zeichenkette „cgi-bin“ in der Anfrage-URL erkennen.) Derartige Programme können in den meisten Sprachen geschrieben werden. Perl ist eine häufige Wahl, da die Sprache auf Textverarbeitung spezialisiert ist und interpretiert wird, so daß sie auf jedem Server unabhängig von Prozessor oder Betriebssystem installiert werden kann. Python (*http://www.python.com*) stellt aufgrund seiner Mächtigkeit und Einfachheit allerdings eine Konkurrenz für Perl dar.

[104] Viele leistungsfähige Webauftritte bauen heute strikt auf CGI auf und Sie können nahezu alles mit CGI bewerkstelligen. Die Pflege auf CGI-Programmen aufbauender Webauftritte kann allerdings schnell übermäßig kompliziert werden. Außerdem besteht nach wie vor das Problem der Antwortzeit. Die Antwort auf eine Anfrage an ein CGI-Programm hängt vom zu übertragenden Datenvolumen sowie der Auslastung von Server und Internet ab. (Darüber hinaus ist der Start eines CGI-Programms tendentiell langsam.) Die ursprünglichen Designer des Internets haben nicht vorausgesehen, wie schnell die Bandbreite durch die unterschiedlichen Anwendungen erschöpft sein würde. Dynamische graphische Darstellungen beispielsweise, sind beinahe unmöglich zu leisten, da pro Ansicht eine *.gif* Datei (Graphics Interchange Format) generiert und vom Server zum Client übertragen werden müßte. Sie haben außerdem mit Sicherheit bereits einen Validierungsvorgang für Formulardaten in einem Webbrowser beobachtet: Sie betätigen die „Submit“-Schaltfläche eines Formulars, woraufhin die Daten an den Server gesendet werden. Der Server startet ein CGI-Programm, welches einen Fehler feststellt, eine HTML-Seite mit einer entsprechenden Fehlermeldung erzeugt und an Sie zurücksendet. Sie müssen die Daten erneut eingeben und einen weiteren Versuch unternehmen. Diese Vorgehensweise ist nicht nur langsam, sondern schlicht und einfach nicht elegant.

[105] Die Lösung ist clientseitiges Programmieren. Die meisten Desktoprechner auf denen Webbrower laufen, sind leistungsfähige Rechner, die beim ursprünglichen Ansatz mit statischen HTML-Seiten ungenutzt laufen und untätig darauf warten, daß der Server die nächste Seite sendet. Clientseitige Programmierung bedeutet, dem Webbrowser soviel Arbeit wie möglich zu übertragen. Aus der Perspektive des Benutzers ist die Anwendung erheblich schneller und gestattet mehr Interaktion mit Ihrem Webauftritt.

[106] Diskussionen über clientseitige Programmierung unterscheiden sich kaum von Diskussion über Programmierung im allgemeinen. Die Parameter sind größtenteils identisch, aber die Plattform ist verschieden. Ein Webbrowser ist wie ein begrenztes Betriebssystem. Letztendlich müssen Sie nach wie vor programmieren und hieraus erwächst eine atemberaubende Menge von Problemen und Lösungen. Dieser Unterabschnitt gibt einen Überblick über die Probleme und Ansätze der clientseitigen Programmierung.

2.13.2.1 Plugins („Erweiterungsmodule“)

[107] Das Konzept des Plugins ist einer der wichtigsten Schritte vorwärts in der clientseitigen Programmierung. Das Plugin gestattet dem Programmierer, die Funktionalität des Webbrowsers durch ein Stück Software zu erweitern, welches sich nach dem Herunterladen von selbst im Webbrowser installiert. Von diesem Zeitpunkt an unterstützt der Webbrowser die neue Funktionalität. (Ein Plugin muß nur einmal heruntergeladen werden.) Plugins erweitern Webbrowser um schnelles und leistungsfähiges Verhalten. Das Entwickeln eines Plugins ist allerdings keine leichte Aufgabe und nichts, was Sie beim Aufsetzen eines Webauftritts nebenbei erledigen können. Der Wert des Plugins für die clientseitige Programmierung besteht darin, daß ein Experte Erweiterungen entwickeln und diese Erweiterungen im Webbrowser ohne Erlaubnis des Anbieters installieren kann. Plugins sind eine „Hintertür“, die die Entwicklung neuer clientseitiger Programmiersprachen ermöglicht (obwohl nicht alle Sprachen in Form vom Plugins implementiert sind).

2.13.2.2 Skriptsprachen

[108] Plugins führten zur Entwicklung webbrowserinterpretierter Skriptsprachen. Eine solche Skriptsprache gestattet die Einbettung des clientseitigen Programms direkt in der HTML-Seite und das zur Interpretation benötigte Plugin wird beim Anzeigen der HTML-Seite automatisch aktiviert. Skriptsprachen sind in der Regel leicht zu verstehen. Ein in einer Skriptsprache geschriebenes Programm wird als einfacher Teil des Quelltextes einer HTML-Seite im Rahmen des Kontaktes mit Servers beim Anfordern der Seite schnell geladen. Der Nachteil ist, daß der Quelltext für jedermann lesbar ist und somit gestohlen werden kann. Im allgemeinen werden in Skriptsprachen aber keine staunenswerten anspruchsvollen Dinge implementiert. Der Nachteil hält sich also in Grenzen.

[109] Es gibt eine Skriptsprache, deren Unterstützung Sie von einem Webbrowser auch ohne Plugins erwarten können: JavaScript. (Diese Skriptsprache hat nur geringe Ähnlichkeit mit Java und Sie müssen zusätzlichen Aufwand investieren, um sie zu erlernen. Der Name wurde gewählt, um vom Marketingimpuls von Java zu profitieren.) Leider implementierten viele Anbieter von Webbrowsern JavaScript ursprünglich individuell und selbst verschiedene Versionen ein und desselben Webbrowsers waren uneinheitlich. Die Standardisierung von JavaScript in Form von ECMAScript war zwar eine Hilfe, es verging aber viel Zeit, bis sich die verschiedenen Anbieter von Webbrowsern an den neuen Standard hielten. (Der von Microsoft herausgegebene eigene VBScript-Standard, mit vagen Ähnlichkeiten zu JavaScript, hat die Vereinheitlichung natürlich nicht unterstützt.) Im allgemeinen müssen Sie beim Programmieren in JavaScript nach dem Prinzip des kleinsten gemeinsamen Nenners vorgehen, damit das Skript in jedem Webbrowser läuft. Die Fehlersuche und -behandlung bei JavaScript kann nur als heilloses Durcheinander beschrieben werden. Man nehme die Tatsache, daß erst in jüngster Zeit jemand ein komplexes Stück Software in JavaScript geschrieben hat (GMail bei Google), wozu exzessive Hingabe und Kenntnisse erforderlich waren, als Beweis dieser Schwierigkeiten.

[110] Dieser Punkt verdeutlicht, daß die von Webbrowsern interpretierten Skriptsprachen eigentlich bestimmten Problemen vorbehalten sind, vorrangig der Entwicklung reichhaltigerer und interaktiverer graphischer Benutzeroberflächen. Eine Skriptsprache kann dennoch in der Lage sein, 80 Prozent der Probleme in der clientseitige Programmierung zu lösen. Ihre Probleme haben gute Aussichten, komplett unter diese 80 Prozent zu fallen. Da Skriptsprachen außerdem eine einfachere und schnellere Entwicklung ermöglichen können, sollten Sie unter Umständen eine Skriptsprache in Betracht ziehen, bevor Sie sich einer anspruchsvolleren Lösung wie der Java-Programmierung zuwenden.

2.13.2.3 Java

[111] Wenn eine webbrowserinterpretierte Skriptsprache 80 Prozent der Probleme in der clientseitige Programmierung lösen kann, wie steht es dann mit den übrigen 20 Prozent, dem harten Kern? Java ist eine beliebige Lösung. Java ist nicht nur eine leistungsfähige, sichere, plattformübergreifende und internationale Programmiersprache, sondern wird auch kontinuierlich um Eigenschaften, Fähigkeiten und Bibliotheken erweitert, mit denen Probleme elegant gelöst werden können, deren Behandlung in anderen Programmiersprachen schwierig ist, zum Beispiel Threadprogrammierung, Datenbankzugriff, Netzwerk- und verteilte Programmierung. Java unterstützt clientseitige Programmierung durch *Applets* und *Java Web Start*.

[112] Ein Applet ist ein kleines Programm, das nur in einem Webbrowser ablaufen kann. Ein Applet wird als Teil einer HTML-Seite automatisch heruntergeladen (analog wie zum Beispiel eine Graphikdatei automatisch heruntergeladen wird). Nach seiner Aktivierung führt das Applet ein Programm aus. Dies ist ein Teil der Schönheit von Applets: Ein Applet bietet eine Möglichkeit, um clientseitige Software automatisch vom Server aus zu verteilen, wenn der Benutzer die Software benötigt und nicht früher. Der Benutzer bekommt die letzte Version der Software ohne Ausfall und ohne schwierige Neuinstallation. Aufgrund des Designs von Java, muß der Programmierer nur ein einziges Programm schreiben, welches automatisch auf jedem Rechner funktioniert, der über einen Webbrowser mit eingebautem Java-Interpreter verfügt. (Dies schließt mit Sicherheit die meisten Rechner ein.) Da Java eine voll ausgestattete Programmiersprache ist, können Sie soviel Funktionalität wie möglich clientseitig erledigen, bevor Anfragen an den Server gesendet werden. Sie müssen beispielsweise ein Anfrageformular nicht über das Internet versenden, um festzustellen, daß Sie ein Datum übersehen oder einen Parameter falsch bewertet haben. Der clientseitige Rechner kann Daten schnell visualisieren, statt darauf zu warten, daß der Server die Visualisierung generiert und als Graphikdatei zurückschickt. Sie erhalten nicht nur den unmittelbaren Gewinn an Geschwindigkeit und Reaktionsfähigkeit, sondern reduzieren auch die Auslastung von Netzwerk und Server, wodurch wiederum der Verlangsamung des gesamten Internets vorgebeugt wird.

2.13.2.4 Alternativen zu Applets

[113] Java-Applets sind, um ehrlich zu sein, ihren Vorschußlorbeeren nicht gerecht geworden. Als Java erschien, galt die größte Begeisterung den Applets, die endlich ernsthafte clientseitige Programmierung ermöglichen würden, also verbesserte Reaktionsfähigkeit gestatten und geringere Anforderungen an die Bandbreite internetbasierter Anwendungen stellen würden. Man malte sich unglaubliche Möglichkeiten aus.

[114] Es gibt tatsächlich einige ziemlich raffinierte Applets im Internet. Der überwältigende Durchbruch der Applets blieb allerdings aus. Das größte Problem war wohl, daß das Herunterladen von 10MB, um die Java-Laufzeitumgebung zu installieren, für den durchschnittlichen Benutzer zu erschreckend war. Die Tatsache, daß Microsoft die Laufzeitumgebung nicht in den Internet Explorer integrierte, mag das Schicksal der Applets besiegelt haben. Applets haben sich jedenfalls nicht in größerem Maßstab durchgesetzt.

[115] Dennoch sind Applets und Anwendung auf der Basis von Java Web Start in manchen Situationen nach wie vor wertvoll. Wann immer Sie Kontrolle über die Rechner der Benutzer haben, zum Beispiel innerhalb eines Unternehmens, ist es vernünftig, Anwendungsclients über diese beiden Technologien zu verteilen und zu aktualisieren. Sie sparen unter Umständen viel Zeit, Aufwand und Geld, vor allem bei häufigen Aktualisierungen.

[116] In Abschnitt 23.13 betrachten wir Macromedia Flex, eine vielversprechende neue Technologie, welche die Entwicklung Flash-basierter Applet-Äquivalente gestattet. Da der Flash Player bei mehr

als 98 Prozent aller Webbrowser installiert ist (Windows, Linux und Mac zusammengenommen), kann die Technologie als akzeptierter Standard angesehen werden. Das Installieren beziehungsweise Aktualisieren des Flash Players ist schnell und einfach. Die Sprache ActionScript basiert auf ECMA-Script und ist somit einigermaßen vertraut, aber Flex gestattet Ihnen, zu Programmieren, ohne sich um spezifische Eigenschaften des Browsers kümmern zu müssen und ist somit viel attraktiver als JavaScript. Im Hinblick auf die clientseitige Programmierung ist Flex eine Alternative, die durchaus in Betracht gezogen werden sollte.

2.13.2.5 .NET und C#

[117] Eine Zeit lang war ActiveX von Microsoft der wichtigste Konkurrent für Java-Applets, obwohl der Clientrechner an Windows gebunden war. Seither hat Microsoft mit der .NET-Plattform und der Sprache C# einen vollwertigen Mitbewerber neben Java geschaffen. Die .NET-Plattform entspricht in etwa der Java-Laufzeitumgebung (Softwareplattform, auf der Java-Programme ausgeführt werden) ~~mit der Standardbibliothek von Java~~ und C# hat unverkennbare Ähnlichkeit mit Java. Dies ist mit Sicherheit die beste Leistung, die Microsoft auf dem Gebiet der Programmiersprachen und -umgebungen vollbracht hat. Microsoft hatte natürlich den nicht unbeträchtlichen Vorteil, studieren zu können, was bei Java gut funktionierte und was nicht und auf diesen Erfahrungen aufzubauen, was aber die Leistung nicht schmälern soll. Java hatte erstmals seit seiner Schöpfung einen echten Gegner. Dementsprechend haben die Java-Designer bei Sun Microsystems C# unter die Lupe genommen, untersucht, was Programmierer zum Wechsel nach C# veranlaßt haben könnte und mit einigen fundamentalen Verbesserungen in der SE 5 reagiert.

[118] Gegenwärtig besteht die Hauptschwachstelle und wichtigste Frage bei .NET darin, ob Microsoft der *vollständigen* Portierung auf andere Plattformen zustimmen wird. Microsoft behauptet, daß in dieser Hinsicht keinerlei Probleme bestehen. Das Mono-Projekt (*http://www.mono-project.com*) hat bereits eine teilweise Implementierung der .NET-Plattform für Linux, aber vor der Fertigstellung dieser Implementierung und bevor sich Microsoft nicht entschieden hat, keinen Teil davon zu unterdrücken, ist eine plattformübergreifende Lösung auf der Basis von .NET eine riskante Wette.

2.13.2.6 Internet und Intranet im Vergleich

[119] Das Internet ist die allgemeinste Lösung des Client/Server-Problems. Es ist somit sinnvoll, denselben Ansatz zu wählen, um eine Teilmenge des Problems zu lösen, genauer das klassische Client/Server-Problem innerhalb eines Unternehmens. Beim traditionellen Client/Server-System besteht einerseits das Problem unterschiedlicher Typen von Clientrechnern und andererseits das Problem der Installation neuer Clientsoftware, die sich beide mit Hilfe von Webbrowsern und clientseitiger Programmierung gewandt lösen lassen. Wird die Internettechnologie auf ein Informationsnetzwerk innerhalb eines Unternehmens eingeschränkt, so spricht man von einem *Intranet*. Intranets bieten erheblich mehr Sicherheit als das Internet, da Sie den Zugriff auf die Server innerhalb des Unternehmens physikalisch kontrollieren können. Das Verständnis des allgemeinen Webbrowserkonzeptes scheint den Umgang mit unterschiedlichen Seiten und Applets erheblich zu erleichtern, so daß der Aufwand an Benutzerschulungen für neue Anwendungen geringer wird.

[120] Das Sicherheitsproblem führt zu einer der Abgrenzungslinien, die die clientseitigen Programmierung automatisch zu definieren scheinen. Wenn Ihr Programm „im Internet“ läuft, wissen Sie nicht, welche Plattform zugrunde liegt und sollten daher besonders sorgfältig darauf achten, kein fehlerhaftes Programm zu verbreiten. Sie brauchen etwas plattformübergreifendes und sicheres wie eine Skriptsprache oder Java.

[121] Läuft Ihr Programm dagegen im Intranet, so können sich andere Einschränkungen ergeben. Nicht selten haben sämtliche Rechner Intel/Windows-Plattformen. In einem Intranet sind Sie für die Qualität Ihres Programms verantwortlich und können entdeckte Fehler korrigieren. Eventuell existiert bereits eine Grundsubstanz aus älterem Quelltext, die Sie in einem eher traditionellen Client/Server-Umfeld verwenden, wobei Sie bei jedem Upgrade Clientsoftware installieren müssen. Der Zeitaufwand für das Installieren von Upgrades ist einer der gewichtigsten Gründe, um zur Verwendung von Webbrowsern überzugehen, da Upgrades hier unsichtbar und automatisch ablaufen (auch Java Web Start ist eine Lösung für dieses Problem). Wenn Sie bereits an einem solche Intranet beteiligt sind, ist es das Vernünftigste, den kürzesten Weg zu wählen, um den bereits vorhandenen Quelltext zu verwenden, statt das gesamte Programm in einer anderen Sprache neuzuschreiben.

[122] Angesichts der verblüffenden Vielfalt von Lösungen des Client/Server-Problems ist die Untersuchung der Kosten und Nutzen die beste Entscheidungstaktik. Betrachten Sie die Einschränkungen Ihres Problems und welcher der kürzeste Weg zur Lösung ist. Da auch die clientseitige Programmierung nach wie vor Programmierarbeit bedeutet, ist es stets sinnvoll, den schnellesten Entwicklungsansatz für Ihre Situation zu wählen. Dies ist eine aggressive Haltung, um auf die unvermeidlichen Zusammenstöße mit den Problemen der Softwareentwicklung vorbereitet zu sein.

2.13.3 Serverseitige Programmierung

[123] In der bisherigen Diskussion wurde die serverseitige Programmierung, also der Bereich, in dem Java seine größten Erfolge verzeichnet, nicht beachtet. Was geschieht, wenn Sie eine Anfrage an einen Server senden? Die meisten Anfragen fordern einfach eine Datei an. Ihr Webbrowser interpretiert die Datei in einer passenden Art und Weise: Als HTML-Seite, Graphikdatei, Java-Applet, Programm in einer Skriptsprache und so weiter.

[124] Eine etwas kompliziertere Anfrage an einen Server beinhaltet im allgemeinen eine Datenbanktransaktion. Ein häufiger Fall ist eine Anfrage mit komplexer Suche in einer Datenbank, deren Ergebnis der Server als HTML-Seite formatiert und an Sie zurücksendet. (Hat der Client durch Java oder eine Skriptsprache mehr „Intelligenz“, so können natürlich die Rohdaten gesendet und clientseitig formatiert werden. Das ist einerseits schneller und belastet andererseits den Server weniger.) Ein anderer häufiger Fall ist die Registrierung Ihres Namens in einer Datenbank, wenn Sie einer Gruppe beitreten oder einen Auftrag hinterlegen und beinhaltet Änderungen in dieser Datenbank. Diese Datenbankanfragen müssen mit Hilfe von Anweisungen auf der Serverseite bewerkstelligt werden. Derartige Funktionalität fällt unter die *serverseitige Programmierung*. Traditionellerweise wird serverseitige Programmierung in den Sprachen Perl, Python, C++ oder einer anderen für CGI-Programme geeigneten Sprache betrieben, wobei mittlerweile auch anspruchsvollere Systeme verfügbar sind. Darunter fallen Java-basierte Webserver, die serverseitige Programmierung mittels sogenannter *Servlets* gestattet. Servlets und ihre Abkömmlinge, die JSP-Seiten (JavaServer Pages), sind zwei der stärksten Gründe dafür, daß Unternehmen, die Webauftritte entwickeln, zu Java hin wechseln, nicht zuletzt auch deshalb, weil JSP-Seiten die Probleme durch unterschiedlich ausgestattete Webbrowser eliminieren. Die serverseitige Programmierung wird in *Thinking in Enterprise Java* ausführlich behandelt.

[125] Trotz der ausführlichen Diskussion über Java und das Internet in diesem Abschnitt, ist Java eine universelle Programmiersprache, mit der Sie ebenfalls alle Probleme lösen können, die Sie auch mit anderen Programmiersprachen lösen können. Die Stärke von Java ist hier nicht nur die Portabilität, sondern auch die Programmierbarkeit, Robustheit, die umfangreiche Standardbibliothek und die zahlreichen Bibliotheken von Drittanbietern, die verfügbar sind und kontinuierlich weiterentwickelt werden.

2.14 Zusammenfassung

[126] Sie wissen, daß ein prozedurales Programm aus Variablendeklarationen und Funktionsaufrufen besteht. Um die Funktionsweise eines solchen Programms zu verstehen, müssen Sie es durcharbeiten, die Funktionsaufrufe und systemnahen Konzepte durchsehen und ein gedankliches Modell konstruieren. Aus diesem Grund brauchen wir beim Entwickeln eines prozeduralen Programms „Zwischendarstellungen“, die wiederum selbst undurchsichtig sind, da sich die Ausdrucksweise mehr nach dem Computer als nach dem zu lösenden Problem richtet.

[127] Nachdem die objektorientierte Programmierung soviele Konzepte auf Dingen aufbaut, die in der prozeduralen Programmierung vorkommen, könnten Sie in natürlicher Weise annehmen, daß das resultierende Java-Programm noch erheblich komplizierter sein muß, als das äquivalente prozedurale Programm. Hier werden Sie angenehm überrascht sein: Ein gut geschriebenes Java-Programm ist in der Regel viel einfacher und leichter verständlich, als ein prozedurales Programm. Sie sehen die Definitionen der Klassen beziehungsweise Objekte, die Konzepte im Problemraum darstellen (im Gegensatz zu den issues of the computer representation) sowie die Benachrichtigungen an diese Objekte, die Aktivitäten im Problemraum repräsentieren. Eine der Freuden an der objektorientierten Programmierung besteht darin, daß sich der Quelltext eines gut geschriebenen Programms durch einfaches Lesen verstehen läßt. Außerdem ist der Quelltext insgesamt kürzer, da sich viele Probleme durch Wiederverwendung existierender Bibliotheken lösen lassen.

[128] Die objektorientierte Programmierung und Java mögen nicht für jedermann geeignet sein. Es ist wichtig, die eigenen Anforderungen zu untersuchen und zu entscheiden, in wieweit sich Java zu ihrer Implementierung eignet oder ob es nicht besser ist, ein anderes System zu nutzen (inklusive des Systems, das Sie zur Zeit verwenden). Wenn Sie wissen, daß Ihre Bedürfnisse in der absehbaren Zukunft sehr anspruchsvoll sind und spezifische Einschränkungen bestehen, die sich mit Java eventuell nicht einhalten lassen, sind Sie es sich selbst schuldig, nach Alternativen zu suchen (ich empfehle besonders Python, siehe *http://www.python.com*). Wenn Sie sich dennoch für Java als Ihre Programmiersprache entscheiden, werden Sie wenigstens verstehen, welche Optionen Ihnen zur Verfügung stehen und eine klare Vorstellung davon haben, warum Sie diese Richtung eingeschlagen haben.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 3

Alles ist Objekt

Inhaltsübersicht

[0] „If we spoke a different language, we would perceive a somewhat different world.“ Ludwig Wittgenstein (1889-1951)

Übersetzt etwa: „Sprächen wir eine andere Sprache, wir würden die Welt anders wahrnehmen.“

Java basiert zwar auf C++, ist aber eher eine „rein“ objektorientierte Sprache.

[1] C++ und Java sind beides hybride Sprachen, aber die Designer haben die Hybridisierung bei Java, verglichen mit C++, als weniger wichtig eingestuft. Ein hybride Sprache gestattet die Programmierung bezüglich mehrerer Paradigmen. C++ ist hybrid, um Rückwärtskompatibilität mit C zu unterstützen. Als Obermenge von C beinhaltet C++ viele unerwünschte Eigenschaften und Fähigkeiten, wodurch manche Aspekte von C++ übermäßig kompliziert sein können.

[2] Java setzt voraus, daß Sie ausschließlich objektorientiert programmieren wollen. Bevor Sie anfangen, müssen Sie Ihre Denkweise in Richtung Objektorientierung verschieben, sofern Sie diese Wandlung nicht bereits vollzogen haben. Der Nutzen dieser anfänglichen Bemühung besteht darin, anschließend in einer Sprache programmieren zu können, die sich einfacher erlernen läßt, als die meisten anderen objektorientierten Programmiersprachen.

3.1 Bedienung von Objekten über Referenzen

[3] Jede Programmiersprache hat ihren eigenen Zugriffsmechanismus für die Elemente im Arbeitsspeicher. Bei manchen Sprachen muß der Programmier stets auf die Wahl des richtigen Mechanismus' achten, das heißt darauf ob direkter Zugriff auf das Element möglich ist oder ob der Zugriff über einen indirekten Mechanismus, wie einen Zeiger bei C oder C++ vor sich geht, der eine spezielle Syntax erfordert.

Java verwendet ein vereinfachtes Zugriffsverfahren: Sie behandeln alles wie ein Objekt, in dem Sie eine einzige konsistente Syntax verwenden. Obwohl Sie alles wie ein Objekt *behandeln*, ist der Bezeichner, den Sie bedienen, in Wirklichkeit eine „Referenz“ auf ein Objekt.[1] Bildlich gesprochen, entspricht die Referenz der Fernsteuerung des Fernsehgerätes. Das Einstellen eines anderen Senders oder der Lautstärke wird über die Fernsteuerung (Referenz) vorgenommen, die wiederum den Fernseher (Objekt) bedient. Wenn Sie sich auf einen anderen Stuhl setzen und weiterhin den Fernseher bedienen wollen, nehmen Sie die Fernsteuerung mit, aber nicht den Fernseher.

[4] Die Fernsteuerung kann auch ohne Fernsehgerät vorhanden sein, das heißt die bloße Existenz einer Fernsteuerung (Referenz) bedeutet nicht, daß sie zu einem Fernsehgerät (Objekt) gehört. Wenn Sie ein Wort oder einen Satz in einem Programm hinterlegen möchten, deklarieren Sie eine *Referenzvariable* vom Typ `String`:

```
String s;
```

Sie haben bis jetzt ein Feld oder eine lokale Variable *zur Aufnahme einer Referenz* auf ein Objekt angelegt, aber noch kein Objekt erzeugt. Wenn Sie über die Referenzvariable `s` eine Benachrichtung

[1] An dieser Frage scheiden sich die Geister. Manche Programmierer betrachten Referenzen als Zeiger, obwohl der Zeigermechanismus eine entsprechende Implementierung voraussetzt. Andererseits sind die Referenzen von Java den Referenzen von C++ syntaktisch viel ähnlicher als Zeigern. In der ersten Auflage dieses Buches habe ich mich entschieden, den neuen Begriff „Handle“ zu erfinden, da zwischen den Referenzkonzepten von Java und C++ einige wesentliche Unterschiede bestehen. Ich kam selbst von C++ und wollte die C++-Programmierer nicht verwirren, die ich für die größte Zielgruppe für Java hielt. In der zweiten Auflage habe ich mich dazu durchgerungen, daß „Referenz“ der häufiger verwendete Begriff ist und Programmierer, die von C++ auf Java umsteigen, mit viel mehr Dingen zurechtkommen müssen, als nur mit der Bezeichnung für das Referenzkonzept. Wenn schon, denn schon. Es gibt aber auch Programmierer, die mit dem Begriff „Referenz“ nicht einverstanden sind. So las ich beispielsweise in einem Buch, es sei völlig falsch, zu sagen, daß Java die Übergabe von Referenzen unterstütze, da die Objektbezeichner bei Java (nach diesem Autor) *eigentlich* „Objektreferenzen“ seien. Alles, so dieser Autor, werde in Wirklichkeit als Wert übergeben. Es finde also keine Übergabe von Referenzen statt, sondern Übergabe von Objektreferenzen in Form von Werten. Man kann über die Genauigkeit derart gewundener Erklärungen streiten. Ich bin aber der Meinung, daß mein Ansatz das Verständnis des Konzeptes erleichert, ohne jemanden zu verletzen. (Die Sprachadvokaten behaupten vielleicht, daß ich Sie anlüge. Ich erwidere, daß ich eine angemessene Abstraktion gewählt habe.)

versenden, bekommen Sie eine Fehlermeldung, da `s` noch auf kein Objekt verweist. Es ist sicherer, eine Referenzvariable an der Stelle ihrer Deklaration sofort zu initialisieren:

```
String s = ''asdf'';
```

Sie sehen hier eine besondere Eigenschaft von Java: Eine Referenzvariable vom Typ `String` kann mit eine Zeichenkette zwischen Anführungszeichen initialisiert werden. Normally, you must use a more general type of initialization for objects.

3.2 Initialisierung von Referenzvariablen

[5] Sie haben eine Referenzvariable angelegt und wollen sie nun mit einem Objekt verknüpfen. Dies geschieht in der Regel mit Hilfe des `new`-Operators. Der `new`-Operator erzeugt ein Objekt des ihm übergebenen Typs. Beispielsweise erzeugt

```
String s = new String(''asdf'');
```

aber nicht nur ein neues `String`-Objekt, sondern übergibt mittels einer initialen Zeichenkette auch eine Information darüber, *wie* das Objekt erzeugt werden soll.

[6] Java wird natürlich mit einer Vielzahl vordefinierter Klassen ausgeliefert. Wichtiger ist aber, daß Sie selbst Ihre eigenen Klassen schreiben können. Das Entwickeln eigener Klasse ist in Wirklichkeit ein grundlegender Vorgang in der Java-Programmierung und der Gegenstand, mit dem wir uns im Rest dieses Buches auseinandersetzen werden.

3.2.1 Die drei Bereiche des Arbeitsspeichers

[7] Es ist sinnvoll, sich ein Bild von gewissen Aspekten zur Laufzeit eines Programms zu machen, vor allem von der Aufteilung und Funktionsweise des Arbeitsspeichers. Daten werden an fünf verschiedenen Orten gespeichert, davon drei im eigentlichen Arbeitsspeicher:

- **Register**: Dies ist der schnellste Speicherplatz und befindet sich an einer anderen Stelle als der übrige Arbeitsspeicher, nämlich im Prozessor. Die Anzahl der Register ist allerdings sehr begrenzt, so daß sie nur bei Bedarf allokiert werden. Sie haben keine direkte Kontrolle über die Register und Ihre Programme liefern keinerlei Hinweis dahingehend, daß sie überhaupt existieren. (C und C++ gestatten Ihnen dagegen, den Compiler zum Allokieren von Registern anzuregen.)
- **Stapelspeicher** („der Stack“): Dieser Speicherbereich liegt im allgemeinen Arbeitsspeicher, wird aber über den Stapelzeiger direkt vom Prozessor unterstützt. Der Stapelzeiger wird nach unten bewegt, um Speicherplatz zu allokieren und nach oben bewegt, um beanspruchten Speicherplatz wieder freizugeben. Der Stapelspeicher gestattet das Allokieren von Arbeitsspeicher auf extrem schnelle und effiziente Weise. Nur Register sind schneller. Der Laufzeitumgebung muß beim Anlegen des aufgerufenen Programms die Lebensdauer jedes Elementes auf dem Stapelspeicher bekannt sein. Diese Einschränkung begrenzt die Flexibilität Ihrer Programme. Obwohl die Laufzeitumgebung Arbeitsspeicher für einen Stapelspeicher beansprucht, insbesondere für *Referenzen* auf Objekte, liegen die Objekte selbst *nicht* auf dem Stapelspeicher.
- **Dynamischer Speicher** (*Freispeicher*, *Heap*): Dieser Speicherbereich ist ein universeller Vorrat an Arbeitsspeicher, liegt ebenfalls im allgemeinen Arbeitsspeicher und ist der „natürliche Lebensraum“ aller Java-Objekte. Der dynamische Speicher hat im Gegensatz zum Stapelspeicher den Vorteil, daß der Compiler nicht „wissen“ muß, wie lange der beanspruchte Speicherplatz benötigt wird. Wann immer Sie ein Objekt brauchen, legen Sie eine entsprechende

Anweisung an, um das Objekt per `new` zu erzeugen und der erforderliche Speicherplatz wird beim Verarbeiten dieser Anweisung im dynamischen Speicher allokiert. Diese Flexibilität hat natürlich einen Preis: Das Allokieren beziehungsweise Freigeben von Speicherplatz im dynamischen Speicher, beansprucht in der Regel mehr Zeit als auf dem Stapelspeicher (angenommen, es wäre bei Java, wie bei C++ möglich, Objekte auf dem Stapelspeicher zu erzeugen).

- **Statischer Speicher** (*constant storage*): Konstante Werte werden häufig direkt in den Quelltext eingesetzt. Dies ist sicher, da sich diese Werte nicht ändern können. Gelegentlich sind Konstanten cordoned off by themselves, so daß sie bei eingebetteten Systemen optional im nur-lesbaren Speicher (ROM) deponiert werden können.[2]

- *Dauerhafte Speicherung außerhalb des Arbeitspeichers*: Daten, die vollkommen außerhalb eines Programms (autark) „lebensfähig“ sind, können über die Laufzeit des Programms hinaus und außerhalb von dessen Kontrolle existieren. Die beiden wichtigsten Beispiele hierfür sind serialisierte Objekte, das heißt in einen Bytestrom umgewandelte Objekte, die in der Regel an einen anderen Rechner versendet werden und *persistente Objekte*, das heißt auf einer Festplatte gespeicherte Objekte, die ihren Zustand über das Programmende hinaus behalten. Der Trick besteht bei dieser Art von Speicherung darin, das Objekt in eine Form umzuwandeln, die auf einem anderen Medium existieren und bei Bedarf als gewöhnliches Objekt im dynamischen Speicher „wiederbelebt“ werden kann. Java unterstützt sowohl *leichtgewichtige Persistenz* als auch anspruchsvolle Mechanismen wie JDBC und Hibernate, um Objekte in Datenbanken ablegen beziehungsweise von dort zurückholen zu können.

3.2.2 Spezialfall: Die primitiven Typen liegen auf dem Aufrufstapel

[8] Eine Gruppe von Typen, die Sie recht häufig verwenden werden, erhält eine Sonderbehandlung: Die primitiven Typen. Der Grund für die Sonderbehandlung besteht darin, daß das Anlegen eines Objektes im dynamischen Speicher nicht sehr effizient vonstatten geht. Dies gilt insbesondere für Felder und lokale Variablen primitiven Typs. Java folgt bei den primitiven Typen dem Ansatz von C und C++ und erzeugt, statt den `new`-Operator zu bemühen, eine „automatische Variable“, die *keine* Referenzvariable ist. Ein Feld beziehungsweise eine lokale Variable primitiven Typs beinhaltet ihre Wert *direkt*, wird auf dem Aufrufstapel angelegt und ist somit erheblich effizienter.

[9] Java definiert die Größe jedes primitiven Typs. Diese Größen sind, im Gegensatz zu den meisten anderen Sprachen, nicht von der Rechnerarchitektur abhängig. Die Größeninvarianz der primitiven Typen ist ein Grund, warum Java-Programme portabler sind, als Programme in den meisten anderen Sprachen. Tabelle 3.1 faßt die primitiven Typen, ihre Größen und Wertebereiche zusammen. Alle numerischen Typen sind vorzeichenbehaftet.

[10] Die Größe des Typs `boolean` ist nicht explizit festgelegt. Ein Feld oder eine lokale Variable dieses Typs muß definitionsgemäß in der Lage sein, die literalen Werte `true` und `false` aufzunehmen.

[11] Die Wrapperklasse eines primitiven Typs gestattet Ihnen, ein Objekt im dynamischen Speicher anzulegen, welches den jeweiligen Wert primitiven Typs repräsentiert, zum Beispiel:

```
char c = 'x';
Character ch = new Character(c);
```

Äquivalent ist:

```
Character ch = new Character('x');
```

[2] Ein Beispiel hierfür ist der „`String`-Pool“: Alle literalen Zeichenketten und zeichenkettenwertigen konstanten Ausdrücke werden automatisch isoliert und im statischen Speicher angelegt.

Primitiver Typ	Länge in Bits	Minimum	Maximum	Wrapperklasse
`boolean`	—	—	—	`Boolean`
`char`	16 Bit	Unicode 0	Unicode $2^{16}-1$	`Character`
`byte`	8 Bit	-128	$+127$	`Byte`
`short`	16 Bit	-2^{15}	$+2^{15}-1$	`Short`
`int`	32 Bit	-2^{31}	$+2^{31}-1$	`Integer`
`long`	64 Bit	-2^{63}	$+2^{63}-1$	`Long`
`float`	32 Bit	IEEE754	IEEE754	`Float`
`double`	64 Bit	IEEE754	IEEE754	`Double`
`void`	—	—	—	`Void`

Tabelle 3.1: Die primitiven Typen von Java mit Größe, Wertebereich und der jeweiligen Wrapperklasse.

Das seit Version 5 der Java Standard Edition (SE 5) implementierte *Autoboxing* konvertiert zwischen primitivem Typ und Wrapperklasse hin:

```
Character ch = 'x';
```

und zurück:

```
char c = ch;
```

Die Gründe für das „Verpacken“ von Werten primitiven Typs werden in einem späteren Kapitel gezeigt.

3.2.2.1 Klassen für numerische Werte hoher Genauigkeit

[12] Die Standardbibliothek von Java beinhaltet zwei Klassen für Arithmetik mit hoher Genauigkeit: `java.math.BigInteger` und `java.math.BigDecimal`. Obwohl beide Klassen näherungsweise in die Kategorie der Wrapperklassen fallen, gibt es zu keiner von beiden einen äquivalenten primitiven Typ.

[13] `BigInteger` und `BigDecimal` definieren Methoden, die den Operationen auf Werten primitiven Typs entsprechen, das heißt Sie können mit Objekten vom Typ `BigInteger` beziehungsweise `BigDecimal` dasselbe tun, wie mit Werten vom Typ `int` oder `float`. Sie müssen lediglich die Methoden aufrufen, statt die Operatoren zu verwenden. Da die Rechenschritte bei `BigInteger`- und `BigDecimal`-Objekten aufwändiger sind, beanspruchen sie mehr Zeit als die Operationen auf Werten primitiven Typs. Sie tauschen Geschwindigkeit gegen Genauigkeit.

[14–16] Die Klasse `BigInteger` unterstützt ganze Zahlen bei belieber Genauigkeit. Sie können als ganzzahlige Werte beliebiger Größe darstellen, ohne bei Rechenoperationen Informationen zu verlieren. Die Klasse `BigDecimal` unterstützt Festkommawerte bei beliebiger Genauigkeit. Sie können Objekte dieser Klasse zum Beispiel für präzise monetary calculations verwenden. In der API-Dokumentation der beiden Klassen finden Sie die Einzelheiten über die verfügbaren Konstruktoren und Methoden.

3.2.3 Initialisierung und Bereichsprüfung von Arrays

[17] Beinahe jede Programmiersprache unterstützt eine Form von Arrays. Die Verwendung von Arrays ist bei C und C++ riskant, da sie dort nur Speicherbereiche sind. Greift ein Programm vor der Initialisierung eines solchen Speicherbereiches oder anschließend auf einen Index außerhalb

dieses Speicherbereiches zu (beides häufige Programmierfehler), so tritt ein unversehbares Ergebnis ein.

[18] Eines der wichtigsten Ziele von Java ist Sicherheit. Die Ursachen vieler Schwierigkeiten, mit denen sich die Programmierer bei C und C++ herumärgern, wurden daher bei Java nicht wiederholt. Bei Java ist ein Array garantiert initialisiert und Zugriffe außerhalb seines Indexbereichs sind nicht möglich. Die Prüfungen kosten zwar pro Array etwas mehr Speicherplatz und Laufzeit, man geht aber davon aus, daß die Sicherheit und erhöhte Produktivität die Unkosten aufwiegen (außerdem kann Java diese Operationen gelegentlich optimieren).

[19] Ein Array von Objekten ist eigentlich ein Array von Referenzvariablen, wobei jede Referenzvariable automatisch mit einem speziellen Wert initialisiert wird, der ein eigenes Schlüsselwort hat: `null`. Erkennt die Laufzeitumgebung den Wert `null`, so „weiß“ sie, daß die fragliche Referenzvariable nicht auf ein Objekt verweist. Sie müssen einer Referenzvariablen eine Referenz auf ein Objekt zuweisen, bevor Sie sie verwenden. Wenn Sie versuchen, eine Referenzvariable mit dem Inhalt `null` zu verwenden, wird zur Laufzeit ein Fehler gemeldet. Die typischen Fehler im Zusammenhang mit Arrays werden bei Java also verhindert.

[20–21] Sie können auch ein Array primitiven Typs anlegen. Auch hier garantiert der Compiler die Initialisierung, in dem jedes Element auf seinen typspezifischen Initialwert gesetzt wird (siehe Unterunterabschnitt 3.4.1.1). Das Thema „Arrays“ wird in den späteren Kapitel abgedeckt.

3.3 Objekte müssen nicht zerstört werden

[22] Bei den meisten Programmiersprachen nimmt das Konzept der Lebensdauer einer Variablen einen deutlichen Anteil des Programmieraufwandes ein. Wie lange ist eine Variable vorhanden? Sofern Sie eine Variable zerstören müssen, wann ist der richtige Zeitpunkt? Unklarheiten hinsichtlich der Lebensdauer von Variablen können zu vielen Fehlern führen. Dieser Abschnitt zeigt, wie Java diesen Aspekt erheblich vereinfacht, in dem sich die Laufzeitumgebung um sämtliche Aufräumarbeiten kümmert.

3.3.1 Geltungsbereich von Feldern und lokalen Variablen primitiven Typs

[23–25] Die meisten prozeduralen Sprachen kennen das Konzept des Geltungsbereichs. Der Geltungsbereich bestimmt sowohl die Lebensdauer als auch die Sichtbarkeit der in ihm deklarierten Felder beziehungsweise lokalen Variablen. Ein Geltungsbereich ist bei Java, C und C++ durch ein Paar geschweifter Klammern definiert, zum Beispiel:

```
{
    int x = 12;
    // Only x variable
    {
        int q = 96;
        // Both x & q available
    }
    // Only x available
    // q is ''out of scope''
}
```

Jedes Feld und jede lokale Variable ist einem Geltungsbereich zugeordnet und steht nur innerhalb dieses Geltungsbereichs zur Verfügung. Das Zeichen `//` leitet einen Kommentar ein, der bis zum Ende der Zeile reicht. Einrückungen erleichtern das Lesen des Quelltextes. Da Java eine formatfreie

Sprache ist, haben zusätzliche Leerzeichen, Tabulatoren und Wagenrückläufe keinen Einfluß auf das resultierende Programm. Folgendes ist bei Java nicht erlaubt (obwohl bei C und C++ zulässig):

```
{
    int x = 12;
    {
        int x = 96;
    }
}
```

Der Compiler meldet in diesem Fall, daß `x` bereits definiert ist. Die bei C und C++ mögliche *Überdeckung* eines beziehungsweise einer in einem übergeordneten Geltungsbereich deklarierten Feldes oder lokalen Variablen ist bei Java nicht erlaubt. Die Designer von Java waren der Auffassung, Überdeckung führe zu undurchsichtigen Programmen.

3.3.2 Geltungsbereich von Referenzvariablen

[26] Java-Objekte haben eine längere Lebensdauer als Felder und lokale Variablen primitiven Typs. Ein per `new`-Operator erzeugtes Objekt überdauert das Endes des Geltungsbereiches einer Referenzvariablen, die auf dieses Objekt verweist. Die Referenzvariable `s` in

```
{
    String s = new String(''a string'');
} // End of scope
```

verschwindet am Ende des Geltungsbereiches. Das von `s` referenzierte `String`-Objekt beansprucht aber nach wie vor Arbeitsspeicher. In diesem Fall gibt es nach dem Ende des Geltungsbereiches keine Möglichkeit mehr, das Objekt zu erreichen, da die einzige Referenzvariable, die auf das Objekt verweist, ihren Geltungsbereich verlassen hat. Sie lernen in den späteren Kapiteln, wie eine Referenz auf ein Objekt während des Programmablaufs weitergegeben und vervielfältigt werden kann.

[27] Es zeigt sich, daß eine Vielzahl von Programmierproblemen bei C++ bei Java schlicht und einfach nicht vorhanden sind, weil per `new`-Operator erzeugte Objekte solange existieren, als Sie wollen. Bei C++ müssen Sie sich nicht nur darum kümmern, daß die Objekte solange erhalten bleiben, wie sie benötigt werden, sondern diese auch nach Gebrauch zerstören.

[28] Das führt zu einer interessanten Frage: Wer oder was sorgt dafür, daß die Objekte nicht nach und nach den gesamten verfügbaren Arbeitsspeicher beanspruchen und das Programm zum Stillstand kommt, wenn die Laufzeitumgebung die Objekte „herumliegen“ läßt? Probleme genau dieser Art stellen sich bei C++ ein. Hier findet etwas Zauberei statt. Die Laufzeitumgebung verfügt über eine *automatische Speicherbereinigung*, die sämtliche, per `new`-Operator erzeugten Objekte beobachtet und ermittelt, auf welche davon keine Verweise mehr existieren. Die Speicherbereinigung gibt den von diesen Objekten beanspruchten Arbeitsspeicher frei, der anschließend für neue Objekte verwendet werden kann. Sie brauchen sich also niemals selbst um die Freigabe beanspruchten Arbeitsspeichers zu kümmern. Sie erzeugen einfach Objekte und wenn Sie sie nicht mehr brauchen, verschwinden sie von selbst. Die automatische Speicherbereinigung verhindert *Speicherlecks*, eine bestimmte Art von Problemen, wobei der Programmierer das Freigeben von Arbeitsspeicher vergißt oder übersieht.

3.4 Definition neuer Datentypen: Klassen

[29] Wodurch ist die Erscheinungsform und das Verhalten einer bestimmten Sorte von Objekten definiert, wenn alles Objekt ist? Was bestimmt den Typ eines Objektes? Eventuell erwarten Sie das

durchaus sinnvolle Schlüsselwort `type`. Historisch hat sich aber ergeben, daß die meisten objektorientierten Sprachen das Schlüsselwort `class` verwenden, um einen neuen Typ von Objekten zu beschreiben. Dem Schlüsselwort `class` (tritt so häufig auf, daß nicht jedes Vorkommen in diesem Buch in nichtproportionaler Schrift gesetzt ist) folgt der Name des neuen Typs, zum Beispiel:

```
class ATypeName { /* Class body goes here */ }
```

Diese Zeile führt einen neuen Typ ein. Der Klassenkörper besteht lediglich aus einem Kommentar (die Sternchen, Schrägstriche und was sich dazwischen befindet, wird später in diesem Kapitel diskutiert) und ist eigentlich zu nichts zu gebrauchen. Dennoch können Sie mit Hilfe des `new`-Operators ein Objekt dieser Klasse erzeugen:

```
ATypeName a new ATypeName();
```

Sie können allerdings nichts mit diesem Objekt anfangen, das heißt keine interessanten Benachrichtigungen an das Objekt versenden, bevor Sie einige Methoden definiert haben.

3.4.1 Felder und Methoden

[30] Wenn Sie eine Klasse definieren (und Sie tun bei Java nichts anderes, als Klassen zu definieren, Objekte dieser Klassen zu erzeugen und Benachrichtigungen an diese Objekte zu senden), können Sie Typen von *Komponenten* darin anlegen: Felder und Methoden. Ein Feld hat entweder *Referenztyp*, kann also Referenzen auf Objekte aufnehmen, über die Sie mit dem Objekt kommunizieren oder primitiven Typ. Hat ein Feld Referenztyp, so müssen Sie es, wie zuvor beschrieben, per `new`-Operator mit einer Referenz auf ein Objekt initialisieren.

[31] Jedes Objekt erhält eigenen Arbeitsspeicher für seine Felder. „Gewöhnliche“ Felder werden nicht von mehreren Objekten gemeinsam verwendet. Ein Beispiel für eine Klasse mit einigen Feldern:

```
class DataOnly {
    int i;
    double d;
    boolean b;
}
```

Diese Klasse hat, außer Daten zu enthalten, keine Funktionalität. Dennoch können Sie wiederum ein Objekt dieser Klasse erzeugen:

```
DataOnly data = new DataOnly();
```

Sie können den Feldern eines Objektes Werte zuweisen, wobei Sie zuerst wissen müssen, wie Sie sich auf eine Komponente eines Objektes beziehen. Dieser Bezug wird über den Namen der Referenzvariablen, gefolgt von einem Punkt (`.`) und dem Namen der Komponente des Objektes bewerkstelligt:

```
objectReference.member
```

Beispielsweise:

```
data.i = 47;
data.d = 1.1;
data.b = false;
```

Ein Objekt kann andere Objekte beinhalten, die wiederum Felder besitzen, auf die Sie zugreifen können. In diesem Fall wenden Sie die Verknüpfung über den Punktoperator einfach mehrmals an, zum Beispiel:

```
myPlane.leftTank.capacity = 100;
```

Die Klasse `DataOnly` kann nur Daten speichern, da sie keine Methoden hat. Das Verständnis von Methoden setzt voraus, daß Sie Argumente und Rückgabewerte verstehen, die im folgenden Unterunterabschnitt kurz beschrieben werden.

3.4.1.1 Typspezifische Initialwerte der primitiven Typen

[32] Deklariert eine Klasse ein Feld primitiven Typs, so wird dieses Feld garantiert mit einem Standardwert versehen, wenn Sie es nicht selbst initialisieren:

Primitiver Typ	Typspezifischer Initialwert
`boolean`	`false`
`char`	\u0000 (Null)
`byte`	(`byte`) 0
`short`	(`short`) 0
`int`	0
`long`	0L
`float`	0.0f
`double`	0.0d

Tabelle 3.2: Die primitiven Typen von Java mit ihren typspezifischen Initialwerten.

Die Initialisierung mit diesen Standardwerten ist nur für die Felder einer Klasse garantiert. Die garantierte Initialisierung der Felder primitiven Typs (C++ hat diese Eigenschaft nicht) reduziert eine Fehlerquelle. Der typspezifische Initialwert kann bezüglich Ihres Programms aber falsch oder sogar unzulässig sein. Es ist daher das Beste, Felder stets explizit zu initialisieren.

[33] Die garantierte Initialisierung erstreckt sich *nicht* auf lokale Variablen, die nicht zu den Feldern der Klasse gehören. Enthält die Definition einer Methode die Deklaration

```
int x;
```

so wird `x`, wie bei C und C++, mit irgendeinem Wert belegt, nicht aber automatisch mit 0 bewertet. Sie sind selbst dafür verantwortlich, `x` vor Verwendung einen passenden Wert zuzuweisen. Übersehen Sie eine nicht initialisierte lokale Variable, so verhält sich der Compiler bei Java definitiv sinnvoller als bei C++: Sie erhalten eine Fehlermeldung, die Ihnen mitteilt, daß die Variable eventuell nicht initialisiert wird. (Viele C++-Compiler warnen Sie bei uninitialisierten lokalen Variablen aber der Java-Compiler stuft ein solches Versäumnis als Fehler ein.)

3.5 Argumente und Rückgabewerte von Methoden

[34] In vielen Programmiersprachen, darunter C und C++, bezeichnet der Begriff *Funktion* eine benannte Subroutine. Bei Java ist dagegen die Bezeichnung *Methode* (im Sinne eines Verfahrens) gebräuchlich. Wenn Sie möchten, können Sie aber beim Begriff „Funktion“ bleiben, es ist eigentlich nur ein formaler Unterschied. Dieses Buch schließt sich aber der bei Java üblichen Bezeichnung „Methode“ an.

[35] Methoden definieren die Benachrichtungen, die an ein Objekt gesendet werden können. Die Bestandteile einer Methode sind Name, Argumentliste, Rückgabetyp und Methodenkörper. Das folgende Beispiel zeigt das Format einer Methodendefinition:

```
ReturnType methodName( /* Argument list */ ) {
    /* Method body */
}
```

Der Rückgabetyp beschreibt den von einer aufgerufenen Methode gelieferten Wert, die Argumentliste die Typen und Namen der Informationen, die Sie einer Methode beim Aufruf übergeben. Methodenname und Argumentliste werden zusammen als *Signatur* der Methode bezeichnet und identifizieren eine Methode eindeutig.

[36–37] Eine Methode kann bei Java nur als Komponente einer Klasse angelegt werden. Methoden können nur auf Objekten[3] aufgerufen werden, die wiederum in der Lage sein müssen den Methodenaufruf zu verarbeiten. Der Versuch, auf einem Objekt eine nicht definierte Methode aufruzufen, bewirkt eine Fehlermeldung zur Übersetzungszeit. Zum Aufrufen einer Methode auf einem Objekt notieren Sie nach der entsprechenden Referenzvariable einen Punkt und den Namen der Methode mit Argumentliste, zum Beispiel:

```
objectName.methodName(arg1, arg2, arg3);
```

Stellen Sie sich eine Methode `f()` vor, die keine Argumente erwartet und einen Wert vom Typ `int` zurückgibt. Verweist die Referenzvariable `a` auf ein Objekt, dessen Klasse die Methode `f()` definiert, so können Sie `f()` folgendermaßen aufrufen:

```
int x = a.f();
```

Der Rückgabetyp der Methode muß mit dem Typ von `x` kompatibel sein. Das Aufrufen einer Methode auf einem Objekt wird häufig als *Senden einer Benachrichtung an ein Objekt* bezeichnet. Im letzten Beispiel ist `f()` die Benachrichtigung, während `a` das Objekt darstellt. Objektorientierte Programmierung wird häufig vereinfacht als „Senden von Benachrichtigungen an Objekte“ zusammengefaßt.

3.5.1 Die Argumentliste einer Methode

[38] Die Argumentliste gibt an, welche Informationen beim Methodenaufruf übergeben werden können. Wie Sie vielleicht schon vermuten, haben diese Informationen, wie alles bei Java, die Form von Objekten. Die Argumentliste definiert, welche Typen von Objekten übergeben werden können und wie diese Objekte innerhalb der Methode bezeichnet werden. Wie überall in Java, übergeben Sie in Wirklichkeit Referenzen[4] auf Objekte, wenn Sie scheinbar mit Objekten hantieren. Die Referenzvariablen in der Argumentliste müssen korrekt typisiert sein. Erwartet eine Methode ein Argument vom Typ `String`, so müssen Sie eine Referenz auf ein `String`-Objekt übergeben. Andernfalls meldet der Compiler einen Fehler.

[39] Das folgende Beispiel zeigt die Definition einer Methode, die ein Argument vom Typ `String` erwartet und in eine Klassendefinition eingesetzt werden muß, um übersetzt werden zu können:

```
int storage(String s) {
    return s.length() * 2;
}
```

Die Methode `storage()` gibt an, wieviele Bytes erforderlich sind, um den Inhalt des übergebenen `String`-Objektes zu speichern. (Jeder Wert vom Typ `char` ist 16 Bit beziehungsweise zwei Byte groß, um Unicodezeichen zu unterstützen.) Das Argument der `storage()`-Methode heißt `s` und ist

[3] Statische Methoden können auf der Klasse aufgerufen werden, ohne daß ein Objekt erforderlich ist. Sie lernen die statischen Methoden später in diesem Kapitel kennen.

[4] Mit Ausnahme der zuvor erwähnten primitiven Typen `boolean`, `char`, `byte`, `short`, `int`, `long`, `float` und `double`. In der Regel übergeben Sie allerdings Referenzen auf Objekte.

vom Typ `String`. Im Methodenkörper kann `s` wie jede andere Referenzvariable verwendet werden, das heißt Sie können Benachrichtigungen an das referenzierte Objekt senden. Die hier aufgerufene Methode `length()` ist in der Klasse `String` definiert und gibt die Anzahl der Zeichen der von ihrem `String`-Objekt repräsentierten Zeichenkette zurück.

[40] Das obige Beispiel zeigt insbesondere die Verwendung des Schlüsselwortes `return`. Es hat zwei Aufgaben: Das Schlüsselwort `return` leitet erstens die Rückkehr aus der Methode ein. Zweitens wird, falls die Methode einen Wert liefert, dieser Rückgabewert rechts unmittelbar neben `return` angegeben. In diesem Fall folgt der Rückgabewert durch Auswertung des Ausdrucks `s.length()*2`.

[41] Sie können jeden beliebigen Rückgabetyp wählen. Liefert eine Methode aber explizit keinen Rückgabewert, so definieren Sie den speziellen Rückgabe `void`. Einige Beispiele:

```
boolean flag() { return true; }
double naturallyLogBase() { return 2.718; }
void nothing() { return; }
void nothing2() { }
```

Definiert eine Methode den Rückgabetyp `void`, so dient das Schlüsselwort `return` lediglich dazu, um die Methode zu verlassen und ist am Ende des Methodenkörpers nicht notwendig. Sie können den Methodenkörper an jeder beliebigen Stelle verlassen. Definiert eine Methode allerdings einen von `void` verschiedenen Rückgabetyp, so zwingt Sie der Compiler durch Fehlermeldungen, unabhängig von welcher Stelle Sie aus der Methode zurückkehren, einen Wert passenden Typs zurückzugeben.

[42] Bis jetzt sieht es so aus, als ob ein Programm aus einer Anzahl von Objekten mit Methoden besteht, die Objekte als Argument erwarten und sich gegenseitig Benachrichtungen senden. Dies ist in der Tat der größte Teil des Geschehens. Im übernächsten Kapitel lernen Sie, wie sich der Programmablauf in einer Methode verzweigen läßt. Für dieses Kapitel genügt es, Benachrichtungen senden zu können.

3.6 Sonstige Bestandteile eines Java-Programms

[43] Sie müssen noch einige weitere Dinge verstehen, bevor wir uns Ihrem ersten Java-Programm widmen können.

3.6.1 Sichtbarkeit von Namen

[44] Die Kontrolle über die verwendeten Namen (Bezeichner) ist ein Problem, dem sich jede Programmiersprache stellen muß. Angenommen, Sie verwenden einen Namen in einem Modul eines Programms und ein anderer Programmierer verwendet denselben Namen in einem anderen Modul desselben Programms. Wie können Sie beide Namen voneinander unterscheiden und Kollisionen verhindern? Dieses Problem tritt vor allem bei C auf, wo ein Programm häufig ein unüberschaubare Menge von Namen beinhaltet. Bei C++ werden Funktionen in Klassen geschachtelt, so daß sie nicht mit den Namen von Funktionen anderer Klassen kollidieren können (die Klassen von Java basieren auf dem Klassenkonzept von C++). Anderseits gestattet C++ nach wie vor globale Variablen und globale Funktionen, so daß Namenskollisionen möglich sind. Bei C++ wird dieses Problem durch das Konzept des *Namensraums* mit einem eigenen Schlüsselwort gelöst.

[45] Java konnte das Problem der Namenskollision durch einen neuen Ansatz vermeiden. Da die Domainnamen im Internet eindeutig sind, schlagen die Erfinder von Java vor, den Domainnamen „rückwärts“ als Anfangsstück zu verwenden, um einer Bibliothek einen eindeutigen Namen zu geben.

Mein Domainnamen lautet zum Beispiel *mindview.net*. Meine Hilfsbibliothek `foibles` würde demnach mit vollqualifiziertem Namen `net.mindview.utility.foibles` heißen. Nachdem umgekehrten Domainnamen, sollen Punkte verwendet werden, um Unterverzeichnisse auszudrücken.

[46] Bei Java 1.0/1.0 wurden die Domainendungen *com*, *edu*, *org*, *net* und so weiter, konventionsmäßig in Großbuchstaben wiedergegeben. Meine Biblithek würde also `NET.mindview.utility.foibles` heißen. Während der Entwicklung von Java 2 zeigten sich allerdings Probleme durch diese Konvention, so daß nun der gesamte sogenannte *Packagename* durchgängig in Kleinbuchstaben geschrieben wird.

[47] Der Packagemechanismus bewirkt, daß Ihre Dateien eigenen Namensräumen zugeordnet werden, wobei jede Klasse in einer Datei ein eindeutigen Namen haben muß. Die Sprache schließt auf diese Weise Namenskollisionen aus.

3.6.2 Importieren existierender Klassen

[48] Wenn Sie eine vordefinierte Klasse in Ihrem Programm verwenden wollen, muß der Compiler „wissen“, wo sich diese Klasse befindet. Die Klasse kann natürlich in derselben Quelltextdatei definiert sein, in der sie verwendet wird. In diesem Fall verwenden Sie die Klasse einfach, auch wenn die Klasse erst später in der Datei definiert wird (Java löst das Problem der „Vorwärtsdeklaration“).

[49] Und wenn die Klasse in einer anderen Datei definiert ist? Vielleicht denken Sie, daß der Compiler schlau genug sein sollte, um die Klasse zu finden, aber dabei übersehen Sie ein Problem. Stellen Sie vor, Sie brauchen eine bestimmte Klasse und es gibt mehr als eine Definition dieser Klasse (die sich voraussichtlich voneinander unterscheiden). Schlimmer noch, stellen Sie sich vor, daß Sie an einem Programm arbeiten und *as you're building it you add a new class to your library* die mit dem Namen einer bereits vorhandenen Klasse kollidiert.

[50] Sie müssen jede potentielle Mehrdeutigkeit ausschalten, um dieses Problem zu lösen. Dies erreichen Sie, indem Sie dem Compiler per `import`-Anweisung exakt mitteilen, welche Klasse Sie meinen. Die `import`-Anweisung weist den Compiler an, ein sogenanntes *Package* (eine Bibliothek) zu importieren. (Bei anderen Sprachen kann eine Bibliothek neben Klassen auch Funktionen und Konstanten enthalten. Denken Sie aber daran, daß bei Java aller Quelltext innerhalb einer Klasse stehen muß.)

[51] Sie werden in der Regel Klassen aus der Standardbibliothek von Java verwenden, die zusammen mit dem Compiler zum Java Development Kit (JDK) gehören. Bei diesen Klassen brauchen Sie sich nicht mit länglichen umgekehrten Domainnamen auseinanderzusetzen. Die Anweisung

```
import java.util.ArrayList;
```

teilt dem Compiler mit, daß Sie die Klasse `ArrayList` verwenden möchten. Das Package `java.util` beinhaltet außer `ArrayList` zahlreiche andere Klassen, von denen Sie eventuell einige brauchen, aber nicht jede explizit importieren wollen. Die folgende `import`-Anweisungen mit `*`-Platzhalter

```
import java.util.*;
```

importiert sämtliche Klassen im Package `java.util`. In der Regel importieren Sie mehrere Klassen über die `*`-Notation, als jede benötigte Klasse einzeln anzugeben.

3.6.3 Der static-Modifikator

[52] Wenn Sie eine Klasse anlegen, beschreiben Sie gewöhnlich, wie sich die Objekte dieser Klasse darstellen und verhalten. Es existiert kein Objekt, bevor Sie per `new`-Operator eines erzeugen. Erst

an dieser Stelle wird Arbeitsspeicher allokiert und die Methoden werden verfügbar.

[53] Es gibt zwei Situationen, in denen dies nicht ausreicht. Der eine Fall ist gegeben, wenn Sie, unabhängig von der Anzahl der Objekte einer Klasse, genau einen Speicherplatz für ein Feld beanspruchen wollen, selbst wenn überhaupt kein Objekt dieser Klasse existiert. Der andere Fall tritt ein, wenn Sie eine Methode brauchen, die nicht zu einem bestimmten Objekt der Klasse gehört, also eine Methode, die Sie auch aufrufen können, wenn noch kein Objekt dieser Klasse vorhanden ist.

[54] Sie können beide Effekte mit dem Schlüsselwort `static` bewerkstelligen. Eine als statisch deklarierte Komponente, das heißt ein statisches Feld oder eine statische Methode, gehört keinem Objekt der entsprechenden Klasse an. Sie können eine statische Methode stets aufrufen beziehungsweise ein statisches Feld stets abfragen oder ändern, auch wenn kein Objekt der Klasse existiert. Bei gewöhnlichen nicht statischen Feldern und Methode erfordert der Zugriff stets ein Objekt, da nicht statische Felder und Methoden „wissen“ müssen, zu welchem Objekt sie gehören.[5]

[55] Einige objektorientierte Sprachen bezeichnen statische Komponenten als *Klassenfelder* beziehungsweise *Klassenmethoden*, um auszudrücken, daß diese Felder und Methoden zur Klasse selbst gehören, nicht aber zu einem bestimmten Objekt der Klasse. Gelegentlich findet man diese Bezeichnungen aber auch in der Java-Literatur.

[56] Sie deklarieren ein Feld oder eine Methode als statisch, indem Sie der Deklaration beziehungsweise Definition den `static`-Modifikator voranstellen. Die folgende Klasse `StaticTest` deklariert und initialisiert ein statisches Feld:

```
class StaticTest {
    static int i = 47;
}
```

Auch wenn Sie zwei oder mehr `StaticTest`-Objekte erzeugen, existiert genau ein Speicherplatz für das Feld `StaticTest.i`:

```
StaticTest st1 = new StaticTest();
StaticTest st2 = new StaticTest();
```

An dieser Stelle beinhalten die beiden Felder `st1.i` und `st2.i` denselben Wert (47), da sie sich auf ein und dieselbe Speicherstelle beziehen.

[57] Es gibt zwei Möglichkeiten, um sich auf ein statisches Feld zu beziehen. Sie können, wie das vorige Beispiel zeigt, über eine Referenz auf ein Objekt auf das statische Feld zugreifen (`st1.i`) oder direkt über den Klassennamen (bei nicht statischen Feldern nicht möglich):

```
StaticTest.i++;
```

Der Operator `++` erhöht („inkrementiert“) den Feldinhalt um Eins. Nun enthalten `st1.i` und `st2.i` beide den Wert 48.

[58] Der Bezug über den Klassennamen ist die bevorzugte Schreibweise für den Zugriff auf ein statisches Feld. Die Schreibweise verdeutlicht nicht nur den statischen Charakter des Feldes, sondern gestattet dem Compiler unter Umständen auch Optimierungen.

[59] Dasselbe gilt für statische Methoden. Sie können eine statische Methode entweder über eine Referenz auf ein Objekt aufrufen, wie jede Methode oder über die Syntax `ClassName.method()`. Die Deklaration per `static`-Modifikator geschieht analog wie oben:

[5] Da das Aufrufen einer statischen Methode kein Objekt der entsprechenden Klasse voraussetzt, hat eine statische Methode natürlich keinen *unmittelbaren* Zugriff auf die nicht statischen Felder und Methoden ihrer Klasse, das heißt ohne Bezug auf ein benanntes (von einer Referenzvariablen referenzierte) Objekt. Nicht statische Felder und Methoden sind an ein bestimmtes Objekt gebunden.

```
class Incrementable {
    static void increment() { StaticTest.i++; }
}
```

Die statische Methode `increment()` der Klasse `Incrementable` inkrementiert das statische Feld `i` mit Hilfe des Operators `++`. Sie können die `increment()`-Methode über eine Referenz auf ein Objekt aufrufen:

```
Incrementable sf = new Incrementable();
sf.increment();
```

oder, als statische Methode auch direkt über die Klasse:

```
Incrementable.increment();
```

Der `static`-Modifikator wirkt sich bei Feldern zwar definitiv auf die Art und Weise aus, in der das Feld angelegt wird (ein Feld pro Klasse im Gegensatz zu einem Feld pro Objekt, bei nicht statischen Feldern), die Auswirkungen auf Methoden sind aber weniger dramatisch. Ein wichtiger Anwendungsfall für statische Methoden ist das Aufrufen einer Methode, ohne ein Objekt erzeugen zu müssen. Dieser Aspekt ist wichtig für die Definition der `main()`-Methode, die den Eintrittspunkt zur Ausführung eines Programms darstellt.

3.7 Das erste Java-Programm

[60] Hier ist nun endlich das erste vollständige Java-Programm. Es gibt eine Zeichenkette und das aktuelle Datum mit Uhrzeit aus. Das Programm verwendet dazu die Klasse `Date` aus der Standardbibliothek von Java:

```
//: object/HelloDate.java
import java.util.*;

public class HelloDate {
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
} /* Output: (55% match)
      Hello, it's:
      Wed Oct 05 14:39:36 MDT 2005
   *///:~
```

Zu Beginn jedes Programms müssen Sie die erforderlichen `import`-Anweisungen anlegen, um die im Quelltext zusätzlich benötigten Klassen zu importierten. Beachten Sie den Zusatz „zusätzlich“. Es gibt nämlich eine Bibliothek von Klassen, die automatisch in jedes Java-Programm importiert wird: `java.lang`. Sehen Sie sich die API-Dokumentation in Ihrem Webbrowser an. (Wenn Sie die API-Dokumentation des JDKs noch nicht von *http://java.sun.com* heruntergeladen haben, ist jetzt der richtige Zeitpunkt.[6] Die API-Dokumentation gehört nicht zu dem komprimierten Archiv, welches das JDK enthält, sondern muß separat heruntergeladen werden.) Die Packageliste unter dem Menüpunkt „Overview“ zeigt alle Klassen der Standardbibliothek von Java. Wählen Sie `java.lang` aus, um eine Liste aller Klassen zu erhalten, die zu dieser Bibliothek gehören. Die `java.lang`-Bibliothek enthält allerdings keine Klasse namens `Date`, so daß Sie eine weitere Bibliothek importieren müssen, um diese Klasse verwenden zu können. Wenn Sie nicht wissen, welcher Bibliothek eine bestimmte Klasse

[6] Der Java-Compiler und die API-Dokumentation von Sun Microsystems ändern sich regelmäßig. Die beste Bezugsquelle ist Sun Microsystems selbst. Indem Sie Compiler und API-Dokumentation selbst herunterladen, haben Sie stets die aktuelle Version zur Verfügung.

zugeordnet ist oder wenn Sie alle verfügbaren Klassen sehen wollen, können Sie den Menüpunkt „Tree“ in der API-Dokumentation wählen. In dieser Darstellung können mit der Suchfunktion Ihres Webbrowsers nach jeder einzelnen Klasse des JDKs suchen, zum Beispiel nach `Date`. Die gesuchte Klasse wird als `java.util.Date` aufgelistet, woraus Sie entnehmen können, daß sie zur Bibliothek `java.util` gehört. Sie müssen also `java.util.*` importieren, um die Klasse `Date` verwenden zu können.

[61] Kehren Sie nun wieder zur `java.lang`-Bibliothek zurück. Die Klasse `System` hat mehrere Felder. Das statische `out`-Feld verweist auf ein Objekt der Klasse `PrintStream`. Da das `out`-Feld statisch ist, brauchen Sie kein Objekt per `new`-Operator zu erzeugen. Das `PrintStream`-Objekt existiert bereits und steht zur Verwendung bereit. Die Verwendungsmöglichkeiten des von `out` referenzierten Objektes hängen von seinem Typ ab: `PrintStream`. Die API-Dokumentation der Klasse `System` enthält links neben dem `out`-Feld einen Verweis auf die Klasse `PrintStream`. Wenn Sie diesem Verweis folgen, gelangen Sie zur API-Dokumentation der Klasse `PrintStream`, die sämtliche verfügbaren Methoden angibt. Die Klasse `PrintStream` definiert eine ganze Reihe von Methoden, die in diesem Buch noch behandelt werden, unter anderem in Kapitel 19. Im Augenblick interessieren wir uns nur für die Methode `println()`, die ihr Argument, gefolgt von einem Zeilenumbruch, auf der Konsole ausgibt. Wenn eines Ihrer Programme eine Ausgabe über die Konsole liefern soll, können Sie also eine Anweisung wie die folgende wählen:

```
System.out.println(''A String of things'');
```

[62] Klassen- und Dateiname stimmen überein. Bei einem eigenständigen Programm wie dem obigen Beispiel muß eine Klasse in der Datei mit dem Namen der Datei bezeichnet werden. Andernfalls gibt der Compiler eine Fehlermeldung aus. Diese Klasse muß eine Methode namens `main()` mit der folgender Signatur und folgendem Rückgabetyp definieren:

```
public static void main(String[] args) {
```

Das Schlüsselwort `public` bedeutet, daß die Methode „öffentlich“, das heißt uneingeschränkt verfügbar ist und wird in Unterabschnitt 7.2.2 detailliert beschrieben. Der Parameter der `main()`-Methode (`args`) ist ein Array von Referenzvariablen vom Typ `String`. Das Array wird in diesem Programm zwar nicht verwendet, aber der Compiler besteht auf seiner Anwesenheit. Der Parameter `args` repräsentiert die auf der Kommandozeile übergebenen Argumente.

[63] Die Zeile, die das Datum und die Uhrzeit ausgibt, ist recht interessant:

```
System.out.println(new Date());
```

Das Argument der `println()`-Methode ist ein `Date`-Objekt und wird nur erzeugt, um seinen Inhalt (der automatisch in ein `String`-Objekt umgewandelt wird) an `println()` zu übergeben. Unmittelbar nach Verarbeitung der `println()`-Anweisung wird das `Date`-Objekt entbehrlich und kann jederzeit der automatische Speicherbereinigung übergeben werden. Wir brauchen uns nicht darum zu kümmern, es aufzuräumen.

[64] Ein Blick in die API-Dokumentation zeigt, daß die Klasse `System` noch viele andere Methoden mit interessanter Funktionalität definiert (einer der wichtigsten Aktivposten von Java ist die umfangreiche Standardbibliothek), zum Beispiel:

```
//: object/ShowProperties.java
public class ShowProperties {
    public static void main(String[] args) {
        System.getProperties().list(System.out);
        System.out.println(System.getProperty(''user.name''));
        System.out.println(System.getProperty(''java.library.path''));
    }
```

```
} ///:~
```

Die erste Zeile in der `main()`-Methode zeigt alle Eigenschaften der Systemumgebung (im Java-Jargon „Properties“) an, in der Sie das Programm aufrufen. Die `list()`-Methode sendet das Ergebnis an ihr Argument (`System.out`). Sie lernen später, daß Sie dieses Ergebnis überall hinsenden können, zum Beispiel auch an eine Datei. Die zweite und dritte Zeile zeigen, wie Sie eine bestimmte einzelne Eigenschaft abfragen können, hier den Benutzernamen und die Eigenschaft `java.library.path`. (Die ungewöhnlichen Kommentare zu Beginn und am Ende des Programms werden ~~später in diesem Buch~~ erklärt.

3.7.1 Übersetzen und Aufrufen des Programmes

[65] Sie brauchen eine Java-Programmierumgebung, um dieses und die übrigen Beispiele in diesem Buch übersetzen und ausführen zu können. Es gibt zwar einige Entwicklungsumgebungen von Drittanbietern, aber ich setze in diesem Buch voraus, daß Sie das originale Java Development Kit (JDK) von Sun Microsystems verwenden. Falls Sie sich für eine andere Entwicklungsumgebung[7] entschieden haben, müssen Sie in der dortigen Dokumentation nachlesen, wie Programme übersetzt und aufgerufen werden.

[66] Rufen Sie die Java-Website von Sun Microsystems auf (*http://java.sun.com*). Sie finden dort Informationen und Verweise zum Herunterladen und zur Installation des JDK für Ihre Plattform.

[67] Nachdem Sie das JDK installiert und die Pfadumgebungsvariable Ihres Betriebssystems aktualisiert haben, so daß Sie die Kommandos `javac` und `java` aufrufen können, laden Sie sich die Quelltextdistribution zu diesem Buch von *http://www.mindview.net* herunter und entpacken sie. Zu jedem Kapitel dieses Buches wird ein separates Unterverzeichnis angelegt. Wechseln Sie ins Unterverzeichnis *objects* und geben Sie das folgende Kommando ein:

```
javac HelloDate.java
```

Dieses Kommando sollte keine Meldungen bewirken. Wird dennoch eine Fehlermeldung ausgegeben, so haben Sie das JDK nicht korrekt installiert. Korrigieren Sie gegebenenfalls die Installation beziehungsweise Konfiguration Ihres JDKs.

[68] Kehrt die Eingabeaufforderung ohne Fehlermeldung zurück, geben Sie folgendes Kommando ein:

```
java HelloDate
```

Sie bekommen eine Ausgabe in Form der obigen Meldung mit Datum und Uhrzeit.

[69] Auf diese Weise läßt sich jedes Programm in diesem Buch übersetzen und aufrufen. Jedes Unterverzeichnis der Quelltextdistribution beinhaltet aber auch eine Datei namens *build.xml*, die Ant-Kommandos beinhaltet, um die Dateien des jeweiligen Kapitels auch automatisch übersetzen zu können. Das Format der *build.xml*-Datei und das Hilfsprogramm Ant werden im Anhang von *http://www.mindview.net/Books/BetterJava* ausführlicher beschrieben. Haben Sie Ant aber bereits installiert (*http://ant.apache.org*), so genügt das Kommando `ant` beziehungsweise `ant run`, um die Beispiele in jedem Kapitel zu übersetzen und aufzurufen. Wenn Sie Ant noch nicht installiert haben, können Sie die Kommandos `javac` und `java` auch einfach von Hand aufrufen.

[7] Der `jikes`-Compiler von IBM ist eine häufige Alternative, da er deutlich schneller als der `javac`-Compiler von Sun Microsystems arbeitet (wenn Sie viele Dateien per Ant übersetzen, schwindet der Unterschied allerdings). Darüber hinaus gibt es auch quelloffene Projekte, die Java-Compiler, -Laufzeitumgebungen und -Bibliotheken entwickeln.

3.8 Kommentare und eingebettete Dokumentation (Javadoc)

[70] Java kennt zwei Varianten von Kommentaren. Die erste Variante ist der traditionelle C-Kommentar, der auch bei C++ verwendet wird. Diese Kommentare beginnen mit dem Symbol `/*`, erstrecken sich unter Umständen über mehrere Zeilen und enden mit dem Symbol `*/`. Viele Programmierer beginnen jede neue Zeile innerhalb eines solchen Kommentars mit einem Sternchen (`*`). Sie werden häufig Kommentare der folgenden Form vorfinden:

```
/* This is a comment
  * that continues
  * across lines.
  */
```

Alle Zeichen zwischen `/*` und `*/` werden nicht beachtet. Der vorige Kommentar ist somit gleichbedeutend mit:

```
/* This is a comment that
continues across lines */
```

Die zweite Variante stammt von C++. Das Symbol `//` leitet einen einzeiligen Kommentar ein, der sich bis zum Zeilenende erstreckt. Diese Art von Kommentaren wird aufgrund ihrer Einfachheit häufig benutzt. Sie brauchen auf Ihrer Tastatur nicht nach `/` und `*` zu suchen (sondern drücken dieselbe Taste zweimal) und müssen den Kommentar auch nicht abschließen. Auch diese Kommentare werden Ihnen oft begegnen.

```
// This is a one-line comment
```

3.8.1 Dokumentationskommentare

[71] Die Pflege der Dokumentation ist wohl das größte Problem beim Dokumentieren des Quelltextes. Sind Dokumentation und Quelltext voneinander getrennt, so wird es zunehmend lästig, bei jeder Änderung am Quelltext die Dokumentation zu aktualisieren. Die Lösung scheint einfach zu sein: Verknüpfen Sie Quelltext und Dokumentation. Die einfachste Möglichkeit besteht darin, beides in ein und derselben Datei unterzubringen. Um das Bild abzurunden, brauchen Sie noch eine spezielle Kommentarsyntax zur Kennzeichnung der Dokumentation sowie ein Hilfsprogramm, das diese *Dokumentationskommentare* extrahiert und formatiert. Java geht diesen Weg.

[72] Das Extraktionsprogramm für die Dokumentationskommentare heißt `javadoc` und gehört zur JDK-Installation. Javadoc nutzt einen Teil der Technologie des Java-Compilers, um nach den speziellen Kommantarsymbolen in Ihrem Quelltext zu suchen. Javadoc extrahiert nicht nur die durch spezielle Kommentarsymbole gekennzeichnete Information, sondern auch den zum jeweiligen Dokumentationskommentar gehörigen Klassen- beziehungsweise Methodennamen. Auf diese Weise können Sie mit minimalen Arbeitsaufwand anständige Dokumentation generieren.

[73] Die Ausgabe von Javadoc ist eine HTML-Datei, die Sie in Ihren Webbrowser laden können. Javadoc gestattet also das Anlegen und Pflegen von Quelltext und Dokumentation in einer gemeinsamen Datei sowie das Generieren von Dokumentation. Da Javadoc einen einfachen Dokumentationsstandard definiert, können Sie bei jeder Java-Bibliothek Dokumentation erwarten, wenn nicht sogar einfordern.

[74] Darüber hinaus können Sie auch eigene Javadoc-Behandler (sogenannte *Doclets*) schreiben, wenn Sie die Information in den Dokumentationskommentaren auf eine individuelle Weise verarbeiten wollen (zum Beispiel, wenn Sie eine eigene Formatierung wünschen). Sie finden eine Einführung in Doclets im Anhang von *http://www.mindview.net/Books/BetterJava*.

[75] Die folgenden Seiten sind lediglich eine Einführung und ein Überblick über die Grundzüge von Javadoc. Sie finden in der Dokumentation des JDK eine vollständige Beschreibung der Technologie. Achten Sie beim Entpacken der Dokumentation auf das Unterverzeichnis *tooldocs* (oder klicken Sie den Verweis *tooldocs* an).

3.8.2 Freistehende und eingebettete Tags

[76] Ein Dokumentationskommentar beginnt mit dem Symbol `/**`, endet mit `*/` und kann zwei Typen von Elementen beinhalten: Eingebettetes HTML und Javadoc-Tags. Javadoc-Tags sind nur zwischen den Symbolen `/**` und `*/` erlaubt. *Freistehende Javadoc-Tags* beginnen mit dem At-Zeichen (`@`) und stehen am Anfang einer Kommentarzeile, wobei ein führendes Sternchen (`*`) ignoriert wird. *Eingebettete Javadoc-Tags* beginnen ebenfalls mit dem At-Zeichen, können überall in einer Kommentarzeile vorkommen und sind von einem Paar geschweifter Klammern umgeben.

[77] Es gibt drei „Typen“ von Dokumentationskommentaren, die sich auf die Komponente beziehen, welche der Dokumentationskommentar vorausgeht: Klassen-, Feld- und Methodenkommentar. Ein „Klassenkommentar“ steht unmittelbar vor der Definition einer Klasse, ein „Feldkommentar“ unmittelbar vor der Deklaration eines Feldes und ein „Methodenkommentar“ direkt vor der Definition einer Methode, zum Beispiel:

```
//: object/Documentation1.java
/** A class comment */
public class Documentation1 {
    /** A field comment */
    public int i;
    /** A method comment */
    public void f() {}
} ///:~
```

Beachten Sie, daß Javadoc im Standardverhalten nur die Dokumentationskommentare von Komponenten mit den Modifikatoren `public` und `protected` verarbeitet. Dokumentationskommentare bei privaten Komponenten und Komponenten unter Packagezugriff (siehe Kapitel 7) werden nicht beachtet, liefern also keine Ausgabe. (Wenn Sie `javadoc` aber mit dem Schalter `-private` aufrufen, werden auch Dokumentationskommentare bei privaten Komponenten einbezogen.) Das Standardverhalten ist sinnvoll, da nur als `public` oder `protected` deklarierte Komponenten außerhalb ihrer Datei, also aus der Perspektive der Clientprogrammierers verfügbar sind.

[78] Die Javadoc-Ausgabe zu dieser Datei ist eine HTML-Datei im standardisierten Format der API-Dokumentation von Java, so daß sich die Clientprogrammierer mühelos mit der komfortablen Dokumentation ihrer Klassen zurechtfinden.

3.8.3 Eingebettetes HTML

[79] Javadoc gibt HTML-Tags an das generierte HTML-Dokument weiter. Das ermöglicht Ihnen, HTML in vollem Umfang zu nutzen. Die primäre Motivation ist allerdings das Formatieren des Quelltextes, zum Beispiel:

```
//: object/Documentation2.java
/**
  * <pre>
  * System.out.println(new Date());
  * </pre>
```

```
  */
public class Documentation2 {}
///:~
```

Sie können HTML-Tags zur Formatierung Ihrer Dokumentation verwenden, wie bei jeder gewöhnlichen HTML-Seite auch:

```
//: object/Documentation3.java
/**
  * You can <em>even</em> insert a list:
  * <ol>
  * <li> Item one
  * <li> Item two
  * <li> Item three
  * </ol>
  */
public class Documentation3 {}
///:~
```

Beachten Sie, daß Javadoc in Dokumentationskommentaren Sternchen (`*`) am Zeilenanfang zusammen mit führendem Leerraum ignoriert. Javadoc formatiert den Inhalt des Dokumentationskommentares um, so daß er der standardisierten Erscheinungsform entspricht. Verwenden Sie bei eingebettetem HTML keine Überschriften wie `<h1>` oder `<hr>`, da Javadoc eigene Gliederungsebenen verwendet, die von Ihren Überschriften störend beeinflußt werden. Alle drei Varianten von Dokumentationskommentaren (Klasse, Feld und Methode) unterstützen eingebettetes HTML.

3.8.4 Einige ausgewählte Tags

[80] Dieser Unterabschnitt stellt einige Javadoc-Tags zusammen, die Sie in Ihrer Dokumentation verwenden können. Lesen Sie, bevor Sie ernsthaft versuchen, Dokumentation mit Javadoc zu schreiben, die verschiedenen Verwendungsmöglichkeiten von Javadoc in der JDK-Dokumentation nach.

3.8.4.1 Das @see-Tag

[81] Das `@see`-Tag ermöglicht das Anlegen einen Bezuges zur Dokumentation einer anderen Klasse. Javadoc generiert einen HTML-Verweis auf die entsprechende Dokumentation. Es gibt die folgenden Varianten:

```
@see classname
@see fully-qualified-classname
@see fully-qualified-classname#method-name
```

Jedes `@see`-Tag bewirkt einen „See Also“-Eintrag zur entsprechenden Dokumentation. Javadoc prüft die Verweisziele nicht auf Gültigkeit.

3.8.4.2 Das {@link package.class#member label}-Tag

[82] Das `{@link package.class#member label}`-Tag ähnelt dem `@see`-Tag, mit der Ausnahme, daß es in eine Kommentarzeile eingebettet werden kann und den `member label` anstelle von „See Also“ als Beschriftung des HTML-Verweises verwendet.

3.8.4.3 Das {@docRoot}-Tag

[83] [Unsicher: Recherchieren/Ausprobieren] Das `{@docRoot}`-Tag liefert den relativen Pfad zum Wurzelverzeichnis der Dokumentation. Nützlich bei expliziten Verweisen auf Seiten im Dokumentationsbaum.

3.8.4.4 Das {@inheritDoc}-Tag

[84] [Unsicher: Recherchieren/Ausprobieren] Das `{@inheritDoc}`-Tag vererbt die Dokumentation der nächsten Basisklasse dieser Klasse in den aktuellen Dokumentationskommentar.

3.8.4.5 Das @version-Tag

[85] Das `@version`-Tag hat das Format:

```
@version version-information
```

Dabei ist `version-information` eine beliebige aussagekräftige passende Information. [Unsicher: Recherchieren/Ausprobieren] Wenn Sie `javadoc` mit dem Schalter `-version` aufrufen, wird die Versionsinformation in die generierte HTML-Dokumentation eingesetzt.

3.8.4.6 Das @author-Tag

[86] Das `@author`-Tag hat das Format:

```
@author author-information
```

Dabei ist `author-information` voraussichtlich Ihr Name, Ihre E-Mailadresse oder eine andere passende Information. [Unsicher: Recherchieren/Ausprobieren] Wenn Sie `javadoc` mit dem Schalter `-author` aufrufen, wird die Autoreninformation in die generierte HTML-Dokumentation eingesetzt.

[87] Sie können bei mehreren Autoren mehrere `@author`-Tags angeben, die aber unmittelbar aufeinander folgen müssen. Alle Autoreninformationen werden in der generierten HTML-Seite zu einem einzigen Absatz zusammengefaßt.

3.8.4.7 Das @since-Tag

[88] Das `@since`-Tag dient zur Angabe, ab welcher Version des Quelltextes eine bestimmte Eigenschaft oder Fähigkeit vorhanden ist. Dieses Tag gibt in der API-Dokumentation von Java an, welche JDK-Version mindestens erforderlich ist.

3.8.4.8 Das @param-Tag

[89] Das `@param`-Tag wird zur Dokumentation von Methoden verwendet und hat das Format:

```
@param parameter-name description
```

Dabei ist `parameter-name` der Name (Bezeichner) in der Parameterliste der Methode und `description` ein gegebenenfalls mehrzeiliger Beschreibungstext. Der Beschreibungstext endet mit dem nächsten Tag. Sie können beliebig viele `@param`-Tags verwenden, voraussichtlich aber eines pro Parameter.

3.8.4.9 Das @return-Tag

[90] Das `@return`-Tag wird zur Dokumentation von Methoden verwendet und hat das Format:

```
@return description
```

Dabei ist `description` ein gegebenenfalls mehrzeiliger Beschreibungstext über den Rückgabewert.

3.8.4.10 Das @throws-Tag

[91] Das Konzept der Ausnahme wird in Kapitel 13 behandelt. Kurzfassung: Ausnahmen sind Objekte, die aus einer Methode ausgeworfen werden können, wenn die Verarbeitung der Methode scheitert. Beim Scheitern einer Methode kann zwar nur eine einzige Ausnahme ausgeworfen werden, aber eine Methode kann eine beliebige Anzahl verschiedener Ausnahmetypen deklarieren, zu denen jeweils eine Beschreibung in der Methodendokumentation gehört. Das `@throws`-Tag wird zur Dokumentation von Methoden verwendet und hat das Format:

```
@throws fully-qualified-class-name description
```

Dabei ist `fully-qualified-class-name` der eindeutige Name einer irgendwo definierten Ausnahmeklasse und `description` ein gegebenenfalls mehrzeiliger Beschreibungstext, der beschreibt, aus welchem Grund ein Aufruf dieser Methode eine Ausnahme dieses Typs hervorrufen kann.

3.8.4.11 Das @deprecated-Tag

[92] Das `@deprecated`-Tag dient dazu, Eigenschaften oder Fähigkeiten zu kennzeichnen, die von einer verbesserten Eigenschaft oder Fähigkeit abgelöst worden sind. Das `@deprecated`-Tag repräsentiert den Vorschlag, die gekennzeichnete Eigenschaft oder Fähigkeit nicht mehr länger zu nutzen, da sie in Zukunft wahrscheinlich entfernt wird. In der SE 5 wurde das `@deprecated`-Tag von der Annotation `@Deprecated` abgelöst (mehr über Annotationen in Kapitel 21).

3.8.5 Ein Dokumentationsbeispiel

[93] Das folgende Beispiel zeigt nochmals das erste Java-Programm von Seite 66, diesesmal aber nicht Dokumentationskommentaren:

```
//: object/HelloDate.java
import java.util.*;

/** The first Thinking in Java example program.
  * Displays a string and today's date.
  * @author Bruce Eckel
  * @author www.MindView.net
  * @version 4.0
  */
public class HelloDate {
    /** Entry point to class & application.
      * @param args array of string arguments
      * @throws exceptions No exceptions thrown
      */
    public static void main(String[] args) {
        System.out.println("Hello, it's: ");
        System.out.println(new Date());
    }
```

```
} /* Output: (55% match)
      Hello, it's:
      Wed Oct 05 14:39:36 MDT 2005
   *///:~
```

Die erste Zeile zeigt meine persönliche Notation für eine Kommentarzeile, die den Namen der Quelltextdatei enthält (`//:`). Die Zeile gibt insbesondere das Unterverzeichnis an, in dem sich die Datei befindet (*object* für dieses Kapitel). Die letzte enthält ebenfalls ein von mir definiertes Abschlußsymbol (`*///:~`). Es zeigt das Ende des Quelltextes an und ermöglicht automatisches Aktualisieren im Text dieses Buches, nachdem das Programm vom Compiler überprüft und übersetzt wurde.

[94] Das Symbol `/* Output:` zeigt den Beginn der Ausgabe an, die von diesem Programm erzeugt wird. Das Programm kann somit automatisch getestet werden, um seine Genauigkeit zu verifizieren. In diesem Fall (55% Übereinstimmung) erkennt das Testsystem, daß die Ausgabe zwischen zwei Programmaufrufen erheblich abweichen kann und erwartet nur eine 55 prozentige Übereinstimmung mit der obigen Ausgabe. Die meisten Beispiele in diesem Buch, die eine Ausgabe erzeugen, liefern ihre Ausgabe in dieser kommentierten Art und Weise, damit Sie nachvollziehen können ob die Ausgabe richtig ist.

3.9 Formatierungs- und Benennungsrichtlinien

[95–96] Die *Code Conventions for the Java Programming Language*[8] besagen, daß der erste Buchstabe eines Klassennamens ein Großbuchstabe ist. Besteht der Klassenname aus mehreren Worten, so werden die Worte ohne Unterstriche direkt zusammengefügt, wobei der erste Buchstabe jedes Wortes ein Großbuchstabe („Binnenmajuskel“) ist, zum Beispiel:

```
class AllTheColorsOfTheRainbow { // ...
```

Diese Schreibweise wird gelegentlich als „CamelCase“ bezeichnet. Bei fast allen übrigen Bezeichnern, das heißt bei Methoden, Feldern und lokalen Variablen sowohl primitiven Typs als auch bei Referenztyp, gilt dieselbe Konvention, wobei allerdings der erste Buchstabe stets ein Kleinbuchstabe ist, zum Beispiel:

```
class AllTheColorsOfTheRainbow {
    int anIntegerRepresentingColors;
    void changeTheHueOfTheColor(int newHue) {
        // ...
    }
    // ...
}
```

Die Clientprogrammierer müssen alle diese Namen schreiben. Seien Sie also rücksichtsvoll. Der Quelltext in der Standardbibliothek von Java folgt der Platzierung der geschweiften Klammernpaare, die Sie in diesem Buch sehen.

3.10 Zusammenfassung

[97] Das Ziel dieses Kapitel sind genügend Java-Kenntnisse, um zu verstehen, wie Sie ein einfaches Programm schreiben können. Sie haben außerdem einen Überblick über die Programmiersprache

[8] Sie finden die Formatierungsrichtlinie unter der Webadresse *http://java.sun.com/docs/codeconv*. Aus Platzgründen konnten nicht alle Aspekte der Richtlinie in diesem Buch und den Schulungsunterlagen eingehalten werden. Sie werden aber erkennen, daß die hier verwendete Formatierung dem Java-Standard so weit wie möglich entspricht.

und einige ihrer grundlegenden Konzepte erhalten. Die Beispiele hatten allerdings die Form von Kochrezepten. Die nächsten beiden Kapitel behandeln die wichtigsten Operatoren in der Java-Programmierung und die Steuerung des Programmablaufs.

3.11 Übungsaufgaben

[98–99] Die Übungsaufgaben sind in der Regel über das ganze Kapitel verteilt. Da Sie aber in diesem Kapitel das Schreiben einfacher Programme erst erlernt haben, stehen die Übungsaufgaben ausnahmeweise am Kapitelende. Die eingeklammerte Zahl nach der Nummer einer Übungsaufgabe ist ein Maß für deren Schwierigkeit auf einer Skala von 1 bis 10.

Übungsaufgabe 1: (2) Schreiben Sie eine Klasse, die je ein nicht initialisiertes Feld vom Typ `int` beziehungsweise `char` enthält. Geben Sie die Inhalte beider Felder aus, um zu verifizieren, daß die Initialisierung mit Standardwerten ausgeführt wird. ■

Übungsaufgabe 2: (1) Schreiben Sie ein „Hello World!"-Programm. Orientieren Sie sich dabei am Beispiel *HelloDate.java* in diesem Kapitel (Seite 66). Sie müssen nur eine einzige Methode definieren, nämlich die `main()`-Methode, die beim Programmstart aufgerufen wird. Denken Sie an den `static`-Modifikator und an die Argumentliste, auch wenn Sie keine Argumente übergeben. Übersetzen Sie das Programm per `javac` und rufen Sie es per `java` auf. Wenn Sie eine andere Entwicklungsumgebung als das Java Development Kit (JDK) verwenden, finden Sie heraus, wie Programme unter Ihrer Umgebung übersetzt und aufgerufen werden. ■

Übungsaufgabe 3: (1) Suchen Sie die Quelltextfragmente der Klasse `ATypeName` (Seite 60) zusammen und kombinieren Sie sie zu einem Programm, das sich übersetzen und aufrufen läßt. ■

Übungsaufgabe 4: (1) Suchen Sie die Quelltextfragmente der Klasse `DataOnly` (Seite 60) zusammen und kombinieren Sie sie zu einem Programm, das sich übersetzen und aufrufen läßt. ■

Übungsaufgabe 5: (1) Ändern Sie Übungsaufgabe 4, so daß die Felder der Klasse `DataOnly` in der `main()`-Methode bewertet und ausgegeben werden. ■

Übungsaufgabe 6: (2) Schreiben Sie ein Programm, das die als Fragment definierte `storage()`-Methode (Seite 62) beinhaltet und aufruft. ■

Übungsaufgabe 7: (1) Suchen Sie die Quelltextfragmente der Klasse `Incrementable` (Seite 66) zusammen und kombinieren Sie sie zu einem Programm, das sich übersetzen und aufrufen läßt. ■

Übungsaufgabe 8: (3) Schreiben Sie ein Programm, um zu zeigen, daß unabhängig von der Anzahl erzeugter Objekte einer Klasse für ein statisches Feld dieser Klasse nur ein Exemplar existiert. ■

Übungsaufgabe 9: (2) Schreiben Sie ein Programm, um zu zeigen, daß Autoboxing zwischen jedem primitiven Typ und dem jeweiligen Wrappertyp funktioniert. ■

Übungsaufgabe 10: (2) Schreiben Sie ein Programm, das drei auf der Kommandozeile übergebene Argumente ausgibt. Werten Sie zu diesem Zweck das `String`-Array der `main()`-Methode aus. ■

Übungsaufgabe 11: (1) Wandeln Sie das Beispiel *AllTheColorsOfTheRainbow.java* (Seite 74) in ein Programm um, das sich übersetzen und aufrufen läßt. ■

Übungsaufgabe 12: (2) Suchen Sie den Quelltext der zweiten Version von *HelloDate.java* (das kleine Beispiel für Dokumentationskommentare, Seite 73). Rufen Sie `javadoc` auf *HelloDate.java* auf und laden Sie das Ergebnis in Ihren Webbrowser. ■

Übungsaufgabe 13: (1) Rufen Sie `javadoc` auf auf den Beispielen *Documentation1.java* (Seite 70), *Documentation2.java* (Seite 70) und *Documentation3.java* (Seite 71) auf. Laden Sie die generierte Dokumentation in Ihren Webbrowser. ■

Übungsaufgabe 14: (1) Legen Sie in der Dokumentation von Übungsaufgabe 13 eine HTML-Aufzählungsliste an. ■

Übungsaufgabe 15: (1) Legen Sie im Programm von Übungsaufgabe 2 Dokumentationskommentare an. Extrahieren Sie die Dokumentation anschließend per `javadoc` im HTML-Format und laden Sie sie in Ihren Webbrowser. ■

Übungsaufgabe 16: (1) Suchen Sie das Beispiel *Overloading.java* in Abschnitt 6.2 (Seite 133) und legen Sie dort Dokumentationskommentare an. Extrahieren Sie die Dokumentation anschließend per `javadoc` im HTML-Format und laden Sie sie in Ihren Webbrowser. ■

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 4

Operatoren

Inhaltsübersicht

[0] Auf der untersten Ebene werden Daten bei Java mit Hilfe von Operatoren verarbeitet.

[1] Da Java von C++ „geerbt hat“, sind C/C++-Programmierern die meisten Operatoren von Java vertraut.

[2] Wenn Ihnen die Syntax von C oder C++ vertraut ist, können Sie dieses und das nächste Kapitel durchblättern und sich auf die Stellen konzentrieren, an denen Java von C und C++ abweicht. Falls

Sie sich beim Durchsehen dieser beiden Kapitel aber noch ein wenig unsicher fühlen, sollten Sie das Multimediaseminar *Thinking in C* durchgehen, das Sie unter der Webadresse *http://www.mindview.net* kostenlos herunterladen können.

4.1 Vereinfachte print-Anweisungen

[3] Im vorigen Kapitel haben Sie die `println()`-Anweisung von Java kennengelernt:

```
System.out.println("Rather a lot to type");
```

Vielleicht haben Sie auch den Eindruck, daß dies nicht nur eine Menge Schreibaufwand (viele redundante Sehnenstöße), sondern auch unübersichtlich zu lesen ist. Die meisten Sprachen vor und nach Java haben einen einfacheren Ansatz für so häufig verwendete Anweisungen gewählt.

[4] Unterabschnitt 7.1.3 führt das Konzept des statischen Imports ein, das seit Version 5 der Java Standard Edition (SE 5) zur Verfügung steht und stellt eine kleine Bibliothek vor, die `print()`-Anweisungen vereinfacht. Die Einzelheiten sind aber nicht notwendig, um mit der Nutzung dieser Bibliothek zu beginnen. Die Bibliothek gestattet, daß Programmbeispiel aus dem vorigen Kapitel folgendermaßen umzuschreiben:

```
//: operators/HelloDate.java
import java.util.*;
import static net.mindview.util.Print.*;

public class HelloDate {
    public static void main(String[] args) {
        print("Hello, it's: ");
        print(new Date());
    }
} /* Output: (55% match)
      Hello, it's:
      Wed Oct 05 14:39:05 MDT 2005
   *///:~
```

Die Anweisungen sind viel klarer. Beachten Sie das zusätzliche Schlüsselwort `static` in der zweiten `import`-Anweisung.

[5] Wenn Sie diese Bibliothek verwenden möchten, müssen Sie die Quelltextdistribution zu diesem Buch von *http://www.mindview.net* oder einem Spiegelserver herunterladen. Entpacken Sie den Verzeichnisbaum mit dem Quelltext und deklarieren Sie sein Wurzelverzeichnis im Klassenpfad (Umgebungsvariable `$CLASSPATH`) Ihres Rechners. (Unterabschnitt 7.1.2 gibt eine Einführung in das Thema „Klassenpfad". Eventuell müssen Sie sich aber auch schon früher mit dem Klassenpfad beschäftigen. Dies ist leider ein häufiger Kampf im Zusammenhang mit Java.)

[6] Auch wenn die Hilfsklasse `net.mindview.util.Print` in der Regel den Quelltext vereinfacht, ist ihre Verwendung nicht überall angebracht. Enthält ein Programm nur eine geringe Anzahl von `print()`-Anweisungen, so verzichte ich auf die `import`-Anweisung und schreibe die volle Anweisung `System.out.println()`.

Übungsaufgabe 1: (1) Schreiben Sie ein Programm, das sowohl die abgekürtze als auch die normale `print()`-Anweisung verwendet. ■

4.2 Operatoranwendung bei Java

[7] Ein Operator erwartet wenigstens einen Operanden (Argument) und erzeugt einen neuen Wert. Die Operanden werden zwar anders notiert, als die Argumente eines gewöhnlichen Methodenaufrufes, aber die Auswertung eines Operators mit Operanden liefert denselben Effekt. Addition und unäre, das heißt einargumentige Inkrementierung (`+`), Subtraktion und unäre Dekrementierung, (`-`), Multiplikation (`*`), Division (`/`) und Zuweisung (`=`) funktionieren in allen Programmiersprachen weitestgehend auf dieselbe Weise.

[8] Alle Operatoren erzeugen einen Wert aus ihren Operanden. Es gibt einige Operatoren die den Wert eines Operanden selbst ändern. Dieses Änderungsverhalten heißt *Seiteneffekt*. Operatoren mit Seiteneffekt werden in der Regel nur aufgrund ihres Seiteneffekts verwendet. Denken Sie aber daran, daß Sie den „Rückgabewert“ eines Operator mit Seiteneffekt ebenso nutzen können, wie das Ergebnis eines Operators ohne Seiteneffekt.

[9] Fast alle Operatoren arbeiten nur mit Argumenten primitiven Typs. Ausnahmen sind der Zuweisungsoperator `=`, die Vergleichsoperatoren `==` und `!=`, die auch Referenzen auf Objekte akzeptieren und die von der Klasse `String` unterstützten Konkatenationsoperatoren `+` und `+=`.

4.3 Rangfolge von Operatoren

[10] Die Operatorrangfolge (Operatorpräzedenz) definiert, wie ein Ausdruck ausgewertet wird, der mehr als einen Operator enthält. Java definiert spezifische Regeln, die die Auswertungsreihenfolge festlegen. Am leichtesten merkt man sich, daß Multiplikationen und Divisionen vor Additionen und Subtraktionen ausgeführt werden („Punkt vor Strich“). Programmierer vergessen häufig die Regeln über die Rangfolge. Es ist daher sinnvoll, Klammern zu verwenden, um die Auswertungsreihenfolge explizit festzulegen. Beachten Sie die Anweisungen (1) und (2) im folgenden Beispiel:

```
//: operators/Precedence.java
public class Precedence {
    public static void main(String[] args) {
        int x = 1, y = 2, z = 3;
        int a = x + y - 2/2 + z; // (1)
        int b = x + (y - 2)/(2 + z); // (2)
        System.out.println(”a = ” + a + ” b = ” + b);
    }
} /* Output:
      a = 5 b = 1
   *///:~
```

Die beiden Zeilen sehen im Großen und Ganzen gleich aus, während die Ausgabe allerdings dokumentiert, daß die Ausdrücke (aufgrund der Klammern) völlig verschiedene Bedeutungen haben.

[11] Beachten Sie, daß das Argument der Anweisung `System.out.println()` den `+`-Operator enthält. In diesem Kontext repräsentiert `+` die Verknüpfung („Konkatenation“) von `String`-Objekten („Zeichenketten“), erforderlichenfalls auch die Umwandlung in ein `String`-Objekt. Erkennt der Compiler ein `String`-Objekt, gefolgt von einem `+`-Operator und einem Operanden, der kein `String`-Objekt ist, so versucht der Compiler, den zweiten Operanden in ein `String`-Objekt umzuwandeln. Wie Sie an der Ausgabe sehen, wurden die `int`-Werte in `a` und `b` erfolgreich in `String`-Objekte umgewandelt.

4.4 Zuweisungen

[12] Ein Zuweisung wird mit dem Operator = („Zuweisungsoperator“) bewerkstelligt und besagt: „Kopiere den Wert des rechten Operanden (häufig als `rvalue` bezeichnet) in den linken Operanden (häufig als `lvalue` bezeichnet).“ Der `rvalue` ist eine beliebige Konstante, ein Feld oder eine lokale Variable primitiven oder auch nicht primitiven Typs oder ein Ausdruck, dessen Auswertung einen Wert liefert. Der `lvalue` ist dagegen ein *bestimmtes Feld* oder eine *bestimmte lokale Variable*, das heißt es muß ein physikalischer Bereich im Arbeitsspeicher existieren, um den zugewiesenen Wert zu speichern. Sie können einem Feld beziehungsweise einer lokalen Variablen `a` zum Beispiel den konstanten Wert 4 zuweisen:

```
a = 4;
```

Andererseits können Sie einem konstanten Wert nichts zuweisen, das heißt eine Konstante ist kein zulässiger `lvalue`. (Die Zuweisung `4 = a;` ist nicht möglich.)

[13] Die Zuweisung von Werten primitiven Typs ist eine einfache Angelegenheit. Da ein Feld beziehungsweise eine lokale Variable primitiven Typs den eigentlichen Wert und keine Referenz auf ein Objekt enthält, wird bei der Zuweisung eines Wertes von primitivem Typ der Inhalt des Quellspeicherbereiches in den Zielspeicherbereich kopiert. Sind beispielsweise `a` und `b` Felder oder lokale Variablen primitiven Typs, so bewirkt die Zuweisung `a = b`, daß der Inhalt von `b` (Quellspeicherbereich) in `a` (Zielspeicherbereich) kopiert wird. Wird `a` im weiteren Verlauf des Programms modifiziert, so bleibt `b` von dieser Änderung natürlich unberührt. Als Programmierer können Sie dieses Verhalten in den meisten Situationen erwarten.

[14] Bei der Zuweisung von Werten an eine **Referenzvariable**, das heißt ein Feld oder eine lokale Variable zur Aufnahme einer Referenz auf ein Objekt, liegen die Dinge dagegen anders. Wenn Sie ein Objekt über eine solche Referenzvariable „bedienen“, arbeiten Sie nicht direkt mit dem Objekt selbst, sondern mit einer Referenz auf das Objekt. Wenn Sie „einem Objekt ein anderes Objekt zuweisen“, kopieren Sie eigentlich den Inhalt der Quellreferenzvariablen in die Zielreferenzvariable. Sind beispielsweise `c` und `d` referenzwertige (nicht primitive) Felder oder lokale Variablen, so bewirkt die Zuweisung `c = d`, daß `c` (Zielreferenzvariable) und `d` (Quellreferenzvariable) auf dasselbe Objekt verweisen, welches ursprünglich nur von `d` referenziert wurde. Das folgende Beispiel demonstriert dieses Verhalten:

```
//: operators/Assignment.java
// Assignment with objects is a bit tricky.
import static net.mindview.util.Print.*;

class Tank {
    int level;
}

public class Assignment {
    public static void main(String[] args) {
        Tank t1 = new Tank();
        Tank t2 = new Tank();
        t1.level = 9;
        t2.level = 47;
        print("1: t1.level: " + t1.level + ", t2.level: " + t2.level);
        t1 = t2;
        print("2: t1.level: " + t1.level + ", t2.level: " + t2.level);
        t1.level = 27;
        print("3: t1.level: " + t1.level + ", t2.level: " + t2.level);
    }
} /* Output:
```

```
    1: t1.level: 9, t2.level: 47
    2: t1.level: 47, t2.level: 47
    3: t1.level: 27, t2.level: 27
  *///:~
```

Die lokalen Variablen `t1` und `t2` der `main()`-Methode referenzieren zwei Objekte der einfachen Klasse `Tank`. Zunächst wird das `level`-Feld in jedem der beiden `Tank`-Objekte indiviuell initialisiert. Anschließend wird der Inhalt von `t2` `t1` zugewiesen und das `level`-Feld des von `t1` referenzierten Objektes geändert. Während Sie bei vielen Programmiersprachen erwarten, daß die Werte von `t1` und `t2` stets voneinander unabhängig sind, schlägt sich, aufgrund der Zuweisung von Referenzen, eine Änderung des von `t1` referenzierten Objektes auch in `t2` nieder. Beide lokalen Variablen, `t1` und `t2`, enthalten ein und dieselbe Referenz, verweisen also auf ein und dasselbe Objekt. (Die in `t1` ursprünglich gespeicherte Referenz auf das Objekt mit dem `level`-Wert 9 wurde durch die Zuweisung überschrieben und ist effektiv verloren. Das zuvor von `t1` referenzierte Objekt wird der automatischen Speicherbereinigung übergeben.)

[15] Die Existenz mehrerer Referenzvariablen, die auf ein und dasselbe Objekt verweisen, wird als *Aliasing* bezeichnet und ist bei Java eine grundlegende Art und Weise im Umgang mit Objekten. Was aber, wenn Sie im obigen Beispiel keine zweite Referenzvariable auf das von `t2` referenzierte `Tank`-Objekt anlegen wollen? Sie können das Aliasing mit der folgenden Zuweisung umgehen:

```
t1.level = t2.level;
```

Diese Zuweisung erhält die beiden separaten Objekte, statt eines zu verwerfen und beide Referenzvariablen auf dasselbe Objekt verweisen zu lassen. Sie werden bald erkennen, daß direktes Abfragen oder Ändern der Felder eines Objektes den Prinzipien guter objektorientierter Programmierung widerspricht. Dies ist kein einfaches Thema. Merken Sie sich einstweilen, daß Sie bei Existenz mehrerer Referenzvariablen für ein und dasselbe Objekt Überraschungen erleben können.

Übungsaufgabe 2: (1) Legen Sie eine Klasse mit einem Feld vom Typ `float` an und führen Sie Aliasing vor. ■

4.4.1 Existenz mehrerer Referenzvariablen während eines Methodenaufrufs

[16] Bei der Übergabe eine Referenz an eine Methode verweist ebenfalls mehr als eine Referenzvariable auf ein und dasselbe Objekt:

```
//: operators/PassObject.java
// Passing objects to methods may not be
// what you're used to.
import static net.mindview.util.Print.*;

class Letter {
    char c;
}

public class PassObject {
    static void f(Letter y) {
        y.c = 'z';
    }
    public static void main(String[] args) {
        Letter x = new Letter();
        x.c = 'a';
        print("1: x.c: " + x.c);
        f(x);
        print("2: x.c: " + x.c);
```

```
    }
} /* Output:
      1: x.c: a
      2: x.c: z
   *///:~
```

Bei vielen Programmiersprachen arbeitet die Methode `f()` innerhalb ihres Geltungsbereiches mit einer Kopie ihres Argumentes. Aber auch hier wird eine Referenz übergeben, so daß die Zeile

```
y.c = 'z';
```

tatsächlich das außerhalb des Geltungsbereiches von `f()` liegende Objekt modizifiert.

[17] Die Existenz mehrerer Referenzvariablen, die auf ein und dasselbe Objekt verweisen, ist eine komplexe Angelegenheit und wird in einem der Online-Anhänge zu diesem Buch behandelt. Dennoch sollten Sie sich des Problems an dieser Stelle bewußt sein, so daß Sie auf Fallen aufmerksam werden.

Übungsaufgabe 3: (1) Legen Sie eine Klasse mit einem Feld vom Typ `float` an und führen Sie Aliasing während eines Methodenaufrufs vor. ■

4.5 Mathematische Operatoren

[18] Die grundlegenden mathematischen Operatoren von Java stimmen mit denjenigen der meisten anderen Programmiersprachen überein: Addition (`+`), Subtraktion (`-`), Multiplikation (`*`), Division (`/`) und Divisionsrest bei ganzzahliger Teilung (`%`). Das Ergebnis einer ganzzahligen Division wird abgebrochen, nicht gerundet.

[19] Java kennt ebenfalls die von C und C++ gewohnte abgekürzte Schreibweise, eine Operation auszuführen und das Ergebnis anschließend zuzuweisen. Die Notation besteht aus dem Operator, gefolgt von einem Gleichheitszeichen und gilt konsistent für alle in der Sprache vorhandenen Operatoren (soweit sinnvoll). Der Ausdruck `x += 4` beispielsweise, addiert 4 zum Wert von `x` und weist `x` das Ergebnis zu.

[20] Das folgende Beispiel zeigt die Verwendung der mathematischen Operatoren:

```
//: operators/MathOps.java
// Demonstrates the mathematical operators.
import java.util.*;
import static net.mindview.util.Print.*;

public class MathOps {
    public static void main(String[] args) {
        // Create a seeded random number generator:
        Random rand = new Random(47);
        int i, j, k;
        // Choose value from 1 to 100:
        j = rand.nextInt(100) + 1;
        print("j : " + j);
        k = rand.nextInt(100) + 1;
        print("k : " + k);
        i = j + k;
        print("j + k : " + i);
        i = j - k;
        print("j - k : " + i);
        i = k / j;
        print("k / j : " + i);
        i = k * j;
```

```
        print("k * j : " + i);
        i = k % j;
        print("k % j : " + i);
        j %= k;
        print("j %= k : " + j);
        // Floating-point number tests:
        float u, v, w; // Applies to doubles, too
        v = rand.nextFloat();
        print("v : " + v);
        w = rand.nextFloat();
        print("w : " + w);
        u = v + w;
        print("v + w : " + u);
        u = v - w;
        print("v - w : " + u);
        u = v * w;
        print("v * w : " + u);
        u = v / w;
        print("v / w : " + u);
        // The following also works for char,
        // byte, short, int, long, and double:
        u += v;
        print("u += v : " + u);
        u -= v;
        print("u -= v : " + u);
        u *= v;
        print("u *= v : " + u);
        u /= v;
        print("u /= v : " + u);
    }
} /* Output:
      j : 59
      k : 56
      j + k : 115
      j - k : 3
      k / j : 0
      k * j : 3304
      k % j : 56
      j %= k : 3
      v : 0.5309454
      w : 0.0534122
      v + w : 0.5843576
      v - w : 0.47753322
      v * w : 0.028358962
      v / w : 9.940527
      u += v : 10.471473
      u -= v : 9.940527
      u *= v : 5.2778773
      u /= v : 9.940527
   *///:~
```

Das Programm verwendet ein Objekt der Klasse `Random`, um Zufallszahlen zu erzeugen. Wenn Sie ein `Random`-Objekt ohne Argumente erzeugen, verwendet Java die aktuelle Uhrzeit als Saat des Zufallsgenerators und erzeugt bei jedem Programmaufruf eine andere Ausgabe. Es ist für die Beispiele in diesem Buch aber wichtig, daß die angegebene Ausgabe am Ende eines Beispiels so konsistent wie möglich ist, damit die Ausgabe mit externen Hilfmitteln verifiziert werden kann. Durch Festlegen einer *Saat* beim Erzeugen des `Random`-Objektes (Anfangswert für den Zufallsgenerator durch den stets

dieselbe *anfangswertabhängige* Folge von Zufallszahlen generiert wird), liefert der Zufallsgenerator bei jedem Aufruf dieselbe Folge von Zufallswerten, so daß die Ausgabe verifizierbar wird.[1]

[21] Das Programm generiert mit Hilfe des `Random`-Objektes durch Aufrufen der Methoden `next()` und `nextFloat()` Zufallswerte verschiedener primitiver Typen, (Sie können auch die `Random`-Methoden `nextLong()` und `nextDouble()` aufrufen.) Das Argument der `nextInt()`-Methode definiert die obere Grenze der generierten Zufallswerte, genauer, den direkten Nachfolger des höchsten möglichen Zufallswertes. Die untere Grenze ist Null und muß durch einen Versatz um den Wert Eins ausgeschlossen werden, um der Division durch Null vorzubeugen.

Übungsaufgabe 4: (2) Schreiben Sie ein Programm, das aus einer konstanten Strecke und einer konstanten Zeit die Geschwindigkeit berechnet. ■

4.5.1 Der unäre Invertierungsoperator

[22] Die unären, das heißt einargumentigen Operatoren `-` und `+` („Vorzeichenoperatoren“) sind identisch mit den „gleichnamigen“ zweiargumentigen Operatoren. Der Compiler erkennt an der Schreibweise eines Ausdrucks, ob der unäre oder der binäre Operator gemeint ist. Die folgende Zeile läßt sich eindeutig interpretieren:

```
x = -a;
```

Der Compiler ist zwar in der Lage, den Ausdruck

```
x = a * -b;
```

richtig zu erkennen, aber die Schreibweise wirtk auf den menschlichen Leser eventuell konfus. Gelegentlich schafft eine explizite Klammerung mehr Klarheit:

```
x = a * (-b);
```

Der unäre Operator `-` invertiert das Vorzeichen des Wertes eines Feldes oder einer lokalen Variablen primitiven Typs. Der unäre Operator `+` ist zwar aus Symmetriegründen definiert, hat aber keine Wirkung.

4.6 Autoinkrement- und Autodekrementoperator

[23] Java verfügt, wie C, über eine Reihe syntaktischer Abkürzungen. Abkürzungen können das Schreiben des Quelltext erleichtern und seine Leserbarkeit entweder vereinfachen oder erschweren.

[24] Der Dekrement- und der Inkrementoperator, häufig auch als Autodekrement- beziehungsweise Autoinkrementoperator bezeichnet (kürzer **De-/Inkrementoperator** beziehungsweise **Autode-/inkrementoperator**), sind zwei nützliche Abkürzungen. Der **Dekrementoperator** ist ein doppeltes Minuszeichen (`--`) und bewirkt die „Abnahme um eine Einheit“. Der **Inkrementoperator** ist ein doppeltes Pluszeichen (`++`) und bewirkt die „Zunahme um eine Einheit“. Ist `a` zum Beispiel ein Feld oder eine lokale Variable vom Typ `int`, so ist der Ausdruck `++a` äquivalent zu `a = a+1`. Der De-/Inkrementoperator modifiziert nicht nur seinen Operanden, sondern liefert über seinen Operanden auch zugleich das Ergebnis der Operation.

[25] Der De-/Inkrementoperator tritt in zwei Varianten auf: Der Präfix- und der Postfixnotation. *Präinkrement* bedeutet, daß der Operator `++` vor dem Namen des Feldes oder der lokalen Variablen notiert ist, *Postinkrement* dagegen, daß der Operator `++` dem Bezeichner folgt. Analog bedeutet

[1] Einer meiner Mitstudenten hielt die Zahl 47 für „magisch“ und dabei ist es geblieben.

Postdekrement, daß der Operator `--` vor dem Namen des Feldes oder der lokalen Variablen notiert ist, *Postinkrement* dagegen, daß sich der Operator `--` an den Bezeichner anschließt. Bei der Präde-/inkrementnotation (zum Beispiel `++a` oder `--a`) wird die Operation *vor* der Rückgabe des Ergebnisses ausgeführt. Bei der Postde-/inkrementnotation (zum Beispiel `a++` oder `a--`) wird die Operation *nach* der Rückgabe des Ergebnisses ausgeführt. Ein Beispiel:

```
//: operators/AutoInc.java
// Demonstrates the ++ and -- operators.
import static net.mindview.util.Print.*;

public class AutoInc {
    public static void main(String[] args) {
        int i = 1;
        print("i : " + i);
        print("++i : " + ++i); // Pre-increment
        print("i++ : " + i++); // Post-increment
        print("i : " + i);
        print("--i : " + --i); // Pre-decrement
        print("i-- : " + i--); // Post-decrement
        print("i : " + i);
    }
} /* Output:
      i : 1
      ++i : 2
      i++ : 2
      i : 3
      --i : 2
      i-- : 2
      i : 1
   *///:~
```

Sie sehen, daß die Präfixnotation das Ergebnis *nach der Auswertung* des Ausdrucks liefert, die Postfixnotation dagegen *vor der Auswertung*. Der De-/Inkrementoperator ist, mit Ausnahme der Zuweisungsoperatoren, der einzige Operator mit Seiteneffekt: Er modifiziert seinen Operanden, statt ihn nur auszuwerten.

[26] Der Inkrementoperator liefert eine Erklärung für den Namen C++, nämlich die Andeutung eines Schrittes über C hinaus. Bill Joy, einer der Erfinder von Java, sagte in einem frühen Vortrag, Java sei C++-- (C plus plus minus minus), um darauf hinzuweisen, daß Java eine Art C++ ohne die unnötig harten Bestandteile und somit eine viel einfachere Sprache sei. Sie werden beim Studium dieses Buches sehen, daß Java in vieler Hinsicht tatsächlich einfacher ist als C++, während andernorts kaum Unterschiede zu verzeichnen sind.

4.7 Vergleichsoperatoren

[27] Vergleichsoperatoren werten die Beziehung zwischen den Inhalten ihrer Operanden aus und liefern ein boolesches Ergebnis. Die Auswertung eines Vergleichsausdrucks ergibt `true`, wenn die Beziehung wahr ist und andernfalls `false`. Die Vergleichsoperatoren sind: „kleiner als“ (`<`), „größer als“ (`>`), „kleiner oder gleich“ (`<=`), „größer oder gleich“ (`>=`) sowie Gleichheit, genauer Gleichwertigkeit/Äquivalenz (`==`) beziehungsweise Ungleichheit, genauer Ungleichwertigkeit oder Nicht-Äquivalenz (`==`) Die Prüfung auf Gleichheit beziehungsweise Ungleichheit ist mit jedem primitiven Typ möglich, während die übrigen Vergleichsoperatoren keine Operanden vom Typ `boolean` akzeptieren.

4.7.1 Gleichwertigkeit von Objekten

[28] Die Vergleichsoperatoren `==` und `!=` akzeptieren auch Referenzen auf Objekte, wobei das Verhalten der Operatoren bei unerfahrenen Java-Programmierern häufig auf Unverständnis stößt. Ein Beispiel:

```
//: operators/Equivalence.java
public class Equivalence {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1 == n2);
        System.out.println(n1 != n2);
    }
} /* Output:
      false
      true
   *///:~
```

Die Anweisung `System.out.println(n1 == n2)` gibt das Ergebnis des booleschen Vergleichs in ihrem Argument zurück. Sollte die Ausgabe nicht erst `true` und dann `false` lauten, da beide `Integer`-Objekt gleichwertig sind? Nein, denn während die *Inhalte* der Objekte übereinstimmen, gehört zu jedem Objekt eine separate Referenz. Die Operatoren `==` und `!=` vergleichen diese Referenzen, so daß die Ausgabe korrekt zuerst `false` und dann `true` lautet. Weniger erfahrene Programmierer sind verständlicherweise zunächst überrascht.

[29] Wie vergleichen Sie die eigentlichen Inhalte zweier Objekte auf Gleichheit? Sie müssen die Methode `equals()` verwenden, die für jedes Objekt definiert ist (nicht aber für Werte primitiven Typs, die sich anstandslos per `==` und `!=` vergleichen lassen). Ein Beispiel:

```
//: operators/EqualsMethod.java
public class EqualsMethod {
    public static void main(String[] args) {
        Integer n1 = new Integer(47);
        Integer n2 = new Integer(47);
        System.out.println(n1.equals(n2));
    }
} /* Output:
      true
   *///:~
```

Dies ist das erwartete Ergebnis. Das inhaltliche Vergleichen mit der `equals()`-Methode hat allerdings einen Haken:

```
//: operators/EqualsMethod2.java
// Default equals() does not compare contents.

class Value {
    int i;
}

public class EqualsMethod2 {
    public static void main(String[] args) {
        Value v1 = new Value();
        Value v2 = new Value();
        v1.i = v2.i = 100;
        System.out.println(v1.equals(v2));
    }
} /* Output:
```

```
    false
*///:~
```

Verwirrenderweise lautet das Ergebnis des *inhaltlichen* Vergleichs wieder `false`. Das Standardverhalten der `equals()`-Methode ist das Vergleichen von Referenzen. Solange Sie die Methoden in Ihrer neuen Klasse nicht *überschreiben*, bekommen Sie nur das Standardverhalten, nicht aber das erwünschte Verhalten. Das Überschreiben von Methoden wird in Kapitel 8 behandelt, das angemessene Definieren einer `equals()`-Methode sogar erst in Kapitel 18. Das Wissen um das tatsächliche Verhalten der `equals()`-Methode kann Ihnen dennoch in der Zwischenzeit einigen Kummer ersparen.

[30] Die meisten Klassen der Standardbibliothek von Java implementieren die `equals()`-Methode, so daß sie die Inhalte von Objekten anstelle der Referenzen vergleicht.

Übungsaufgabe 5: (2) Legen Sie eine Klasse namens `Dog` mit zwei `String`-Feldern `name` und `says` an. Erzeugen Sie in der `main()`-Methode zwei `Dog`-Objekte mit den Namen „Spot“ (bellt „Ruff!“) und „Scruffy“ (bellt „Wurff!“). Zeigen Sie die Namen an und lassen Sie die Hunde bellen. ■

Übungsaufgabe 6: (3) Erzeugen Sie ein neues `Dog`-Objekt in Übungsaufgabe 5 und weisen Sie es der Referenzvariablen von „Spot“ zu. Vergleichen Sie alle Referenzen sowohl per `==` als auch per `equals()`. ■

4.8 Logische Operatoren

[31] Jeder der logischen Operatoren `AND` (`&&`), `OR` (`||`) und `NOT` (`!`) liefert einen booleschen Wert (`true` oder `false`) in Abhängigkeit von der logischen Beziehung zwischen seinen Operanden. Das folgende Beispiel verwendet Vergleichsoperatoren und logische Operatoren:

```
//: operators/Bool.java
// Relational and logical operators.
import java.util.*;
import static net.mindview.util.Print.*;

public class Bool {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt(100);
        int j = rand.nextInt(100);
        print("i = " + i);
        print("j = " + j);
        print("i > j is " + (i > j));
        print("i < j is " + (i < j));
        print("i >= j is " + (i >= j));
        print("i <= j is " + (i <= j));
        print("i == j is " + (i == j));
        print("i != j is " + (i != j));
        // Treating an int as a boolean is not legal Java:
        //! print("i && j is " + (i && j));
        //! print("i || j is " + (i || j));
        //! print("!i is " + !i);
        print("(i < 10) && (j < 10) is "
              + ((i < 10) && (j < 10)) );
        print("(i < 10) || (j < 10) is "
              + ((i < 10) || (j < 10)) );
    }
```

```
} /* Output:
      i = 58
      j = 55
      i > j is true
      i < j is false
      i >= j is true
      i <= j is false
      i == j is false
      i != j is true
      (i < 10) && (j < 10) is false
      (i < 10) || (j < 10) is false
   *///:~
```

Die Operatoren `AND`, `OR` und `NOT` können nur auf boolesche Operanden angewendet werden. Die Verwendung nicht-boolescher Werte als boolesche Werte in logischen Ausdrücken ist im Gegensatz zu C und C++ bei Java nicht erlaubt. Die gescheiterten Versuch sind mit `//!` auskommentiert. (Die Kommentarsyntax ermöglicht das automatische Entfernen von Kommentaren, um das Testen zu erleichtern.) Die folgenden Ausdrücke liefern boolesche Werte, indem sie Vergleichsergebnisse über logische Operatoren verknüpfen.

[32] Beachten Sie, daß ein `boolean`-Wert automatisch in ein passendes `String`-Objekt umwandelt wird, wenn ein `String`-Objekt erwartet wird.

[33] Sie können im obigen Programm den Typ `int` durch jeden primitiven Typ außer `boolean` ersetzen. Beachten Sie, daß der Vergleich von Fließkommazahlen sehr streng ist. Ein Operand der sich auch nur um den kleinstmöglichen Bruchteil von einem anderen Operanden unterscheidet ist nicht äquivalent. Eine Zahl die um den kleinstmöglichen Betrag über Null liegt, ist nicht Null.

Übungsaufgabe 7: (3) Schreiben Sie ein Programm, das einen Münzwurf simuliert. ■

4.8.1 Kurzschlußverhalten

[34] Im Zusammenhang mit logischen Operatoren stoßen Sie im allgemeinen auch auf das Phänomen des „Kurzschlußverhaltens“ bei der Auswertung von Ausdrücken mit Operatoren. Ein Ausdruck wird dabei nur soweit ausgewertet, bis die Wahrheit oder Falschheit des gesamten Ausdrucks zweifelsfrei feststeht. Dadurch wird ein Endstück eines logischen Ausdrucks unter Umständen nicht ausgewertet. Das folgende Beispiel zeigt das Kurzschlußverhalten:

```
//: operators/ShortCircuit.java
// Demonstrates short-circuiting behavior
// with logical operators.
import static net.mindview.util.Print.*;

public class ShortCircuit {
    static boolean test1(int val) {
        print("test1(" + val + ")");
        print("result: " + (val < 1));
        return val < 1;
    }
    static boolean test2(int val) {
        print("test2(" + val + ")");
        print("result: " + (val < 2));
        return val < 2;
    }
    static boolean test3(int val) {
        print("test3(" + val + ")");
```

```
        print("result: " + (val < 3));
        return val < 3;
    }
    public static void main(String[] args) {
        boolean b = test1(0) && test2(2) && test3(2);
        print("expression is " + b);
    }
} /* Output:
      test1(0)
      result: true
      test2(2)
      result: false
      expression is false
   *///:~
```

Jede Testmethode vergleicht ihr Argument mit einer bestimmten Zahl und gibt entweder `true` oder `false` zurück. Außerdem gibt jede Methode eine Textmeldung aus, damit Sie nachvollziehen können, daß die Methode aufgerufen wurde. Die Testergebnisse werden in einen logischen Ausdruck eingesetzt:

```
test1(0) && test2(2) && test3(2)
```

Es liegt nahe, daß alle drei Tests ausgeführt werden, aber die Ausgabe dokumentiert das Gegenteil. Der erste Test liefert das Ergebnis `true` und die Auswertung wird fortgesetzt. Die Auswertung des zweiten Ausdrucks ergibt dagegen `false`. Nachdem der gesamte Ausdruck gezwungermaßen ebenfalls `false` liefert, besteht kein Anlaß mehr, den Rest des Ausdrucks ebenfalls auszuwerten. Die Auswertung des restlichen Ausdrucks kann teuer sein. Das Kurzschlußverhalten soll einen möglichen Performanzgewinn begünstigen, wenn nicht alle Teile eines logischen Ausdrucks ausgewertet werden müssen.

4.9 Literale Werte primitiven Typs

[35] Wenn Sie in einem Programm einen literalen Wert verwenden, „weiß“ der Compiler in der Regel, welchem primitiven Typ dieser Wert entspricht. Es gibt aber auch uneindeutige Fälle. In einer solchen Situation müssen Sie dem Compiler eine Hilfestellung geben, indem Sie den literalen Wert mit einer zusätzlichen Information in Form eines Buchstabens kennzeichnen. Das folgende Beispiel dokumentiert diese Kennzeichnungen:

```
//: operators/Literals.java
import static net.mindview.util.Print.*;

public class Literals {
    public static void main(String[] args) {
        int i1 = 0x2f; // Hexadecimal (lowercase)
         print("i1: " + Integer.toBinaryString(i1));
        int i2 = 0X2F; // Hexadecimal (uppercase)
        print("i2: " + Integer.toBinaryString(i2));
        int i3 = 0177; // Octal (leading zero)
         print("i3: " + Integer.toBinaryString(i3));
        char c = 0xffff; // max char hex value
        print("c: " + Integer.toBinaryString(c));
        byte b = 0x7f; // max byte hex value
        print("b: " + Integer.toBinaryString(b));
        short s = 0x7fff; // max short hex value
        print("s: " + Integer.toBinaryString(s));
```

```
        long n1 = 200L; // long suffix
        long n2 = 200l; // long suffix (but can be confusing)
        long n3 = 200;
        float f1 = 1;
        float f2 = 1F; // float suffix
        float f3 = 1f; // float suffix
        double d1 = 1d; // double suffix
        double d2 = 1D; // double suffix
        // (Hex and Octal also work with long)
    }
} /* Output:
      i1: 101111
      i2: 101111
      i3: 1111111
      c: 111111111111111
      b: 1111111
      s: 111111111111111
   *///:~
```

Ein Buchstabe am Ende eines literalen Wertes gibt dessen primitiven Typ an. Die Buchstaben `L`/`l` stehen für den Typ `long` (wobei ein kleines `l` leicht mit einer `1` verwechselt werden kann). Die Buchstaben `F`/`f` repräsentieren den Typ `float`, die Buchstaben `D`/`d` den Typ `double`.

[36] Felder und lokale Variablen aller ganzzahligen primitiven Typen akzeptieren hexadezimale Werte (Basis 16), die durch die beiden führenden Zeichen `0x`/`0X` gekennzeichnet werden. Der eigentliche Hexadezimalwert schließt sich als Kombination aus den Ziffern 0-9 und den Buchstaben A-F (wahlweise Groß- oder Kleinbuchstaben) an. Beim Versuch ein Feld oder eine lokale Variable, unabhängig von der Basis des Stellenwertsystems, mit einem größeren Wert zu initialisieren, als der jeweilige primitive Typ erlaubt, gibt der Compiler eine Fehlermeldung aus. Beachten Sie die im obigen Beispiel angegebenen Höchstwerte für die primitiven Typen `char`, `byte` und `short`. Wenn Sie einen dieser Höchstwerte überschreiten, erweitert der Compiler den Wert automatisch in den Typ `int` und teilt Ihnen mit, daß Sie in der Zuweisung eine *Typverengung* einsetzen müssen (Typerweiterungen und -verengungen werden in Abschnitt 4.15 behandelt). Sie erfahren es, wenn Sie die Grenzlinie übertreten.

[37] Literale oktale Werte (Basis 8) werden durch eine führende Null gekennzeichnet und bestehen aus Ziffern zwischen 0 und 7.

[38] C, C++ und Java unterstützen keine literale Darstellung binärer Werte. Es ist bei der Arbeit mit hexadezimaler und oktaler Notation aber nützlich, die Ergebnisse in binärem Format darstellen. Zu diesem Zweck definieren die Klassen `Integer` und `Long` die statische Methode `toBinaryString()`. Beachten Sie bei der Übergabe von Werten kleinerer primitiver Typen (das heißt `char`, `byte` und `short`) an `Integer.toBinaryString()`, daß der Typ des Argumentes automatisch in den Typ `int` erweitert wird.

Übungsaufgabe 8: (2) Zeigen Sie, daß die Hexadezimal- und Oktalnotation auch beim primitiven Typ `long` funktioniert. Verwenden Sie die statische `Long`-Methode `toBinaryString()`, um die Ergebnisse auszugeben. ■

4.9.1 Wissenschaftliche Notation (Exponentialschreibweise)

[39] Bei literalen Werten für Exponentialzahlen wird die folgende Schreibweise verwendet:

```
//: operators/Exponents.java
// ''e'' means ''10 to the power.''
```

```
public class Exponents {
    public static void main(String[] args) {
        // Uppercase and lowercase 'e' are the same:
        float expFloat = 1.39e-43f;
        expFloat = 1.39E-43f;
        System.out.println(expFloat);
        double expDouble = 47e47d; // 'd' is optional
        double expDouble2 = 47e47; // Automatically double
        System.out.println(expDouble);
    }
} /* Output:
      1.39E-43
      4.7E48
   *///:~
```

In den Natur- und Ingenieurwissenschaften bezeichnet $e \approx 2.718$ die Basis des natürlichen Logarithmus'. (Die Konstante `Math.E` stellt einen präziseren Wert zur Verfügung.) Das e tritt in diesem Kontext in Exponentialausdrücken wie $1.39 \cdot e^{-43}$ auf, ausgeschrieben $1.39 \cdot 2.718^{-43}$. Die Erfinder der Programmiersprache Fortran haben sich allerdings entschieden, den Kleinbuchstaben `e` in der Bedeutung von „10 hoch ...“ zu verwenden. Diese Entscheidung ist merkwürdig, da Fortran für Naturwissenschaftler und Ingenieure entwickelt wurde und man davon ausgehen möchte, daß die Designer feinfühlig genug sind, um eine solche Zweideutigkeit zu umgehen.[2] Auf jeden Fall hat sich diese Gepflogenheit auch in den Sprachen C, C++ und Java durchgesetzt. Sofern Sie es gewohnt sind, sich unter `e` den natürlichen Logarithmus vorzustellen, müssen Sie eine mentale Transformation vollziehen, wenn Sie in einem Java-Quelltext auf einen Ausdruck wie `1.39e-43` stoßen. Der Ausdruck bedeutet in Wirklichkeit $1.39 \cdot 10^{-43}$.

[40] Beachten Sie, daß die Buchstabenkennzeichung am Ende eines literalen Wertes fortlassen können, wenn der Compiler den passenden Typ selbst ermitteln kann. Beispielsweise besteht bei

```
long n3 = 200;
```

keine Uneindeutigkeit, so daß ein `L` nach der 200 überflüssig wäre. Andererseits behandelt der Compiler literale Werte in wissenschaftlicher Notation als `double`-Werte:

```
float f4 = 1e-43f; // 10 to the power
```

[2] John Kirkham schreibt: „I started computing in 1962 using FORTRAN II on an IBM 1620. At that time, and throughout the 1960s and into the 1970s, FORTRAN was an all uppercase language. This probably started because many of the early input devices were old teletype units that used 5 bit Baudot code, which had no lowercase capability. The 'E' in the exponential notation was also always uppercase and was never confused with the natural logarithm base 'e', which is always lowercase. The 'E' simply stood for exponential, which was for the base of the number system used-usually 10. At the time octal was also widely used by programmers. Although I never saw it used, if I had seen an octal number in exponential notation I would have considered it to be base 8. The first time I remember seeing an exponential using a lowercase 'e' was in the late 1970s and I also found it confusing. The problem arose as lowercase crept into FORTRAN, not at its beginning. We actually had functions to use if you really wanted to use the natural logarithm base, but they were all uppercase.“

Übersetzt etwa: „Ich habe 1962 mit Fortran II auf einem Lochkartenrechner vom Typ IBM 1620 zu programmieren begonnen. Von damals bis in 1970er Jahre hinein, war Fortran eine Sprache in der mit Großbuchstaben programmiert wurde. Die Ursache dafür waren wohl die damaligen Eingabegeräte, nämlich ältere Fernschreiber mit 5 Bit Baudot-Code, ohne Kleinbuchstaben. Auch das „E“ in der Exponentialschreibweise war folglich ein Großbuchstabe und konnte nicht mit dem natürlichen Logarithmus verwechselt werden, der stets mit einem kleinen „e“ bezeichnet wurde. Das große „E“ stand einfach für „hoch ...“ und bezog sich auf die Basis des üblicherweise verwendeten Zahlensystems, das heißt die Zahl 10. Zur damaligen Zeit war auch die Oktalnotation unter den Programmierern verbreitet. Es ist mir zwar nie begegnet, aber hätte ich einen Oktalwert in Exponentialschreibweise gesehen, so hätte ich ihn relativ zur Basis 8 betrachtet. Ich habe zum ersten Mal Ende der 1970er Jahre ein kleines „e“ in der Exponentialschreibweise gesehen und empfand es ebenfalls als verwirrend. Das Uneindeutigkeitsproblem trat erst auf, nachdem sich Kleinbuchstaben in Fortran einzuschleichen begannen, nicht aber am Anfang. Wir mußten sogar Funktionen benutzen, wenn die Basis des natürlichen Logarithmus benötigt wurde, aber diese Funktionen waren durchgängig in Großbuchstaben geschrieben.

Der Compiler würde ohne das `f` eine Fehlermeldung ausgeben, der zufolge Sie den literalen `double`-Wert durch eine Typverengung in den Typ `float` konvertieren müssen.

Übungsaufgabe 9: (1) Zeigen Sie den größten und den kleinsten, in den primitiven Typen `float` und `double` darstellbaren Wert in Exponentionalschreibweise an. ■

4.10 Die bitweisen Operatoren

[41] Die bitweisen Operatoren gestatten das Umschalten einzelner Bits in einem Wert ganzzahligen primitiven Typs. Ein bitweiser Operator erzeugt sein Ergebnis mittels boolescher Algebra auf den korrespondierenden Bits seiner beiden Operanden.

[42] Die bitweisen Operatoren stammen aus der systemnahen Orientierung von C, wo Sie häufig Hardware direkt bedienen und Bits in Hardwareregistern umschalten müssen. Java war ursprünglich zur Einbettung in TV-Digitalempfänger gedacht, so daß die Systemnähe nach wie vor gerechtfertigt ist. Sie werden die bitweisen Operatoren aber wahrscheinlich nicht oft brauchen.

[43] Der bitweise `AND`-Operator (`&`) gibt Eins zurück, wenn seine beiden Operanden Eins sind und sonst Null. Der bitweise `OR`-Operator (`|`) gibt Eins zurück, wenn einer seiner beiden Operanden Eins ist und nur dann Null, wenn beiden Operanden Null sind. Der bitweise `EXCLUSIVE-OR`- oder `XOR`-Operator (`^`) gibt Eins zurück, wenn genau ein Operand Eins ist, nicht aber beide Operanden. Der bitweise `NOT`-Operator (~, auch als *Einerkomplementoperator* bezeichnet) ist ein unärer, das heißt einargumentiger Operator. (Alle anderen bitweisen Operatoren sind binäre, also zweiargumentige Operatoren.) Der bitweise `NOT`-Operator liefert das logische Gegenteil seines Operanden, das heißt eine Eins für den Operanden Null und eine Null für den Operanden Eins.

[44] Da die bitweisen und die logischen Operatoren dieselben Zeichen verwenden, ist es nützlich, eine „Eselsbrücke“ parat zu haben, um sich die jeweiligen Symbole zu merken: Da Bits „klein“ sind, haben die bitweisen Operatoren nur ein Zeichen.

[45] Die bitweisen Operatoren können mit dem Gleichheitszeichen kombiniert werden, um Operation und Zuweisung zusammenzufassen. Zulässig sind: `&=`, `|=` und `^=`. (~ kann als unärer Operator nicht mit dem Zuweisungsoperator kombiniert werden.)

[46] Bitweise Operatoren behandeln Operanden des primitiven Typs `boolean` wie einzelne Bits. Sie können bitweises `AND`, `OR` und `XOR` ausführen, nicht aber bitweises `NOT` (vermutlich, um Verwechslungen mit dem logischen `NOT` vorzubeugen). Die bitweisen Operatoren haben bei Operanden vom Typ `boolean` dieselbe Wirkung wie die logischen Operatoren, abgesehen davon, daß sie kein Kurzschlußverhalten zeigen. Zu den bitweisen Operationen auf Operanden vom Typ `boolean` gehört ein logischer `XOR`-Operator, der nicht unter den logischen Operatoren in Abschnitt 4.8 aufgezählt wurde. Außerdem können Operanden vom Typ `boolean` keinem, der im nächsten Abschnitt beschriebenen Verschiebungsoperatoren übergeben werden.

Übungsaufgabe 10: (3) Schreiben Sie ein Programm mit zwei konstanten Werten, den einen mit alternierenden Nullen und Einsen sowie einer Null im niedrigstwertigen Bit, den anderen ebenfalls mit alternierenden Nullen und Einsen sowie mit einer Eins im niedrigstwertigen Bit. (Tip: Am einfachsten verwenden Sie hexadezimale Konstanten.) Verknüpfen Sie die beiden Werte in jeder möglichen Kombination mit allen bitweisen Operatoren verwenden Sie die statische `Integer`-Methode `toBinaryString()`, um das Ergebnis anzuzeigen. ■

4.11 Die Verschiebungsoperatoren

[47] Die Verschiebungsoperatoren schalten ebenfalls Bits um und können nur auf ganzzahlige Operanden primitiven Typs anwendet werden. Der Linksverschiebungsoperator (`<<`) liefert seinen linken Operanden nach der Linksverschiebung um die durch seinen rechten Operanden definierte Anzahl von Bits, wobei die niederwertigen Bits durch Nullen aufgefüllt werden. Der Rechtsverschiebungsoperator *mit Vorzeichenerweiterung* (`>>`) liefert seinen linken Operanden nach der Rechtsverschiebung um die durch seinen rechten Operanden definierte Anzahl von Bits. Ist der linke Operand positiv, so werden die höherwertigen Bits mit Nullen aufgefüllt. Ist der linke Operand negativ, so werden die höherwertigen Bits mit Einsen aufgefüllt. Der Rechtsverschiebungsoperator *ohne Vorzeichenerweiterung* („Nullerweiterung“, `>>>`) füllt die höherwertigen Bits unabhängig vom Vorzeichen des linken Operanden mit Nullen auf. C und C++ haben keinen solchen Operator.

[48] Ein Operand vom Typ `char`, `byte` oder `short` wird vor der Verschiebung in den Typ `int` erweitert und das Ergebnis ist vom Typ `int`. Die Verschiebung erfaßt nur die fünf niederwertigen Bits auf der rechten Seite, um eine Verschiebung um mehr als 32 Stellen zu verhindern (die Länge des Typs `int`). Die Verschiebung eines Operanden vom Typ `long` liefert ein Ergebnis vom Typ `long`. Die Verschiebung erfaßt nur die sechs niederwertigen Bits auf der rechten Seite, um eine Verschiebung um mehr als 64 Stellen zu verhindern (die Länge des Typs `long`).

[49] Die Verschiebungsoperatoren können mit dem Zuweisungsoperator kombiniert werden (`<<=`, `>>=` oder `>>>=`). Der `lvalue` wird dabei durch den `lvalue`, verschoben um den `rvalue` ersetzt. Die Rechtsverschiebung ohne Vorzeichenerweiterung liefert bei Kombination mit dem Zuweisungsoperator für Operanden vom Typ `byte` oder `short` aber falsche Ergebnisse. Der Wert des Operanden wird in den Typ `int` erweitert, rechtsverschoben und vor der Rückzuweisung an den ursprünglichen Operanden abgeschnitten, so daß Sie in diesen Fällen den Wert -1 bekommen. Das folgende Beispiel zeigt dieses Problem:

```
//: operators/URShift.java
// Test of unsigned right shift.
import static net.mindview.util.Print.*;

public class URShift {
    public static void main(String[] args) {
        int i = -1;
        print(Integer.toBinaryString(i));
        i >>>= 10;
        print(Integer.toBinaryString(i));
        long l = -1;
        print(Long.toBinaryString(l));
        l >>>= 10;
        print(Long.toBinaryString(l));
        short s = -1;
        print(Integer.toBinaryString(s));
        s >>>= 10;
        print(Integer.toBinaryString(s));
        byte b = -1;
        print(Integer.toBinaryString(b));
        b >>>= 10;
        print(Integer.toBinaryString(b));
        b = -1;
        print(Integer.toBinaryString(b));
        print(Integer.toBinaryString(b>>>10));
    }
} /* Output:
```

```
    11111111111111111111111111111111
    11111111111111111111111
    1111111111111111111111111111111111111111111111111111111111111111111111
    11111111111111111111111111111111111111111111111111111111
    11111111111111111111111111111111
    11111111111111111111111111111111
    11111111111111111111111111111111
    11111111111111111111111111111111
    11111111111111111111111111111111
    11111111111111111111111
  *///:~
```

Das Ergebnis der letzten Verschiebung wird direkt ausgegeben und nicht in `b` zurück gespeichert, so daß sich das richtige Verhalten zeigt.

[50–51] Das nächste Beispiel zeigt die Anwendung aller bitorientierten Operatoren:

```
//: operators/BitManipulation.java
// Using the bitwise operators.
import java.util.*;
import static net.mindview.util.Print.*;

public class BitManipulation {
    public static void main(String[] args) {
        Random rand = new Random(47);
        int i = rand.nextInt();
        int j = rand.nextInt();
        printBinaryInt("-1", -1);
        printBinaryInt("+1", +1);
        int maxpos = 2147483647;
        printBinaryInt("maxpos", maxpos);
        int maxneg = -2147483648;
        printBinaryInt("maxneg", maxneg);
        printBinaryInt("i", i);
        printBinaryInt("~i", ~i);
        printBinaryInt("-i", -i);
        printBinaryInt("j", j);
        printBinaryInt("i & j", i & j);
        printBinaryInt("i | j", i | j);
        printBinaryInt("i ^ j", i ^ j);
        printBinaryInt("i << 5", i << 5);
        printBinaryInt("i >> 5", i >> 5);
        printBinaryInt("(~i) >> 5", (~i) >> 5);
        printBinaryInt("i >>> 5", i >>> 5);
        printBinaryInt("(~i) >>> 5", (~i) >>> 5);

        long l = rand.nextLong();
        long m = rand.nextLong();
        printBinaryLong("-1L", -1L);
        printBinaryLong("+1L", +1L);
        long ll = 9223372036854775807L;
        printBinaryLong("maxpos", ll);
        long lln = -9223372036854775808L;
        printBinaryLong("maxneg", lln);
        printBinaryLong("l", l);
        printBinaryLong("~l", ~l);
        printBinaryLong("-l", -l);
        printBinaryLong("m", m);
        printBinaryLong("l & m", l & m);
```

```
        printBinaryLong(''l | m'', l | m);
        printBinaryLong(''l ^ m'', l ^ m);
        printBinaryLong(''l << 5'', l << 5);
        printBinaryLong(''l >> 5'', l >> 5);
        printBinaryLong(''(~l) >> 5'', (~l) >> 5);
        printBinaryLong(''l >>> 5'', l >>> 5);
        printBinaryLong(''(~l) >>> 5'', (~l) >>> 5);
    }
    static void printBinaryInt(String s, int i) {
        print(s + '', int: '' + i + '', binary:\n '' + Integer.toBinaryString(i));
    }
    static void printBinaryLong(String s, long l) {
        print(s + '', long: '' + l + '', binary:\n '' + Long.toBinaryString(l));
    }
} /* Output:
      -1, int: -1, binary:
      11111111111111111111111111111111
      +1, int: 1, binary:
      1
      maxpos, int: 2147483647, binary:
      1111111111111111111111111111111
      maxneg, int: -2147483648, binary:
      10000000000000000000000000000000
      i, int: -1172028779, binary:
      10111010001001000100001010010101
      ~i, int: 1172028778, binary:
      1000101110110111011110101101010
      -i, int: 1172028779, binary:
      1000101110110111011110101101011
      j, int: 1717241110, binary:
      1100110010110110000010100010110
      i & j, int: 570425364, binary:
      100010000000000000000000010100
      i | j, int: -25213033, binary:
      11111100111111101000111110010111
      i ^ j, int: -595638397, binary:
      11011100011111101000111100000011
      i << 5, int: 1149784736, binary:
      1000100100010000101001010100000
      i >> 5, int: -36625900, binary:
      11111101110100010010001000010100
      (~i) >> 5, int: 36625899, binary:
      1000101110110111011101011
      i >>> 5, int: 97591828, binary:
      1011101000100100010000010100
      (~i) >>> 5, int: 36625899, binary:
      1000101110110111011101011
      ...
  *////:~
```

Die beiden Methoden `printBinaryInt()` und `printBinaryLong()` gegen Ende des Quelltextes erwarten ein Argument vom Typ `int` beziehungsweise `long` und geben es nach einem kurzen Beschreibungstext einmal als Dezimal- und einmal als Dualwert aus. Das Beispiel demonstriert nicht nur die Wirkung aller bitweisen Operatoren bei Operanden vom Typ `int` beziehungsweise `long`, sondern auch die dezimalen und dualen Minimal- und Maximalwerte der primitiven Typ `int` und `long` sowie die Werte +1 und -1. Das höchstwertige Bit gibt das Vorzeichen an: Eine Null bedeutet „positiv“, eine Eins „negativ“. Die Ausgabe der `int`-Hälfte des Beispiels ist oben angegeben. Die

binäre Darstellung der Zahlen wird als vorzeichenbehaftetes Zweierkomplement bezeichnet.

Übungsaufgabe 11: (3) Beginnen Sie mit einer Dualzahl, deren höchstwertige Stelle eine Eins enthält (Tip: Verwenden Sie eine hexadezimale Konstante). Verwenden Sie den Rechtsverschiebungsoperator mit Vorzeichenerweiterung, um diese Dualzahl nacheinander durch sämtliche binären Positionen zu verschieben und geben Sie das Ergebnis jedes Schrittes mit Hilfe der statischen `Integer`-Methode `toBinaryString()` aus. ■

Übungsaufgabe 12: (3) Beginnen Sie mit einer Dualzahl die durchgängig aus Einsen besteht und verschieben Sie die Dualzahl nach links. Verwenden Sie anschließend den Rechtsverschiebungsoperator ohne Vorzeichenerweiterung, um die Dualzahl nacheinander durch sämtliche binären Positionen zu verschieben und geben Sie das Ergebnis jedes Schrittes mit Hilfe der statischen `Integer`-Methode `toBinaryString()` aus. ■

Übungsaufgabe 13: (3) Schreiben Sie eine Methode, die `char`-Werte als Dualzahlen ausgibt. Führen Sie die Funktionsweise der Methode an mehreren verschiedenen Zeichen vor. ■

4.12 Der ternäre Operator (konditionaler Operator)

[52] Der *ternäre Operator* oder auch *konditionale Operator* ist dadurch ungewöhnlich, daß er drei Operanden hat. Der ternäre Operator liefert, im Gegensatz zur gewöhnlichen `if`/`else`-Kombination, die Sie in Abschnitt 5.2 kennenlernen, einen Wert und ist somit tatsächlich ein echter Operator. Ein Ausdruck, der den ternären Operator enthält, hat das Format:

```
boolean-exp ? value0 : value1
```

Ergibt die Auswertung des booleschen Ausdrucks `boolean-exp` den Wert `true`, so wird der erste Teilausdruck `value0` ausgewertet und sein Ergebnis als Rückgabewert des ternären Operators übergeben. Liefert der boolesche Ausdruck dagegen `false`, so wird der zweite Teilausdruck `value1` ausgewertet und sein Ergebnis als Rückgabewert geliefert.

[53] Natürlich können Sie anstelle des ternären Operators auch eine gewöhnliche `if`/`else`-Kombination wählen, der Operator ist aber syntaktisch viel knapper. Obwohl sich C (woher dieser Operator stammt) rühmt, eine knappe Sprache zu sein und der ternäre Operator eventuell zum Teil aus Effizienzgründen eingeführt wurde, sollten Sie ihn nur mit Bedacht einsetzen. Sie schreiben sonst mühelos unleserlichen Quelltext.

[54] Der ternäre Operator unterscheidet sich von einer `if`/`else`-Kombination dadurch, daß er einen Wert liefert. Das folgende Beispiel vergleicht beide Ansätze:

```
//: operators/TernaryIfElse.java
import static net.mindview.util.Print.*;

public class TernaryIfElse {
    static int ternary(int i) {
        return i < 10 ? i * 100 : i * 10;
    }
    static int standardIfElse(int i) {
        if(i < 10)
            return i * 100;
        else
            return i * 10;
    }
    public static void main(String[] args) {
        print(ternary(9));
```

```
        print(ternary(10));
        print(standardIfElse(9));
        print(standardIfElse(10));
    }
} /* Output:
      900
      100
      900
      100
   *///:~
```

Die Implementierung der `ternary()`-Methode ist offensichtlich viel kompakter als die Lösung ohne ternären Operator in der `standardIfElse()`-Methode. Andererseits ist `standardIfElse()` leichter zu verstehen und der Mehraufwand beim Tippen hält sich in Grenzen. Eine Hilfestellung beim Abwägen, ob Sie den ternären Operator verwenden sollen: Der Einsatz des ternären Operators ist im allgemeinen berechtigt, wenn Sie ein Feld oder eine lokale Variable mit einem von zwei Werten initialisieren müssen.

4.13 Die Konkatenationsoperatoren + und += für String-Objekte

[55] Zwei Operatoren von Java haben eine zweite Funktionsweise: Die Operatoren `+` und `+=` können zur Verknüpfung („Konkatenation“) von `String`-Objekten verwendet werden, wie Sie, zum Beispiel auf Seite 79 bereits gesehen haben. Dieser Verwendungszweck wirkt völlig natürlich, auch wenn er sich nicht in den traditionellen Kontext dieser beiden Operatoren einfügt.

[56] Dieses variable Verhalten von Operatoren schien den Designern von C++ eine gute Idee zu sein, so daß die *Operatorüberladung* in C++ implementiert wurde, um den Programmierern zu ermöglichen, nahezu jeden Operator mit zusätzlichem Verhalten auszustatten. Leider hat sich die Operatorüberladung, kombiniert mit einigen Einschränkungen von C++, beim Entwickeln eigener Klassen, als recht komplizierte Eigenschaft herausgestellt. Obwohl sich Operatorüberladung bei Java viel einfacher implementieren ließe als bei C++ (wie die Sprache C# mit ihrer einfachen Operatorüberladung zeigt), galt diese Spracheigenschaft nach wie vor als zu kompliziert, so daß Java-Programmierer, im Gegensatz zu Ihren Kollegen bei C++ und C#, Operatoren nicht selbst überladen können.

[57] Bei Anwendung der `String`-Operatoren `+` und `+=` zeigt sich ein weiteres interessantes Verhalten. Beginnt nämlich ein Ausdruck mit einem `String`-Objekt, so müssen alle folgenden Operanden ebenfalls `String`-Objekte sein (denken Sie daran, daß der Compiler jede Zeichenkette zwischen doppelten Anführungszeichen in ein `String`-Objekt umwandelt):

```
//: operators/StringOperators.java
import static net.mindview.util.Print.*;

public class StringOperators {
    public static void main(String[] args) {
        int x = 0, y = 1, z = 2;
        String s = "x, y, z ";
        print(s + x + y + z);
        print(x + " " + s); // Converts x to a String
        s += "(summed) = "; // Concatenation operator
        print(s + (x + y + z));
        print("" + x); // Shorthand for Integer.toString()
    }
} /* Output:
```

```
    x, y, z 012
    0 x, y, z
    x, y, z (summed) = 3
    0
*///:~
```

Beachten Sie, daß die erste `print()`-Anweisung „012“ und nicht „3“ ausgibt, also die Summe der drei ganzen Zahlen. Der Java-Compiler wandelt die Inhalte der lokalen Variablen `x`, `y` und `z` in äquivalente `String`-Objekte um und verknüpft diese `String`-Objekte miteinander, statt sie zuvor zu addieren. Die zweite `print()`-Anweisung wandelt den Inhalt der führenden lokalen Variablen `x` in ein `String`-Objekt um, das heißt die Umwandlung eines Ausdrucks in ein `String`-Objekt hängt nicht davon ab, ob der der Ausdruck mit einem `String`-Objekt beginnt oder nicht. Abschließend sehen Sie ein Anwendungsbeispiel für den Konkatenationsoperator `+=`, um eine literale Zeichenkette an das von `s` referenzierte `String`-Objekt anzufügen sowie die Verwendung von Klammern, um die Auswertungsreihenfolge des Ausdrucks festzulegen, damit die `int`-Werte vor ihrer Ausgabe tatsächlich addiert werden.

[58] Beachten Sie die letzte `print()`-Anweisung in der `main()`-Methode: Hin und wieder stoßen Sie auf die literale leere Zeichenkette ’’’’, gefolgt von einem `+`-Operator und einem Wert primitiven Typs, um die Umwandlung in ein `String`-Objekt ohne den expliziten und sperrigen Aufruf einer zu bewerkstelligen (in diesem Fall die `Integer`-Methode `toString()`).

4.14 Häufige Fallen beim Anwenden von Operatoren

[59] Ein Fehler im Umgang mit Operatoren ist das Fortlassen von Klammern, wenn Sie auch nur um ein weniges unsicher sind, wie die Auswertung des Ausdrucks vor sich geht. Dies gilt auch bei Java.

[60] Ein sehr häufiger Fehler bei C und C++ ist:

```
while (x = y) {
    // ...
}
```

Der Programmierer wollte zweifellos auf Gleichheit testen (`==`), statt eine Zuweisung zu vollziehen. Bei C und C++ liefert diese Zuweisung stets `true`, wenn `y` von Null verschiedenen ist, so daß Sie wahrscheinlich eine unendliche Schleife bekommen. Bei Java gibt diese Zuweisung keinen Wert vom Typ `boolean` zurück. Andererseits erwartet der Compiler aber einen booleschen Ausdruck und nimmt keine Konvertierung von `int` nach `boolean` vor. Stattdessen erhalten Sie zur Übersetzungszeit eine Fehlermeldung und fangen das Problem bereits ab, bevor Sie überhaupt versuchen können, das Programm aufzurufen. Diese Falle droht also bei Java nicht. (Die Fehlermeldung des Compilers bleibt nur in einem einzigen Fall aus, nämlich wenn `x` und `y` vom Typ `boolean` sind. In diesem Fall ist `x = y` eine gültige Zuweisung und stellt im obigen Beispiel voraussichtlich einen Programmierfehler dar.)

[61] Die Wahl der bitweisen `AND`- und `OR`-Operatoren anstelle der logischen Operatoren ist bei C und C++ ein ähnliches Problem. Die bitweisen Operatoren bestehen aus einem Zeichen (`&`, `|`), die logischen Operatoren aus zwei Zeichen (`&&`, `||`). Wie bei `=` und `==` schreibt man leicht ein Zeichen statt der erforderlichen zwei. Wiederum verhindert der Java-Compiler den Mißbrauch, indem er die Verwendung eines anderen als des erwarteten primitiven Typs nicht erlaubt.

4.15 Typerweiterung und -verengung bei Werten primitiven Typs

[62] Die Begriffe *Typerweiterung* und *Typverengung* werden im Sinne von „umformen“ oder „in eine Form gießen“ gebraucht. Java ändert den primitiven Typ des Inhaltes eines Feldes oder einer lokalen Variablen in entsprechenden Fällen automatisch. Wenn Sie beispielsweise einem Feld vom Typ `float` einen Wert vom Typ `int` zuweisen, konvertiert der Compiler den `int`-Wert automatisch in einen `float`-Wert. Die explizite Angabe einer Typerweiterung oder -verengung kennzeichnet eine Konvertierung oder erzwingt eine Konvertierung, die normalerweise nicht vollzogen werden würde.

[63] Eine Typerweiterung oder -verengung besteht aus dem eingeklammerten Namen des gewünschten Typs und steht links neben dem zu konvertierenden Wert:

```
//: operators/Casting.java
public class Casting {
    public static void main(String[] args) {
        int i = 200;
        long lng = (long)i;
        lng = i; // ''Widening,'' so cast not really required
        long lng2 = (long)200;
        lng2 = 200;
        // A ''narrowing conversion'':
        i = (int)lng2; // Cast required
    }
} ///:~
```

Eine Typerweiterung oder -vereinung kann sich sowohl auf einen literalen Wert als auch auf ein Feld oder eine lokale Variable beziehen. Beachten Sie die nicht benötigten Typerweiterungen: Der Compiler erweitert zum Beispiel einen `int`-Wert bei Bedarf automatisch in einen `long`-Wert. Explizite Typerweiterung sind dennoch zulässig, sofern Sie Wert darauf legen oder etwas verdeutlichen möchten. Es gibt andererseits Situationen, in denen eine Typ*verengung* erforderlich ist, damit sich der Quelltext übersetzen läßt.

[64] Typkonvertierungen sind bei C und C++ nicht unproblematisch. Abgesehen von *Typverengungen*, das heißt beim Übergang von einem Datentyp der mehr Informationen speichert zu einem Typ mit geringerer Kapazität, so daß Informationsverluste auftreten können, sind Typkonvertierungen bei Java sicher. Der Compiler erzwingt in solchen Fällen eine explizite Bestätigung für die Typverengung, da die Konvertierung „gefährlich“ ist. Bei einer Typerweiterung ist dagegen keine explizite Syntax erforderlich, da der neue Typ mehr Informationen speichern kann als der alte, so daß Informationsverluste ausgeschlossen sind.

[65] Mit Ausnahme des Typs `boolean`, der keinerlei Konvertierung zuläßt, kann jeder primitive Typ zu jedem anderen primitiven Typ erweitert beziehungsweise eingeengt werden. Klassen *selbst*, gestatten keine Typerweiterung oder -verengung. Die „Konvertierung“ einer Klasse in eine andere erfordert spezielle Verfahren, wie beispielsweise Ableitung oder Komposition. (Sie lernen im Laufe dieses Buches, daß der Typ eines Objektes oder präziser der Typ einer Referenz auf ein Objekt innerhalb einer Familie von Typen umgewandelt werden kann. Beispielsweise kann eine Referenz auf ein `Oak`-Objekt in den „verwandten“ Typ `Tree` umwandelt werden, nicht aber in den “fremden“ Typ `Rock`.)

4.15.1 Abschneiden und Runden

[66] Bei Typverengung müssen Sie auf das Abschneiden und Runden des Operanden achten. Wie geht Java vor, wenn Sie zum Beispiel einen Wert vom Typ `float` in einen ganzzahligen Typ verengen?

Wenn Sie etwa einem Feld oder einer lokalen Variablen vom Typ `int` den Wert 29.7 zuweisen, lautet das Ergebnis dann 29 oder 30? Das folgende Beispiel liefert die Antwort:

```
//: operators/CastingNumbers.java
// What happens when you cast a float
// or double to an integral value?
import static net.mindview.util.Print.*;

public class CastingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print(''(int)above: '' + (int)above);
        print(''(int)below: '' + (int)below);
        print(''(int)fabove: '' + (int)fabove);
        print(''(int)fbelow: '' + (int)fbelow);
    }
} /* Output:
      (int)above: 0
      (int)below: 0
      (int)fabove: 0
      (int)fbelow: 0
   *///:~
```

Beim der Verengung eines Wertes vom Typ `float` oder `double` in den Typ `int` wird der Wert stets abgeschnitten. Wenn Sie ein gerundetes Ergebnis brauchen, können Sie die `round()`-Methode der Klasse `java.lang.Math` verwenden:

```
//: operators/RoundingNumbers.java
// Rounding floats and doubles.
import static net.mindview.util.Print.*;

public class RoundingNumbers {
    public static void main(String[] args) {
        double above = 0.7, below = 0.4;
        float fabove = 0.7f, fbelow = 0.4f;
        print(''Math.round(above): '' + Math.round(above));
        print(''Math.round(below): '' + Math.round(below));
        print(''Math.round(fabove): '' + Math.round(fabove));
        print(''Math.round(fbelow): '' + Math.round(fbelow));
    }
} /* Output:
      Math.round(above): 1
      Math.round(below): 0
      Math.round(fabove): 1
      Math.round(fbelow): 0
   *///:~
```

Da die Klasse `Math` zum Package `java.lang` gehört, ist keine `import`-Anweisung erforderlich.

4.15.2 Typerweiterung

[67] Wenn Sie eine mathematische oder bitweise Operation auf Operanden eines kleineren primitiven Typs als `int` ausführen (also `char`, `byte` oder `short`), werden Sie entdecken, daß die Operanden vor der Operation in `int`-Werte erweitert werden und das Ergebnis der Operation ein `int`-Wert ist. Beim Rückzuweisen des Ergebnisses an den kleineren Typ ist somit eine Typverengung erforderlich, bei der ein Informationsverlust eintreten kann. Im allgemeinen bestimmt der größte Datentyp eines

Ausdrucks den Ergebnistyp des Ausdrucks: Die Multiplikation von `float` und `double` liefert einen `double`-Wert, die Addition von `int` und `long` einen `long`-Wert.

4.16 Java hat keinen sizeof-Operator

[68] Der `sizeof()`-Operator von C und C++ gibt an, wieviele Bytes für eine Datenstruktur allokiert wurden. Der wichtigste Grund für diesen Operator in C und C++ ist die Portabilität. Datentypen können auf unterschiedlichen Maschinen verschieden groß sein, so daß der Programmierer die Größe dieser Typen ermitteln muß, wenn das Programm typgrößenabhängige Operationen beinhaltet. Beispielsweise kann ein Rechner einen `int`-Wert in 32 Bit speichern, ein anderer in 16 Bit. Ein Programm kann auf dem ersten Rechner größere `int`-Werte verarbeiten. Sie können sich vorstellen, daß die Portabilität für C- und C++-Entwickler ein erhebliches Problem ist.

[69] Java braucht keinen `sizeof()`-Operator, da alle Datentypen auf allen Maschinen gleich groß sind. Sie brauchen sich auf dieser Ebene keine Gedanken über Portabilität machen, da sie bereits im Design der Sprache enthalten ist.

4.17 Ein Kompendium von Java-Operatoren

[70] Das folgende Beispiel zeigt, welche primitiven Typen mit bestimmten Operatoren kombiniert werden können. Im Grunde genommen wird, mit verschiedenen Datentypen, immer dasselbe wiederholt. Die Datei läßt sich ohne Fehlermeldungen übersetzen, da alle problematischen Zeilen per `//!` auskommentiert sind:

```
//: operators/AllOps.java
// Tests all the operators on all the primitive data types
// to show which ones are accepted by the Java compiler.

public class AllOps {
    // To accept the results of a boolean test:
    void f(boolean b) {}
    void boolTest(boolean x, boolean y) {
        // Arithmetic operators:
        //! x = x * y;
        //! x = x / y;
        //! x = x % y;
        //! x = x + y;
        //! x = x - y;
        //! x++;
        //! x--;
        //! x = +y;
        //! x = -y;
        // Relational and logical:
        //! f(x > y);
        //! f(x >= y);
        //! f(x < y);
        //! f(x <= y);
        f(x == y);
        f(x != y);
        f(!y);
        x = x && y;
        x = x || y;
        // Bitwise operators:
```

```
        //! x = ~y;
        x = x & y;
        x = x | y;
        x = x ^ y;
        //! x = x << 1;
        //! x = x >> 1;
        //! x = x >>> 1;
        // Compound assignment:
        //! x += y;
        //! x -= y;
        //! x *= y;
        //! x /= y;
        //! x %= y;
        //! x <<= 1;
        //! x >>= 1;
        //! x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // Casting:
        //! char c = (char)x;
        //! byte b = (byte)x;
        //! short s = (short)x;
        //! int i = (int)x;
        //! long l = (long)x;
        //! float f = (float)x;
        //! double d = (double)x;
    }
    void charTest(char x, char y) {
        // Arithmetic operators:
        x = (char)(x * y);
        x = (char)(x / y);
        x = (char)(x % y);
        x = (char)(x + y);
        x = (char)(x - y);
        x++;
        x--;
        x = (char)+y;
        x = (char)-y;
        // Relational and logical:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //! f(!x);
        //! f(x && y);
        //! f(x || y);
        // Bitwise operators:
        x= (char)~y;
        x = (char)(x & y);
        x = (char)(x | y);
        x = (char)(x ^ y);
        x = (char)(x << 1);
        x = (char)(x >> 1);
        x = (char)(x >>> 1);
```

```
        // Compound assignment:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // Casting:
        //! boolean bl = (boolean)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
        double d = (double)x;
    }
    void byteTest(byte x, byte y) {
        // Arithmetic operators:
        x = (byte)(x* y);
        x = (byte)(x / y);
        x = (byte)(x % y);
        x = (byte)(x + y);
        x = (byte)(x - y);
        x++;
        x--;
        x = (byte)+ y;
        x = (byte)- y;
        // Relational and logical:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //! f(!x);
        //! f(x && y);
        //! f(x || y);
        // Bitwise operators:
        x = (byte)~y;
        x = (byte)(x & y);
        x = (byte)(x | y);
        x = (byte)(x ^ y);
        x = (byte)(x << 1);
        x = (byte)(x >> 1);
        x = (byte)(x >>> 1);
        // Compound assignment:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
```

```
        x >>= 1;
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // Casting:
        //! boolean bl = (boolean)x;
        char c = (char)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
        double d = (double)x;
    }
    void shortTest(short x, short y) {
        // Arithmetic operators:
        x = (short)(x * y);
        x = (short)(x / y);
        x = (short)(x % y);
        x = (short)(x + y);
        x = (short)(x - y);
        x++;
        x--;
        x = (short)+y;
        x = (short)-y;
        // Relational and logical:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //! f(!x);
        //! f(x && y);
        //! f(x || y);
        // Bitwise operators:
        x = (short)~y;
        x = (short)(x & y);
        x = (short)(x | y);
        x = (short)(x ^ y);
        x = (short)(x << 1);
        x = (short)(x >> 1);
        x = (short)(x >>> 1);
        // Compound assignment:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        x <<= 1;
        x >>= 1;
        x >>>= 1;
        x &= y;
        x ^= y;
        x |= y;
        // Casting:
        //! boolean bl = (boolean)x;
```

```
    char c = (char)x;
    byte b = (byte)x;
    int i = (int)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
void intTest(int x, int y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    long l = (long)x;
    float f = (float)x;
    double d = (double)x;
}
```

```
void longTest(long x, long y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
    x++;
    x--;
    x = +y;
    x = -y;
    // Relational and logical:
    f(x > y);
    f(x >= y);
    f(x < y);
    f(x <= y);
    f(x == y);
    f(x != y);
    //! f(!x);
    //! f(x && y);
    //! f(x || y);
    // Bitwise operators:
    x = ~y;
    x = x & y;
    x = x | y;
    x = x ^ y;
    x = x << 1;
    x = x >> 1;
    x = x >>> 1;
    // Compound assignment:
    x += y;
    x -= y;
    x *= y;
    x /= y;
    x %= y;
    x <<= 1;
    x >>= 1;
    x >>>= 1;
    x &= y;
    x ^= y;
    x |= y;
    // Casting:
    //! boolean bl = (boolean)x;
    char c = (char)x;
    byte b = (byte)x;
    short s = (short)x;
    int i = (int)x;
    float f = (float)x;
    double d = (double)x;
}
void floatTest(float x, float y) {
    // Arithmetic operators:
    x = x * y;
    x = x / y;
    x = x % y;
    x = x + y;
    x = x - y;
```

```
        x++;
        x--;
        x = +y;
        x = -y;
        // Relational and logical:
        f(x > y);
        f(x >= y);
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //! f(!x);
        //! f(x && y);
        //! f(x || y);
        // Bitwise operators:
        //! x = ~y;
        //! x = x & y;
        //! x = x | y;
        //! x = x ^ y;
        //! x = x << 1;
        //! x = x >> 1;
        //! x = x >>> 1;
        // Compound assignment:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        //! x <<= 1;
        //! x >>= 1;
        //! x >>>= 1;
        //! x &= y;
        //! x ^= y;
        //! x |= y;
        // Casting:
        //! boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        double d = (double)x;
    }
    void doubleTest(double x, double y) {
        // Arithmetic operators:
        x = x * y;
        x = x / y;
        x = x % y;
        x = x + y;
        x = x - y;
        x++;
        x--;
        x = +y;
        x = -y;
        // Relational and logical:
        f(x > y);
        f(x >= y);
```

```
        f(x < y);
        f(x <= y);
        f(x == y);
        f(x != y);
        //! f(!x);
        //! f(x && y);
        //! f(x || y);
        // Bitwise operators:
        //! x = ~y;
        //! x = x & y;
        //! x = x | y;
        //! x = x ^ y;
        //! x = x << 1;
        //! x = x >> 1;
        //! x = x >>> 1;
        // Compound assignment:
        x += y;
        x -= y;
        x *= y;
        x /= y;
        x %= y;
        //! x <<= 1;
        //! x >>= 1;
        //! x >>>= 1;
        //! x &= y;
        //! x ^= y;
        //! x |= y;
        // Casting:
        //! boolean bl = (boolean)x;
        char c = (char)x;
        byte b = (byte)x;
        short s = (short)x;
        int i = (int)x;
        long l = (long)x;
        float f = (float)x;
    }
} ///:~
```

Beachten Sie die recht begrenzen Verwendungsmöglichkeiten für den Typ `boolean`. Sie können einem Feld oder einer lokalen Variablen vom Typ `boolean` entweder `true` oder `false` zuweisen und es auf Wahrheit oder Falschheit testen, `boolean`-Werte aber nicht addieren oder eine andere Operation darauf ausführen.

[71] An den Typen `char`, `byte` und `short` können Sie den Effekt der Typerweiterung bei arithmetischen Operatoren beobachten. Jede arithmetische Operation mit Operanden vom Typ `char`, `byte` oder `short`, liefert ein Ergebnis vom Typ `int`, welches vor seiner Rückzuweisung explizit in den ursprünglichen Typ konvertiert werden muß (durch eine Typverengung mit eventuellem Informationsverlust). Bei Operanden vom Typ `int` ist dagegen keine Typerweiterung erforderlich. Geben Sie sich dennoch keinesfalls der Illusion hin, alles sei sicher. Wenn Sie beispielsweise zwei genügend große `int`-Werte multiplizieren, erzeugen Sie einen Überlauf (*overflow*), wie das folgende Beispiel zeigt:

```
//: operators/Overflow.java
// Surprise! Java lets you overflow.

public class Overflow {
    public static void main(String[] args) {
```

```
        int big = Integer.MAX_VALUE;
        System.out.println("big = " + big);
        int bigger = big * 4;
        System.out.println("bigger = " + bigger);
    }
} /* Output:
      big = 2147483647
      bigger = -4
   *///:~
```

Der Compiler gibt weder eine Warnung noch eine Fehlermeldung aus und zur Laufzeit wird keine Ausnahme ausgeworfen. Java ist gut, aber nicht *so* gut.

[72] Zusammengesetzte Zuweisungen verlangen bei Operanden vom Typ `char`, `byte` und `short` keine explizite Typverengung, obwohl sie Typerweiterungen mit denselben Ergebnissen veranlassen, wie bei den direkten arithmetischen Operationen. Andererseits vereinfacht die nicht benötigte explizite Typverengung den Quelltext.

[73] Mit Ausnahme von `boolean` kann jeder primitive Typ in jeden anderen primitiven Typ konvertiert werden. Beachten Sie die Auswirkungen einer Typverengung in einen kleineren Typ, da Sie andernfalls während der Konvertierung unbemerkt Informationen verlieren können.

Übungsaufgabe 14: (3) Schreiben Sie eine Methode, die zwei Argumente vom Typ `String` erwartet. Wenden Sie alle `boolean`-wertigen Vergleiche auf die beiden Argument an und geben Sie die Ergebnisse aus. Wenden Sie zusätzlich zu `==` und `!=` auch die `equals()`-Methode an. Rufen Sie Ihre Methode aus der `main()`-Methode mit einigen unterschiedlichen `String`-Objekten auf. ■

4.18 Zusammenfassung

[74] Wenn Sie bereits Erfahrung in einer Programmiersprache mit C-ähnlicher Syntax haben, gibt es angesichts der Ähnlichkeit mit den Java-Operatoren praktisch nichts neues zu lernen. Falls Sie aber Schwierigkeiten mit diesem Kapitel hatten, sollten Sie das Multimediaseminar *Thinking in C* durchgehen, das Sie unter der Webadresse *http://www.mindview.net* kostenlos herunterladen können.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 5

Steuerung des Programmablaufs

Inhaltsübersicht

[0] Ein Programm beinflußt während seiner Ausführung seine Umgebung und trifft Entscheidungen, wie ein fühlendes Wesen. Bei Java werden Entscheidungsmöglichkeiten mit Hilfe von *Ausführungskontrollanweisungen* implementiert.

[1] Java verwendet sämtliche Ausführungskontrollanweisungen von C. Wenn Sie also bereits mit C oder C++ programmiert haben, wird Ihnen der Inhalt dieses Kapitels größtenteils bereits vertraut sein. Die meisten prozeduralen Programmiersprachen haben solche Anweisungen, die sich namentlich oder funktionell häufig überlappen. Java kennt die Schlüsselworte `if/else`, `while`, `do/while`, `for`, `return`, `break` sowie `switch` (für Auswahlanweisungen). Das so häufig verschmähte Schlüsselwort `goto` wird von Java nicht unterstützt (obwohl es für bestimmte Problemtypen noch immer die am besten geeignete Lösung sein kann). Sie können ein `goto`-ähnliches Sprungverhalten implementieren, welches allerdings stärkeren Beschränkungen unterliegt, als eine typische `goto`-Anweisung.

5.1 Die booleschen Werte true und false

[2] Alle *bedingenden Anweisungen* werten die Wahrheit beziehungsweise Falschheit eines *boolesches Ausdrucks* aus, um den Ausführungspfad zu bestimmen. Der Ausdruck `a == b` verwendet zum Beispiel den Vergleichsoperator `==`, um zu ermitteln, ob die Werte von `a` und `b` identisch sind. Die Auswertung eines booleschen Ausdrucks ergibt entweder `true` oder `false`. Sie können alle

Vergleichsoperatoren aus dem vorigen Kapitel verwenden, um einen booleschen Ausdruck zu konstruieren. Während bei C und C++ jeder von Null verschiedene Wert als wahr und die Null selbst als falsch interpretiert wird, ist die Verwendung von Zahlen als boolesche Werte bei Java nicht erlaubt. Wenn Sie einen nicht-booleschen Wert in einem Test auswerten möchten, der ein boolesches Ergebnis liefert, zum Beispiel `if (a)`, müssen Sie den Ausdruck zuerst in einen booleschen Ausdruck umformen, hier `if (a != 0)`.

5.2 Die if/else-Anweisung

[3] Die Kombination der bedingenden Anweisungen `if` und `else` ist die einfachste Möglichkeit zur Steuerung des Programmablaufs. Der `else`-Teil ist optional, so daß Sie `if` auf zwei Arten verwenden können:

```
if (Boolean-expression)
    statement
```

oder

```
if (Boolean-expression)
    statement
else
    statement
```

Der boolesche Ausdruck `Boolean-expression` muß einen booleschen Wert liefern. Die Anweisung `statement` ist entweder eine einfache, per Semikolon abgeschlossene Anweisung oder eine zusammengesetzte Anweisung, das heißt eine Abfolge mehrerer einfacher Anweisungen, umschlossen von einem Paar geschweifter Klammern. Der Begriff „Anweisung“ bezeichnet von nun an stets eine einfache oder zusammengesetzte Anweisung.

[4] Die `test()`-Methode im folgenden Beispiel `if/else` gibt an, ob eine geratene Zahl kleiner, größer oder gleich einer gegebenen Zahl ist:

```
//: control/IfElse.java
import static net.mindview.util.Print.*;

public class IfElse {
    static int result = 0;
    static void test(int testval, int target) {
        if(testval > target)
            result = +1;
        else if(testval < target)
            result = -1;
        else
            result = 0; // Match
    }
    public static void main(String[] args) {
        test(10, 5);
        print(result);
        test(5, 10);
        print(result);
        test(5, 5);
        print(result);
    }
} /* Output:
      1
      -1
```

```
    0
*///:~
```

Die Kombination `else if` in der `test()`-Methode ist kein neues Schlüsselwort, sondern nur eine `else`-Anweisung, die eine neue `if`-Anweisung beinhaltet.

[5] Obwohl Java, wie seine Vorgänger C und C++ eine formatfreie Sprache ist, wird der Körper einer bedingenden Anweisung üblicherweise eingerückt, damit der Leser leichter erkennt, wo der Block beginnt beziehungsweise endet.

5.3 Schleifen

[6] Iteratives („schleifenartiges“) Verhalten wird mit Hilfe der *Iterationsanweisungen* („Schleifenanweisungen“) `while`, `do/while` und `for` definiert. Die Anweisung `statement` (der „Schleifenkörper“) wird solange wiederholt, bis der boolesche Ausdruck `Boolean-expression` den Wert `false` liefert. Die `while`-Schleife hat das Format:

```
while (Boolean-expression)
    statement
```

Der boolesche Ausdruck wird einmal vor Beginn der Schleife und anschließend vor jeder Verarbeitung des Schleifenkörpers ausgewertet.

[7] Das folgende einfache Beispiel erzeugt solange Zufallszahlen, bis eine bestimmte Bedingung erfüllt ist:

```
//: control/WhileTest.java
// Demonstrates the while loop.

public class WhileTest {
    static boolean condition() {
        boolean result = Math.random() < 0.99;
        System.out.print(result + ", ");
        return result;
    }
    public static void main(String[] args) {
        while(condition())
            System.out.println("Inside 'while'");
        System.out.println("Exited 'while'");
    }
} /* (Execute to see output) *////:~
```

Die Methode `condition()` ruft die statische `Math`-Methode `random()` auf, die einen `double`-Wert zwischen 0 und 1 zurückgibt (die Wertemenge enthält 0, nicht aber 1). Die lokale Variable `result` speichert den, von dem Vergleichsoperator `<` ermittelten `boolean`-Wert. Ein `boolean`-Wert wird bei der Ausgabe über `System.out.println()` automatisch in die Zeichenkette „true“ beziehungsweise „false“ umgewandelt. Der boolesche Ausdruck im Kopf der `while`-Schleife besagt, daß der Schleifenkörper solange ausgeführt wird, als die `condition()`-Methode `true` zurückgibt.

5.3.1 Die do/while-Schleife

[8] Die `do/while`-Schleife hat das Format:

```
do
    statement
while (Boolean-expression);
```

Der einzige Unterschied zwischen einer `while`- und einer `do`/`while`-Schleife besteht darin, daß der Schleifenkörper stets mindestens einmal ausgeführt wird, selbst wenn die Auswertung des booleschen Ausdrucks bereits beim ersten Mal `false` ergibt. Liefert der boolesche Ausdruck einer `while`-Schleife dagegen bereits bei seiner ersten Auswertung `false`, so wird der Schleifenkörper *nicht* ausgeführt. Die `do`/`while`-Schleife tritt in der Praxis weniger häufig auf, als die `while`-Schleife.

5.3.2 Die for-Schleife

[9] Die `for`-Schleife ist der wohl am häufigsten verwendete Schleifentyp. Die `for`-Schleife führt vor der ersten Verarbeitung des Schleifenkörpers eine Initialisierungsphase aus. Anschließend wird ein boolescher Ausdruck ausgewertet und nach jeder Verarbeitung des Schleifenkörpers eine „Schrittweite“ addiert oder subtrahiert. Die `for`-Schleife hat das Format:

```
for (initialization; Boolean-expression; step)
    statement
```

Initialisierungsausdruck, boolescher Ausdruck und De-/Inkrementierungsausdruck können leer sein. Der boolesche Ausdruck wird vor jeder Iteration ausgewertet. Liefert die Auswertung `false`, so wird das Programm mit der ersten Anweisung *nach* der `for`-Schleife fortgesetzt. Der De-/Inkrementierungsausdruck wird am Ende jeder Iteration ausgewertet.

[10] `for`-Schleifen dienen in der Regel zum Zählen:

```
//: control/ListCharacters.java
// Demonstrates ''for'' loop by listing
// all the lowercase ASCII letters.

public class ListCharacters {
    public static void main(String[] args) {
        for(char c = 0; c < 128; c++)
            if(Character.isLowerCase(c))
                System.out.println(''value: '' + (int)c + '' character: '' + c);
    }
} /* Output:
      value: 97 character: a
      value: 98 character: b
      value: 99 character: c
      value: 100 character: d
      value: 101 character: e
      value: 102 character: f
      value: 103 character: g
      value: 104 character: h
      value: 105 character: i
      value: 106 character: j
      ...
   *///:~
```

Beachten Sie, daß die lokale Variable `c` nicht am Anfang der `main()`-Methode deklariert wird, sondern im Kopf der Schleife, in der sie verwendet wird. Der Geltungsbereich von `c` ist der Körper der `for`-Schleife.

[11] Dieses kleine Programm nutzt außerdem die „Wrapperklasse“ `java.lang.Character`, die nicht nur einen primitiven `char`-Wert in einem Objekt verpackt, sondern auch eine Reihe von Hilfsme-

thoden zur Verfügung stellt. Die statische Methode `isLowerCase()` gibt zum Beispiel an, ob das fragliche Zeichen ein Kleinbuchstabe ist.

[12] Traditionelle prozedurale Sprachen wie C verlangen, daß alle Variablen am Anfang eines Blocks deklariert werden, so daß der Compiler beim Erzeugen eines Blocks den für diese Variablen erforderlichen Arbeitsspeicher allokieren kann. Bei Java und C++ können Sie Variablendeklarationen dagegen über den ganzen Block verteilen und dort platzieren, wo sie benötigt werden. Die Organisation des Quelltextes wird dadurch natürlicher und der Quelltext selbst leichter verständlich.

Übungsaufgabe 1: (1) Schreiben Sie ein Programm, das die Werte von 1 bis 100 ausgibt. ■

Übungsaufgabe 2: (2) Schreiben Sie ein Programm, das 25 zufällige `int`-Werte erzeugt. Verwenden Sie eine `if`/`else`-Anweisung, um jeweils zu ermitteln, ob der `int`-Wert größer, kleiner oder gleich einem zweiten, ebenfalls zufällig bestimmten `int`-Wert ist. ■

Übungsaufgabe 3: (1) Ändern Sie Übungsaufgabe 2, so daß Ihre Anweisungen in einer „unendlichen“ `while`-Schleife stehen. Das Programm läuft solange, bis Sie es per Tastatur abbrechen (typischerweise mittels Ctrl+C). ■

Übungsaufgabe 4: (3) Schreiben Sie ein Programm, das mit Hilfe zweier geschachtelter `for`-Schleifen und des Divisionsrestoperators (`%`) Primzahlen erkennt und ausgibt. (Primzahlen sind, mit Ausnahme der 2, ungerade ganze Zahlen, die nur durch sich selbst und 1 geteilt werden können.) ■

Übungsaufgabe 5: (5) Wiederholen Sie Übungsaufgabe 10 aus Abschnitt 4.10 (Seite 92). Verwenden Sie den ternären Operator und einen bitweisen Vergleich, anstelle der statischen `Integer`-Methode `toBinaryString()`, um die Einsen und Nullen anzuzeigen. ■

5.3.3 Der Kommaoperator

[13] Der Kommaoperator (nicht das *Trennzeichen* bei Definitionen und Methodenparametern) kommt bei Java nur an einer einzigen Stelle vor, nämlich im Kopf von `for`-Schleifen. Sowohl der Initialisierungs- als auch der De-/Inkrementierungsausdruck können aus mehreren, kommaseparierten Teilausdrücken bestehen, die nacheinander ausgewertet werden.

[14] Der Kommaoperator gestattet die Deklaration mehrerer lokaler Variablen im Körper einer `for`-Schleife, die allerdings alle vom gleichen Typ sein müssen:

```
//: control/CommaOperator.java
public class CommaOperator {
    public static void main(String[] args) {
        for(int i = 1, j = i + 10; i < 5; i++, j = i * 2) {
            System.out.println("i = " + i + " j = " + j);
        }
    }
} /* Output:
      i = 1 j = 11
      i = 2 j = 4
      i = 3 j = 6
      i = 4 j = 8
  *///:~
```

Die `int`-Deklaration im Kopf der `for`-Schleife erfaßt sowohl `i` als auch `j`. Der Initialisierungsausdruck kann beliebig viele Variablendeklarationen desselben Typs beinhalten. Die Deklaration lokaler

Variablen im Schleifenkopf ist auf den Schleifentyp `for` begrenzt. Diese Notation ist bei den anderen Selektions- und Schleifenanweisungen nicht möglich.

[15] Sie sehen, daß die Anweisungen sowohl im Initialisierungs- als auch im De-/Inkrementierungsausdruck sequentiell ausgewertet werden.

5.4 Die erweiterte for-Schleife („Foreach-Syntax“)

[16] Version 5 der Java Standard Edition (SE 5) führt eine neue, prägnantere Syntax zur Verarbeitung von Arrays (siehe Kapitel 17) und Containern (siehe Kapitel 12 und 18) ein. Diese *erweiterte `for`-Schleife* wird auch häufig als *Foreach-Syntax* bezeichnet und bewirkt, daß Sie keine lokale `int`-Variable mehr zu deklarieren brauchen, um eine Folge von Elementen nacheinander zur Verarbeitung auszuwählen. Die erweiterte `for`-Schleife liefert automatisch ein Element nach dem anderen.

[17] Angenommen, Sie wollten jedes Element eines `float`-Arrays einmal auswählen:

```
//: control/ForEachFloat.java
import java.util.*;

public class ForEachFloat {
    public static void main(String[] args) {
        Random rand = new Random(47);
        float f[] = new float[10];
        for(int i = 0; i < 10; i++)
            f[i] = rand.nextFloat();
        for(float x : f)
            System.out.println(x);
    }
} /* Output:
      0.72711575
      0.39982635
      0.5309454
      0.0534122
      0.16020656
      0.57799757
      0.18847865
      0.4170137
      0.51660204
      0.73734957
   */ //:~
```

Das Array wird mittels einer traditionellen `for`-Schleife bewertet, da die Wertzuweisung nur über den Index möglich ist. Sie sehen die erweiterte `for`-Schleife in der Zeile:

```
for (float x: f) {
```

Der Schleifenkopf deklariert eine Variable `x` vom Typ `float`, der nacheinander jedes Element des von `f` referenzierten Arrays einmal zugewiesen wird.

[18] Jede Methode, die eine Referenz auf ein Array zurück gibt, ist ein Kandidat für die erweiterte `for`-Schleife. Die Methode `toCharArray()` der Klasse `String` gibt beispielsweise ein `char`-Array zurück und gestattet zusammen mit der erweiterten `for`-Schleife die elementweise Verarbeitung der Zeichen in einer Zeichenkette:

```
//: control/ForEachString.java
public class ForEachString {
    public static void main(String[] args) {
```

```
        for(char c : ''An African Swallow''.toCharArray() )
            System.out.print(c + '' '');
    }
} /* Output:
      A n A f r i c a n S w a l l o w
   *////:~
```

In Abschnitt 12.13 lernen Sie, daß die erweiterte `for`-Schleife auf alle Objekte sogenannter *iterabler Klassen* angewendet werden kann (eine Klasse ist *iterabel*, wenn sie das Interface `Iterable` implementiert).

[19] Die traditionelle `for`-Schleife durchläuft eine Folge von ganzen Zahlen, zum Beispiel:

```
for (int i = 0; i < 100; i++)
```

Diese Funktionalität läßt sich nicht ohne weiteres durch die erweiterte `for`-Schleife ersetzen. Sie brauchen zuerst ein Array mit den entsprechenden `int`-Werten. Die Hilfsklasse `net.mindview.util.Range` definiert eine Methode namens `range()`, die automatisch ein passendes Array erzeugt, um diese Voraussetzung abzudecken. Die Klasse `Range` ist zum statischen Import vorgesehen:

```
//: control/ForEachInt.java
import static net.mindview.util.Range.*;
import static net.mindview.util.Print.*;

public class ForEachInt {
    public static void main(String[] args) {
        for(int i : range(10)) // 0..9
            printnb(i + '' '');
        print();
        for(int i : range(5, 10)) // 5..9
            printnb(i + '' '');
        print();
        for(int i : range(5, 20, 3)) // 5..20 step 3
            printnb(i + '' '');
        print();
    }
} /* Output:
      0 1 2 3 4 5 6 7 8 9
      5 6 7 8 9
      5 8 11 14 17
   *////:~
```

[20] Die `range()`-Methode ist *überladen*, das heißt derselbe Methodenname kann mit verschiedenen Argumentlisten kombiniert werden (das Überladen von Methoden wird in Abschnitt 6.2 behandelt). Die erste überladene Version von `range()` beginnt bei Null und liefert sämtliche Werte bis zum Vorgänger des übergebenen Höchstwertes. Die zweite Version beginnt bei ihrem ersten Argument und liefert sämtliche Werte bis zum Vorgänger des zweiten Argumentes. Die dritte Version inkrementiert die gelieferten Werte um die als drittes Argument übergebene Schrittweite. Die `range()`-Methode ist eine sehr einfache Version eines sogenannten *Generators* (das Konzept des Generators wird in Abschnitt 16.3 vorgestellt).

[21] Beachten Sie, daß die `range()`-Methode zwar die Anwendungsmöglichkeiten der Foreach-Syntax erweitert und damit wohl die Lesbarkeit verbessert, aber auch etwas weniger effizient ist. Zur Feinabstimmung der Performanz eines Programms empfiehlt sich der Einsatz eines Profilers, das heißt eines Hilfsprogramms, das die Performanz Ihres Programms mißt.

[22] Sie haben sicher die Verwendung von `printnb()` zusätzlich zur `print()`-Methode bemerkt. Die `printnb()`-Methode fügt keinen Zeilenumbruch an ihr Argument an und gestattet daher die

Ausgabe einer Zeile in Stücken.

[23] Die erweiterte `for`-Schleife spart nicht nur Zeit beim Schreiben des Quelltextes. Sie ist auch besser zu lesen und sagt aus, *was* Sie tun (elementweise Zugriff auf ein Array), statt Einzelheiten dazu anzugeben, *wie* Sie etwas tun (deklarieren eines Index, um nacheinander alle Elemente einmal auszuwählen). Die erweiterte `for`-Schleife wird in diesem Buch so oft wie möglich verwendet.

5.5 Die return-Anweisung

[24] Einige Schlüsselwörter repräsentieren *unbedingte Verzweigungen*, das heißt in einfacheren Worten, daß die Verzweigung ohne eine vorherige Prüfung ausgeführt wird. Diese Schlüsselwörter sind `return`, `break`, `continue` sowie der Sprung zu einer markierten Anweisung, ähnlich der `goto`-Anweisung in anderen Sprachen.

[25] Das Schlüsselwort `return` hat zwei Aufgaben: Es gibt an, welchen Wert eine Methode zurückgibt (falls die Methode nicht den Rückgabetyp `void` deklariert) und veranlaßt die Rückkehr aus der gegenwärtig ausgeführten Methode, wobei dieser Wert zurückgegeben wird. Das folgende Beispiel zeigt nochmals die `test()`-Methode aus dem Beispiel *IfElse.java* auf Seite 112, diesesmal aber mit Rückgabewert:

```
//: control/IfElse2.java
import static net.mindview.util.Print.*;

public class IfElse2 {
    static int test(int testval, int target) {
        if(testval > target)
            return +1;
        else if(testval < target)
            return -1;
        else
            return 0; // Match
    }
    public static void main(String[] args) {
        print(test(10, 5));
        print(test(5, 10));
        print(test(5, 5));
    }
} /* Output:
      1
      -1
      0
   *///:~
```

Die Anweisungen im Methodenkörper nach der ausgeführten `return`-Anweisung werden nicht mehr verarbeitet.

[26] Wenn Sie in einer Methode ohne Rückgabewert (`void`) keine `return`-Anweisung definieren, wird am Ende des Methodenkörpers eine implizite `return`-Anweisung eingesetzt. Deklariert Ihre Methode aber einen von `void` verschiedenen Rückgabetyp, so müssen Sie sicherstellen, daß jeder Ausführungszweig einen Wert zurückgibt.

Übungsaufgabe 6: (2) Ändern Sie die `test()`-Methoden der Beispiele *IfElse.java* (Seite 112) und *IfElse2.java* (Seite 118), so daß die Methoden zwei weitere Argumente `begin` und `end` erwarten und ermitteln, ob das Argument `testval` in dem durch `begin` und `end` definierten Abschnitt liegt (inklusive der Randpunkte). ■

5.6 Die Anweisungen break und continue

[27] Die Anweisungen `break` und `continue` gestatten die Steuerung der Verarbeitung innerhalb des Schleifenkörpers bei allen Schleifentypen. Die `break`-Anweisung bricht die Verarbeitung der Schleife ab, ohne die restlichen Anweisungen im Schleifenkörper auszuführen. Die `continue`-Anweisung bricht den aktuellen Schleifendurchlauf ab und kehrt zum Schleifenkopf zurück, um die nächste Iteration zu beginnen.

[28] Das folgenden Beispiel zeigt die Anweisungen `break` und `continue` in `for`- und `while`-Schleifen:

```
//: control/BreakAndContinue.java
// Demonstrates break and continue keywords.
import static net.mindview.util.Range.*;

public class BreakAndContinue {
    public static void main(String[] args) {
        for(int i = 0; i < 100; i++) {
            if(i == 74) break; // Out of for loop
            if(i % 9 != 0) continue; // Next iteration
            System.out.print(i + " ");
        }
        System.out.println();
        // Using foreach:
        for(int i : range(100)) {
            if(i == 74) break; // Out of for loop
            if(i % 9 != 0) continue; // Next iteration
            System.out.print(i + " ");
        }
        System.out.println();
        int i = 0;
        // An "infinite loop":
        while(true) {
            i++;
            int j = i * 27;
            if(j == 1269) break; // Out of loop
            if(i % 10 != 0) continue; // Top of loop
            System.out.print(i + " ");
        }
    }
} /* Output:
      0 9 18 27 36 45 54 63 72
      0 9 18 27 36 45 54 63 72
      10 20 30 40
   *///:~
```

Die lokale Variable `i` in der ersten `for`-Schleife erreicht den Wert 99 niemals, da die `break`-Anweisung die Schleife abbricht, wenn `i` den Wert 74 erreicht hat. Im allgemeinen wählen Sie nur dann eine `break`-Anweisung, wenn Sie nicht vorhersehen können, wann die Abbruchbedingung eintritt. Die `continue`-Anweisung bewirkt, daß die Programmausführung jedesmal zum Schleifenkopf zurückspringt (die Variable `i` also inkrementiert), wenn `i` nicht ohne Rest durch 9 teilbar ist. Andernfalls wird `i` ausgegeben.

[29] Die zweite `for`-Schleife zeigt dasselbe Verhalten, aber mit der Foreach-Syntax und liefert dasselbe Ergebnis. Die dritte Schleife ist eine „unendliche“ `while`-Schleife, die theoretisch bis in alle Ewigkeit läuft. Der Schleifenkörper enthält aber eine `break`-Anweisung, um die Schleife abzubrechen.

[30] In allen drei Schleifen kehrt die Programmausführung bei `continue` zum Schleifenkopf zurück, ohne die restlichen Anweisungen nach `continue` zu beachten. Die zweite Iteration liefert also nur dann eine Ausgabe, wenn `i` ohne Rest durch 9 geteilt werden kann. Die beiden `for`-Schleifen geben die 0 aus, weil 0 ohne Rest durch 9 teilbar ist.

[31] Es gibt noch eine zweite Syntaxvariante der unendlichen Schleife: `for (;;)`. Der Compiler behandelt `while (true)` und `for (;;)` gleich. Sie können also die Schreibweise wählen, die Ihnen eher zusagt.

Übungsaufgabe 7: (1) Ändern Sie Übungsaufgabe 1 (Seite 115), so daß das Programm beim Wert 99 per `break`-Anweisung beendet wird. Probieren Sie statt dessen auch die `return`-Anweisung. ■

5.7 Die berüchtigte goto-Anweisung und markierte Schleifen

[32] Das Schlüsselwort `goto` war von Anfang an in vielen Programmiersprachen vorhanden. Tatsächlich war `goto` sogar der Ursprung der Programmsteuerung in Assembler: „Wenn Bedingung `A` gilt, dann springe hier hin, andernfalls springe dort hin.“ Wenn Sie sich den Assemblercode ansehen, der letztendlich von fast jedem Compiler generiert wird, werden Sie feststellen, daß die Programmsteuerung viele Sprünge beinhaltet. (Der Java-Compiler erzeugt einen eigenen „Assemblercode“, der allerdings von der Laufzeitumgebung und nicht direkt vom Prozessor ausgeführt wird.)

[33] Eine `goto`-Anweisung ist eine Sprunganweisung auf Quelltextebene und diese Eigenschaft hat `goto` in Verruf gebracht. Gibt es, wenn ein Programm stets von einer Stelle zur nächsten springt, keine Möglichkeit, den Quelltext zu umzuordnen, daß der Kontrollfluß weniger sprunghaft ist? Die `goto`-Anweisung fiel nach Edsger Wybe Dijkstras berühmtem Artikel *Goto considered harmful* wirklich in Ungnade. Seither ist das öffentliche Beschimpfen von `goto` ein beliebter Zeitvertreib, bei dem die Befürworter des verstoßenen Schlüsselwortes in Deckung gehen.

[34] Typischerweise ist in einer solchen Situation ein mittleres Niveau am fruchtbarsten. Das Problem besteht nicht etwa im Gebrauch der `goto`-Anweisung selbst, sondern in der *übertriebenen* Verwendung. Es gibt seltene Fälle, in denen `goto` eigentlich die beste Möglichkeit ist, um den Kontrollfluß zu strukturieren

[35] Das Schlüsselwort `goto` ist bei Java zwar reserviert, wird aber in der Sprache nicht angewendet, kurz: Java kennt keine `goto`-Anweisung. Die `goto`-Anweisung hat aber eine gewisse Ähnlichkeit mit den Anweisungen `break` und `continue`, die allerdings keine Sprunganweisungen sind, sondern Möglichkeiten zum Abbruch der Verarbeitung des aktuellen Schleifendurchlaufs darstellen. Der Grund, warum `break` und `continue` in Diskussionen über `goto` vorkommen, besteht darin, daß sie auf dem demselben Mechanismus beruhen, nämlich Marken.

[36] Eine Marke ist ein mit einem Doppelpunkt abgeschlossener Bezeichner:

```
label1:
```

Es gibt bei Java nur eine einzige Stelle, um eine Marke sinnvoll zu platzieren, nämlich *unmittelbar* vor einer Schleife. Zwischen Marke und Schleife sind keine Anweisungen erlaubt. Der einzige Anwendungfall für eine Markierung ist eine Schleife, die eine Schleife oder eine `switch`-Anweisung (siehe Abschnitt 5.8) enthält, da `break` und `continue` in der Regel nur auf die aktuelle Schleife wirken. Folgt `break` oder `continue` aber der Name einer Marke, so beziehen sich die Schlüsselwörter auf die entsprechende markierte Schleife:

```
label1:
outer-iteration {
    inner-iteration {
```

```
        // ...
        break;            // (1)
        // ...
        continue;         // (2)
        // ...
        continue label1;  // (3)
        // ...
        break label1;     // (4)
    }
}
```

Die `break`-Anweisung bei (1) bricht die innere Schleife ab. Die Verarbeitung der äußeren Schleife wird fortgesetzt. Die `continue`-Anweisung bei (2) springt zum Kopf der inneren Schleife und beginnt den nächsten Durchlauf der inneren Schleife. Die `continue`-Anweisung mit Marke bei (3) bricht die innere Schleife ab, springt zum Kopf der äußeren Schleife und beginnt den nächsten Durchlauf der äußeren Schleife. Die `break`-Anweisung mit Marke bei (4) bricht die innere und die äußere Schleife ab. Beide Schleifen werden nicht fortgesetzt.

[37] Ein Beispiel mit geschachtelten `for`-Schleifen:

```
//: control/LabeledFor.java
// For loops with ''labeled break'' and ''labeled continue.''
import static net.mindview.util.Print.*;

public class LabeledFor {
    public static void main(String[] args) {
        int i = 0;
        outer: // Can't have statements here
        for(; true ;) { // infinite loop
            inner: // Can't have statements here
            for(; i < 10; i++) {
                print(''i = '' + i);
                if(i == 2) {
                    print(''continue'');
                    continue;
                }
                if(i == 3) {
                    print(''break'');
                    i++; // Otherwise i never
                    // gets incremented.
                    break;
                }
                if(i == 7) {
                    print(''continue outer'');
                    i++; // Otherwise i never
                    // gets incremented.
                    continue outer;
                }
                if(i == 8) {
                    print(''break outer'');
                    break outer;
                }
                for(int k = 0; k < 5; k++) {
                    if(k == 3) {
                        print(''continue inner'');
                        continue inner;
                    }
                }
```

```
                }
            }
            // Can't break or continue to labels here
        }
    } /* Output:
          i = 0
          continue inner
          i = 1
          continue inner
          i = 2
          continue
          i = 3
          break
          i = 4
          continue inner
          i = 5
          continue inner
          i = 6
          continue inner
          i = 7
          continue outer
          i = 8
          break outer
       *///:~
```

Beachten Sie, daß die `break`-Anweisung die `for`-Schleife abbricht, der Inkrementierungsausdruck aber erst am Ende eines Schleifendurchgangs ausgewertet wird. Da die `break`-Anweisung den Inkrementierungsausdruck übergeht, wird `i` bei `i == 3` direkt erhöht. Die `continue`-Anweisung mit der Marke `outer` bei `i == 7` springt zum Kopf der äußeren Schleife und läßt die ebenfalls die Inkrementierung aus, so daß `i` auch hier direkt erhöht wird.

[38] Außer der `break`-Anweisung mit der Marke `outer` gibt es für die innere Schleife keine Möglichkeit, die äußere Schleife zu verlassen, da sich eine `break`-Anweisung stets auf die innerste Schleife bezieht (analog für `continue`).

[39] Bedeutet das Abbrechen einer Schleife zugleich die Rückkehr aus einer Methode, so können Sie auch einfach die `return`-Anweisung verwenden.

[40] Das nächste Beispiel zeigt markierte `break`- und `continue`-Anweisungen bei geschachtelten `while`-Schleifen:

```
//: control/LabeledWhile.java
// While loops with ''labeled break'' and ''labeled continue.''
import static net.mindview.util.Print.*;

public class LabeledWhile {
    public static void main(String[] args) {
        int i = 0;
        outer:
        while(true) {
            print(''Outer while loop'');
            while(true) {
                i++;
                print(''i = '' + i);
                if(i == 1) {
                    print(''continue'');
                    continue;
                }
```

```
                    if(i == 3) {
                        print(''continue outer'');
                        continue outer;
                    }
                    if(i == 5) {
                        print(''break'');
                        break;
                    }
                    if(i == 7) {
                        print(''break outer'');
                        break outer;
                    }
                }
            }
        }
    } /* Output:
          Outer while loop
          i = 1
          continue
          i = 2
          i = 3
          continue outer
          Outer while loop
          i = 4
          i = 5
          break
          Outer while loop
          i = 6
          i = 7
          break outer
       *///:~
```

[41] Für `while`- und `for`-Schleifen gelten dieselben Regeln:

- Eine unmarkierte `continue`-Anweisung springt zum Kopf der innersten Schleife und beginnt mit dem nächsten Durchlauf.
- Eine markierte `continue`-Anweisung springt zum Kopf der markierten Schleife und beginnt mit dem nächsten Durchlauf.
- Eine unmarkierte `break`-Anweisung bricht die Schleife ab. Das Programm wird mit der ersten Anweisung nach der Schleife fortgesetzt.
- Eine markierte `break`-Anweisung bricht die markierte Schleife ab. Das Programm wird mit der ersten Anweisung nach der markierten Schleife fortgesetzt.

Denken Sie daran, daß es bei Java nur einen Grund gibt, um Marken zu verwenden: Sie haben geschachtelte Schleifen und müssen mittels `break` oder `continue` mehr als eine Schachtelungsebene überwinden.

[42] Dijkstra beanstandet in seinem Artikel *Goto considered harmful* vor allem die Marken, weniger die `goto`-Anweisung. Dijkstra hatte beobachtet, daß die Anzahl der Fehler mit der Anzahl der Marken in einem Programm wächst und daß Marken und `goto`-Anweisungen die Untersuchung von Programmen verkomplizieren. Beachten Sie, daß die Marken bei Java nicht unter diesem Problem leiden und nicht zu dem Zweck gedacht sind, den Programmablauf zu ändern. Marken sind außerdem ein Beispiel für die Steigerung des Nutzens einer Spracheigenschaft durch die Einschränkung einer Anweisung.

5.8 Die switch-Anweisung

[43] Die `switch`-Anweisung (auch *Selektionsanweisung* wählt durch die Auswertung eines Ausdrucks, der einen ganzzahligen Wert liefert, eine einfache oder zusammengesetzte Anweisung zur Verarbeitung aus. Die `switch`-Anweisung hat das Format:

```
switch (integral-selector) {
    case integral-selector1: statement; break;
    case integral-selector2: statement; break;
    case integral-selector3: statement; break;
    case integral-selector4: statement; break;
    case integral-selector5: statement; break;
    // ...
    default: statement;
}
```

Der Ausdruck `integral-selector` liefert einen ganzzahligen Wert. Die `switch`-Anweisung vergleicht den Wert dieses Ausdrucks mit jedem `integral-selectorX`. Bei Übereinstimmung wird die korrespondierende einfache oder zusammengesetzte Anweisung (Klammern sind hier nicht erforderlich) ausgeführt. Findet die `switch`-Anweisung keine Übereinstimmung, so wird der `default`-Zweig ausgeführt.

[44] Sicher ist Ihnen aufgefallen, daß jeder `case`-Zweig mit `break` endet. Die `break`-Anweisung veranlaßt die Programmausführung zum Sprung an das Ende des Körpers der `switch`-Anweisung. Dies ist die konventionelle Art, eine `switch`-Anweisung anzulegen, aber die `break`-Anweisung ist optional. Ohne `break` werden die Anweisungen der folgenden `case`-Zweige ausgeführt, bis eine `break`-Anweisung auftritt. Dieses Verhalten ist zwar in der Regel unerwünscht, kann aber für den erfahrenen Programmierer nützlich sein. Beachten Sie, daß die letzte Anweisung im `default`-Zweig nicht durch `break` abgeschlossen ist, da die Programmausführung ohnehin bis zu der Stelle „durchfällt“, an die sie mit `break` gelangen würde. Sie können den `default`-Zweig aber mit einer `break`-Anweisung abschließen, wenn Sie es aus stilistischen Gründen für wichtig halten.

[45] Die `switch`-Anweisung eignet sich, um eine Mehrfachselektion (das heißt eine Auswahl aus mehreren verschiedenen Ausführungspfaden) sauber zu implementieren, setzt allerdings ein Selektionskriterium voraus, das sich in einen ganzzahligen Wert wie `int` oder `char` umwandeln läßt. Wenn Sie beispielsweise eine Zeichenkette oder eine Fließkommazahl als Kriterium verwenden möchten, scheidet die `switch`-Anweisung aus. Bei einem nicht-ganzzahligen Kriterium, müssen Sie `if`-Anweisungen kombinieren. In Abschnitt 6.9 finden Sie ein Beispiel für die Lockerung dieser Einschränkung durch die neuen Aufzählungstypen in der SE 5, die entwurfsbedingt gut mit `switch`-Anweisungen zusammenarbeiten.

[46] Das folgende Beispiel erzeugt zufällige Buchstaben und bestimmt anschließend, ob es Vokale oder Konsonanten sind:

```
//: control/VowelsAndConsonants.java
// Demonstrates the switch statement.
import java.util.*;
import static net.mindview.util.Print.*;

public class VowelsAndConsonants {
    public static void main(String[] args) {
        Random rand = new Random(47);
        for(int i = 0; i < 100; i++) {
            int c = rand.nextInt(26) + 'a';
            printnb((char)c + ", " + c + ": ");
            switch(c) {
```

```
            case 'a':
            case 'e':
            case 'i':
            case 'o':
            case 'u': print(''vowel'');
                break;
            case 'y':
            case 'w': print(''Sometimes a vowel'');
                break;
            default: print(''consonant'');
            }
        }
    }
} /* Output:
      y, 121: Sometimes a vowel
      n, 110: consonant
      z, 122: consonant
      b, 98: consonant
      r, 114: consonant
      n, 110: consonant
      y, 121: Sometimes a vowel
      g, 103: consonant
      c, 99: consonant
      f, 102: consonant
      o, 111: vowel
      w, 119: Sometimes a vowel
      z, 122: consonant
      ...
   *///:~
```

Die `Random`-Methode `nextInt()` liefert Werte zwischen 0 und 25 (inklusive der Randpunkte). Es genügt, den „Versatz“ `'a'` zu diesem Zufallswert zu addieren, um die Kleinbuchstaben zu bekommen. Die Zeichen in einfachen Anführungszeichen in den `case`-Zweigen entsprechen ebenfalls ganzen Zahlen und dienen zum Vergleichen der zufällig gewählten Buchstaben.

[47] Beachten Sie die „Stapelung“ der `case`-Zweige, um eine bestimmte Anweisung in mehr als einem Fall aufrufen zu können. Das Beispiel soll Ihnen auch die Bedeutung der `break`-Anweisung am Ende eines `case`-Zweiges bewußt machen. Andernfalls „fällt“ die Programmausführung einfach bis zum nächsten `case`-Zweig „durch“ und verarbeitet die dortige(n) Anweisung(en).

[48] In der Zeile

```
int c = rand.nextInt(26) + 'a';
```

gibt die `Random`-Methode `nextInt()` einen zufälligen `int`-Wert zwischen 0 und 25 (inklusive der Randpunkte) zurück, zu dem der Wert `'a'` addiert wird. Das bedeutet, daß `'a'` automatisch in einen `int`-Wert umgewandelt wird, um die Addition auszuführen.

[49] Soll `c` als Zeichen ausgegeben werden, so muß `c` in den Typ `char` umgewandelt werden. Andernfalls erfolgt die Ausgabe als ganzzahliger Wert.

Übungsaufgabe 8: (2) Legen Sie in einer `for`-Schleife eine `switch`-Anweisung an, die pro `case`-Zweig eine Textmeldung ausgibt. Die `for`-Schleife bewirkt, daß nacheinander alle `case`-Zweige aufgerufen werden. Beenden Sie jeden `case`-Zweig mit einer `break`-Anweisung und testen Sie das Programm. Entfernen Sie nun die `break`s und beobachten Sie was geschieht. ■

Übungsaufgabe 9: (4) Die Fibonacci-Folge besteht aus den Zahlen 1, 1, 2, 3, 5, 8, 13, 21, 34 und so weiter, wobei jedes Glied (ab dem dritten) die Summe seiner beiden Vorgänger ist. Schreiben Sie

eine Methode, die eine ganze Zahl erwartet (die Anzahl der Folgenglieder) und das entsprechende Anfangsstück der Fibonacci-Folge anzeigt. Die Ausgabe von `java Fibonacci 5` (`Fibonacci` ist der Name der Klasse, in die Methode definiert ist) lautet beispielsweise: 1, 1, 2, 3, 5. ■

Übungsaufgabe 10: (5) Eine Vampirzahl hat eine gerade Anzahl von Ziffern und wird durch Multiplikation zweier Zahlen gebildet, welche die halbe Anzahl von Ziffern haben. Die Ziffern stammen aus der ursprünglichen Zahl und dürfen *einmal* umgeordnet werden. Paare nachlaufender Nullen sind nicht erlaubt. Beispiele für Vampirzahlen sind: 1260 = 21∗60, 1827 = 21∗87 und 2187 = 27∗81. Schreiben Sie ein Programm, das sämtliche vierstelligen Vampirzahlen ermittelt (vorgeschlagen von Dan Forhan). ■

5.9 Zusammenfassung

[50] Dieses Kapitel beschließt die Behandlung der fundamentalen Eigenschaften, die in den meisten Programmiersprachen vorkommen: Berechnungen, Operatorvorrang, Typumwandlung, Selektion und Schleifen. Sie sind nun bereit für die ersten Schritte in die Welt der objektorientierten Programmierung. Das folgende Kapitel widmet sich dem wichtigen Thema der Initialisierung und des Aufräumens von Objekten, das übernächste Kapitel behandelt Zugriffsberechtigung (das Verbergen der Implementierung).

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Teil III

Grundlagen

Kapitel 6

Initialisierung und Aufräumen von Objekten

Inhaltsübersicht

[0] Während der „Computerrevolution“ hat sich die „unsichere“ Programmierung als wesentlicher Kostenverursacher herausgestellt.

[1] Die Initialisierung und das Aufräumen von Objekten sind zwei dieser Sicherheitsaspekte. Viele Fehler in C-Programmen kommen daher, daß der Programmierer eine Variable nicht initialisiert hat. Dies gilt in besonderem Maß für Bibliotheken, wenn die Clientprogrammierer nicht wissen, wie eine Bibliothekskomponente initialisiert wird beziehungsweise, daß sie überhaupt initialisiert werden muß. Das Problem beim Aufräumen besteht darin, daß eine Komponente eines Programmes nach ihrer Verwendung leicht in Vergessenheit gerät, da sie nicht länger gebraucht wird. Die von dieser Komponente beanspruchten Ressourcen verharren in diesem Zustand und können im weiteren Programmablauf knapp werden (vor allem der Arbeitsspeicher).

[2] C++ hat das Konzept des Konstruktors eingeführt, einer besonderen Methode, die beim Erzeugen eines Objektes automatisch aufgerufen wird. Java hat den Konstruktor übernommen und verfügt zusätzlich über eine automatische Speicherbereinigung, die Arbeitsspeicher freigibt, wenn er nicht mehr verwendet wird. Dieses Kapitel untersucht die Themen Initialisierung und Aufräumen sowie ihre Unterstützung in Java.

6.1 Garantierte Initialisierung von Objekten per Konstruktor

[3] Sie könnten in jeder Ihrer Klassen eine Methode namens `initialize()` anlegen. Der Name wäre ein Hinweis dahingehend, daß diese Methode vor der Verwendung eines Objektes einer solchen Klasse aufgerufen werden muß. Der Clientprogrammierer muß allerdings stets daran denken, diese Methode aufzurufen. Bei Java kann der Programmierer einer Klasse die Initialisierung der Objekte garantieren, indem er einen Konstruktor definiert. Verfügt eine Klasse über einen Konstruktor, so wird er beim Erzeugen eines Objektes dieser Klasse automatisch von der Laufzeitumgebung aufgerufen, bevor der Clientprogrammierer die Klasse „in die Hand bekommt“. Die Initialisierung ist somit gewährleistet.

[4] Die nächste Aufgabe besteht darin, einen Namen für diese Methode zu finden. Dabei sind zwei Dinge zu berücksichtigen. Erstens kann jeder Name den Sie wählen mit dem Namen einer Komponente der Klasse kollidieren. Zweitens ist der Compiler dafür zuständig, den Konstruktor aufzurufen, muß also stets wissen, wie die aufzurufende Methode heißt. Die für C++ gewählte Lösung scheint die einfachste und logischste zu sein und wurde daher auch bei Java verwendet: Der Name des Konstruktors stimmt mit dem Namen der Klasse überein. Es leuchtet ein, daß eine Methode mit diesem Namen während der Objekterzeugung automatisch aufgerufen wird.

[5] Das folgende Beispiel zeigt eine einfache Klasse mit einem Konstruktor:

```
//: initialization/SimpleConstructor.java
// Demonstration of a simple constructor.

class Rock {
    Rock() { // This is the constructor
        System.out.print("Rock ");
    }
}

public class SimpleConstructor {
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++)
            new Rock();
    }
```

```
} /* Output:
      Rock Rock Rock Rock Rock Rock Rock Rock Rock Rock
   *///:~
```

Beim Erzeugen eines `Rock`-Objektes per

```
new Rock();
```

wird Arbeitsspeicher allokiert und der Konstruktor aufgerufen. Das Objekt wird garantiert richtig initialisiert, bevor Sie anfangen können, damit zu arbeiten.

[6] Beachten Sie, daß die Richtlinie, einen Kleinbuchstaben als ersten Buchstaben eines Methodenamen zu wählen, bei Konstruktoren nicht gilt, da der Name des Konstruktors und der Klassenname *exakt* übereinstimmen müssen.

[7–8] Ein Konstruktor der keine Argumente erwartet (keine Parameter definiert), heißt *Standardkonstruktor*. Die englischsprachige Java-Dokumentation verwendet typischerweise die Bezeichnung *No-arg constructor*. Der Begriff „Standardkonstruktor“ hatte sich andererseits schon viele Jahre vor Java etabliert und ich tendiere dazu, diese Bezeichnung zu verwenden. Ein Konstruktor kann, wie jede Methode, Argumente erwarten, mit denen Sie die Erzeugung eines Objektes beeinflussen können. Das obige Beispiel läßt sich leicht ändern, so daß der Konstruktor ein Argument erwartet:

```
//: initialization/SimpleConstructor2.java
// Constructors can have arguments.

class Rock2 {
    Rock2(int i) {
        System.out.print("Rock " + i + " ");
    }
}

public class SimpleConstructor2 {
    public static void main(String[] args) {
        for(int i = 0; i < 8; i++)
            new Rock2(i);
    }
} /* Output:
      Rock 0 Rock 1 Rock 2 Rock 3 Rock 4 Rock 5 Rock 6 Rock 7
   *///:~
```

Konstruktorargumente gestatten Ihnen, die Initialisierung eines Objektes zu parametrisieren. Die Klasse `Tree` könnte beispielsweise einen Konstruktor mit einem einzelnen Parameter definieren, der die Höhe des Baumes angibt. Sie würden ein `Tree`-Objekt folgendermaßen erzeugen:

```
Tree t = new Tree(12); // 12-foot tree
```

Ist `Tree(int)` der einzige Konstruktor der Klasse `Tree`, so gestattet Ihnen der Compiler nur auf diese eine Weise, ein `Tree`-Objekt zu erzeugen.

[9] Konstruktoren lösen eine Reihe von Problemen und wirken sich positiv auf die Lesbarkeit des Quelltextes aus. Der obige Aufruf des `Tree`-Konstruktors ist nicht mit dem Aufruf einer Initialisierungsmethode verknüpft, deren Wirkung konzeptionell von der Objekterzeugung getrennt ist. Objekterzeugung und -initialisierung gehören bei Java zusammen. Sie können das eine nicht ohne das andere haben.

[10] Ein Konstruktor hat keinen Rückgabewert und ist mit dieser Eigenschaft eine ungewöhnliche Methode. Ein Konstruktor unterscheidet sich allerdings deutlich von einer Methode mit dem Rückgabetyp `void`, die zwar keinen Wert zurückgibt, deren Rückgabeverhalten aber geändert werden kann. Konstruktoren liefern keinen Rückgabewert und Sie können nichts daran ändern (der `new`-

Ausdruck gibt zwar die Referenz auf das neu erzeugte Objekt zurück, aber der Konstruktor selbst hat keinen Rückgabewert). Hätten Konstruktoren Rückgabewerte und könnten die Programmierer diese Werte selbst bestimmen, so müßte der Compiler irgendwie darüber informiert werden, was er mit diesen Rückgabewerten anfangen soll.

Übungsaufgabe 1: (1) Schreiben Sie eine Klasse, die ein nicht initialisiertes `String`-Feld enthält. Zeigen Sie, daß dieses Feld bei Java mit `null` initialisiert wird. ■

Übungsaufgabe 2: (2) Schreiben Sie eine Klasse mit zwei `String`-Feldern. Eines wird an der Stelle seiner Deklaration initialisiert, das andere vom Konstruktor. Wodurch unterscheiden sich die beiden Initialisierungsmöglichkeiten? ■

6.2 Überladen von Methoden

[11] Die Verwendung von Namen (Bezeichnern) ist eine grundlegende Anforderung an jede Programmiersprache. Beim Erzeugen eines Objektes, verknüpfen Sie einen Speicherbereich mit einem Namen. Eine Methode bezeichnet eine Handlung. Der Bezug auf ein Objekt oder eine Methode geschieht stets durch seinen beziehungsweise ihren Namen. Gutgewählte Namen gestatten anderen Programmierern den Quelltext eines Programms oder einer Anwendung leichter zu verstehen und zu ändern. Es ist wie beim Schreiben von Prosa: Das Ziel ist, mit dem Leser zu kommunizieren.

[12] Die Abbildung des Konzeptes der Nuance in einer menschlichen Sprache auf eine Programmiersprache birgt ein Problem. Ein und dasselbe Wort drückt häufig verschiedene Bedeutungen aus, ist also hinsichtlich seiner Bedeutung *überladen*. Sie sagen beispielsweise „Wasche das Hemd“, „Wasche das Auto“ und „Wasche den Hund“. Es wäre unsinnig, sagen zu müssen „Hemd-wasche das Hemd“, „Auto-wasche das Auto“ und „Hund-wasche den Hund“, nur damit der Zuhörer keine Unterscheidung zwischen den einzelnen Handlungen vornehmen muß. Die meisten menschlichen Sprachen sind redundant, so daß sich eine Aussage auch dann noch ermitteln läßt, wenn einige Worte fehlen. Sie brauchen keine eindeutigen Bezeichnungen, da Sie die Interpretation aus dem Kontext ableiten können.

[13] Die meisten Programmiersprachen (darunter insbesondere C) verlangen für jede Methode einen eindeutigen Namen (Methoden werden in diesen Sprachen häufig als „Funktionen“ bezeichnet). In einer solchen Programmiersprache können Sie nicht eine `print()`-Funktion für ganzzahlige Werte und eine andere `print()`-Funktion für Fließkommazahlen definieren, da jede Funktion einen eindeutigen Namen haben muß.

[14] Bei Java (und C++) erzwingt noch ein anderer Faktor die Überladung von Methodennamen, nämlich der Konstruktor. Da der Name des Konstruktors durch den Klassennamen vorgegeben ist, kommt nur ein Konstruktorname in Frage. Aber was ist, wenn Sie auf mehr als eine Weise Objekte erzeugen können wollen? Stellen Sie zum Beispiel vor, eine Klasse soll ihre Objekt entweder auf eine standardisierte Weise oder durch Einlesen von Informationen aus einer Datei initialisieren. In diesem Fall brauchen Sie zwei Konstruktoren, nämlich einen Standardkonstruktor und einen, der ein Argument vom Typ `String` erwartet (den Namen der Datei aus der das Objekt initialisiert werden soll). Beide sind Konstruktoren und daher an den Klassennamen gebunden. Das Überladen von Methoden ist also wesentlich, um die Verwendung eines Methodennamens mit unterschiedlichen Argumenttypen zu ermöglichen. Das Überladen ist für Konstruktoren zwingend erforderlich und zugleich eine komfortable Fähigkeit, die sich auch auf jede andere Methode anwenden läßt.

[15–16] Das folgende Beispiel zeigt sowohl einen überladenen Konstruktor als auch eine überladene Methode:

```
//: initialization/Overloading.java
// Demonstration of both constructor
// and ordinary method overloading.
import static net.mindview.util.Print.*;

class Tree {
    int height;
    Tree() {
        print("Planting a seedling");
        height = 0;
    }
    Tree(int initialHeight) {
        height = initialHeight;
        print("Creating new Tree that is " +
              height + " feet tall");
    }
    void info() {
        print("Tree is " + height + " feet tall");
    }
    void info(String s) {
        print(s + ": Tree is " + height + " feet tall");
    }
}

public class Overloading {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            Tree t = new Tree(i);
            t.info();
            t.info("overloaded method");
        }
        // Overloaded constructor:
        new Tree();
    }
} /* Output:
      Creating new Tree that is 0 feet tall
      Tree is 0 feet tall
      overloaded method: Tree is 0 feet tall
      Creating new Tree that is 1 feet tall
      Tree is 1 feet tall
      overloaded method: Tree is 1 feet tall
      Creating new Tree that is 2 feet tall
      Tree is 2 feet tall
      overloaded method: Tree is 2 feet tall
      Creating new Tree that is 3 feet tall
      Tree is 3 feet tall
      overloaded method: Tree is 3 feet tall
      Creating new Tree that is 4 feet tall
      Tree is 4 feet tall
      overloaded method: Tree is 4 feet tall
      Planting a seedling
   *///:~
```

Ein `Tree`-Objekt kann entweder als Setzling (ohne Argumente) oder als Pflanze in einer Baumschule erzeugt werden, die bereits auf eine bestimmte Höhe angewachsen ist. Die Klasse `Tree` definiert daher einen Standardkonstruktor und einen weiteren Konstruktor, der die vorhandene Höhe erfaßt.

[17] Auch die `info()`-Methode kann auf zwei verschiedene Arten aufgerufen werden. Die `in-`

`fo(String)`-Version gestattet die Ausgabe einer zusätzlichen Meldung, während Sie die parameterlose `info()`-Version wählen können, wenn keine weiteren Informationen ausgegeben werden sollen. Es wäre irreführend, zwei verschiedene Namen für dasselbe Konzept zu vergeben. Die Methodenüberladung gestattet, beide Methoden identisch zu benennen.

6.2.1 Unterscheidung überladener Versionen

[18] Woran erkennt Java, welche Methode Sie meinen, wenn beide Methoden denselben Namen haben? Die einfache Regel lautet: Jede Version einer überladenen Methode muß eine eindeutige Liste von Parametertypen definieren.

[19] Wenn Sie diese Regel einen Moment auf sich wirken lassen, erkennen Sie, das sie sinnvoll ist. Wie sonst könnte der Programmierer einen Unterschied zwischen zwei gleichnamigen Methoden definieren, als über die Typen der Parameter?

[20] Selbst das Umordnen der Parameterliste reicht aus, um zwei Versionen einer Methode unterscheiden zu können. Sie sollten allerdings im allgemeinen von dieser Möglichkeit keinen Gebrauch machen, da der entstehende Quelltext schwierig zu pflegen ist:

```
//: initialization/OverloadingOrder.java
// Overloading based on the order of the arguments.
import static net.mindview.util.Print.*;

public class OverloadingOrder {
    static void f(String s, int i) {
        print("String: " + s + ", int: " + i);
    }
    static void f(int i, String s) {
        print("int: " + i + ", String: " + s);
    }
    public static void main(String[] args) {
        f("String first", 11);
        f(99, "Int first");
    }
} /* Output:
      String: String first, int: 11
      int: 99, String: Int first
   *///:~
```

Beide Versionen der Methode `f()` haben identische Parameter, aber in verschiedener Reihenfolge und dies ist zugleich der einzige Unterschied zwischen beiden Versionen.

6.2.2 Implizite und explizite Typumwandlungen bei Überladung mit Parametern primitiven Typs

[21–22] Ein Wert primitiven Typs kann automatisch von einem kleineren Typ in einen größeren umgewandelt werden. Dieser Mechanismus verursacht in Kombination mit der Methodenüberladung verwirrende Effekte. Das folgende Beispiel zeigt, was geschieht, wenn einer überladenen Methode ein Argument primitiven Typs übergeben wird:

```
//: initialization/PrimitiveOverloading.java
// Promotion of primitives and overloading.
import static net.mindview.util.Print.*;

public class PrimitiveOverloading {
```

```
void f1(char x) { printnb("f1(char) "); }
void f1(byte x) { printnb("f1(byte) "); }
void f1(short x) { printnb("f1(short) "); }
void f1(int x) { printnb("f1(int) "); }
void f1(long x) { printnb("f1(long) "); }
void f1(float x) { printnb("f1(float) "); }
void f1(double x) { printnb("f1(double) "); }

void f2(byte x) { printnb("f2(byte) "); }
void f2(short x) { printnb("f2(short) "); }
void f2(int x) { printnb("f2(int) "); }
void f2(long x) { printnb("f2(long) "); }
void f2(float x) { printnb("f2(float) "); }
void f2(double x) { printnb("f2(double) "); }

void f3(short x) { printnb("f3(short) "); }
void f3(int x) { printnb("f3(int) "); }
void f3(long x) { printnb("f3(long) "); }
void f3(float x) { printnb("f3(float) "); }
void f3(double x) { printnb("f3(double) "); }

void f4(int x) { printnb("f4(int) "); }
void f4(long x) { printnb("f4(long) "); }
void f4(float x) { printnb("f4(float) "); }
void f4(double x) { printnb("f4(double) "); }

void f5(long x) { printnb("f5(long) "); }
void f5(float x) { printnb("f5(float) "); }
void f5(double x) { printnb("f5(double) "); }

void f6(float x) { printnb("f6(float) "); }
void f6(double x) { printnb("f6(double) "); }

void f7(double x) { printnb("f7(double) "); }

void testConstVal() {
    printnb("5: ");
    f1(5);f2(5);f3(5);f4(5);f5(5);f6(5);f7(5); print();
}
void testChar() {
    char x = 'x';
    printnb("char: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testByte() {
    byte x = 0;
    printnb("byte: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testShort() {
    short x = 0;
    printnb("short: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testInt() {
    int x = 0;
    printnb("int: ");
    f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
}
void testLong() {
    long x = 0;
```

```
        printnb("long: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
    }
    void testFloat() {
        float x = 0;
        printnb("float: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
    }
    void testDouble() {
        double x = 0;
        printnb("double: ");
        f1(x);f2(x);f3(x);f4(x);f5(x);f6(x);f7(x); print();
    }
    public static void main(String[] args) {
        PrimitiveOverloading p = new PrimitiveOverloading();
        p.testConstVal();
        p.testChar();
        p.testByte();
        p.testShort();
        p.testInt();
        p.testLong();
        p.testFloat();
        p.testDouble();
    }
} /* Output:
      5: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
      char: f1(char) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
      byte: f1(byte) f2(byte) f3(short) f4(int) f5(long) f6(float) f7(double)
      short: f1(short) f2(short) f3(short) f4(int) f5(long) f6(float) f7(double)
      int: f1(int) f2(int) f3(int) f4(int) f5(long) f6(float) f7(double)
      long: f1(long) f2(long) f3(long) f4(long) f5(long) f6(float) f7(double)
      float: f1(float) f2(float) f3(float) f4(float) f5(float) f6(float) f7(double)
      double: f1(double) f2(double) f3(double) f4(double) f5(double) f6(double) f7(double)
   *///:~
```

Die Konstante 5 wird als `int`-Wert behandelt. Hat die überladene `fX()`-Methode eine `int`-Version, so wird diese gewählt. In allen anderen Fällen wird ein Argumenttyp der kleiner ist als der Parametertyp der Methode, in den größeren Typ umgewandelt. Der primitive Typ `char` verhält sich abweichend von dieser Regel: Ist keine Version mit einem `char`-Parameter vorhanden, so wird `char` in `int` umgewandelt.

[23] Was geschieht, wenn der Argumenttyp größer ist als der Parametertyp? Eine geänderte Fassung des vorigen Programms gibt Auskunft:

```
//: initialization/Demotion.java
// Demotion of primitives and overloading.
import static net.mindview.util.Print.*;

public class Demotion {
    void f1(char x) { print("f1(char)"); }
    void f1(byte x) { print("f1(byte)"); }
    void f1(short x) { print("f1(short)"); }
    void f1(int x) { print("f1(int)"); }
    void f1(long x) { print("f1(long)"); }
    void f1(float x) { print("f1(float)"); }
    void f1(double x) { print("f1(double)"); }

    void f2(char x) { print("f2(char)"); }
```

```
    void f2(byte x) { print(''f2(byte)''); }
    void f2(short x) { print(''f2(short)''); }
    void f2(int x) { print(''f2(int)''); }
    void f2(long x) { print(''f2(long)''); }
    void f2(float x) { print(''f2(float)''); }

    void f3(char x) { print(''f3(char)''); }
    void f3(byte x) { print(''f3(byte)''); }
    void f3(short x) { print(''f3(short)''); }
    void f3(int x) { print(''f3(int)''); }
    void f3(long x) { print(''f3(long)''); }

    void f4(char x) { print(''f4(char)''); }
    void f4(byte x) { print(''f4(byte)''); }
    void f4(short x) { print(''f4(short)''); }
    void f4(int x) { print(''f4(int)''); }

    void f5(char x) { print(''f5(char)''); }
    void f5(byte x) { print(''f5(byte)''); }
    void f5(short x) { print(''f5(short)''); }

    void f6(char x) { print(''f6(char)''); }
    void f6(byte x) { print(''f6(byte)''); }

    void f7(char x) { print(''f7(char)''); }

    void testDouble() {
        double x = 0;
        print(''double argument:'');
        f1(x);f2((float)x);f3((long)x);f4((int)x);
        f5((short)x);f6((byte)x);f7((char)x);
    }
    public static void main(String[] args) {
        Demotion p = new Demotion();
        p.testDouble();
    }
} /* Output:
      double argument:
      f1(double)
      f2(float)
      f3(long)
      f4(int)
      f5(short)
      f6(byte)
      f7(char)
   *////:~
```

In diesem Beispiel sind die Parametertypen der Methoden kleiner als die Argumenttypen. Ist der Argumenttyp größer als der Parametertyp, so müssen Sie explizit eine verkleinernde Typumwandlung vornehmen. Wenn Sie diese Typumwandlung auslassen, meldet der Compiler einen Fehler.

6.2.3 „Überladung“ bezüglich des Rückgabetyps

[24–25] Häufig tritt die Frage auf, warum sich die Unterscheidung der Versionen einer überladenen Methode nach dem Klassennamen (bei Konstruktoren) und der Parameterliste richtet, nicht aber nach dem Rückgabetyp. Die beiden folgenden „Versionen“ der Methode `f()` haben zwar übereinstimmende Namen und Parameterlisten, sind aber am Rückgabetyp mühelos unterscheidbar:

```
void f() {}
```

```
int f() { return 1; }
```

Das könnte funktionieren, solange der Compiler die gewünschte „Version“ aus dem Kontext herleiten kann, etwa bei `int x = f()`. Sie können eine Methode allerdings auch aufrufen, ohne den Rückgabewert zu beachten. Man sagt in diesem Fall, *die Methode werde wegen ihres Seiteneffektes aufgerufen*, da Sie sich nicht um den Rückgabewert kümmern, sondern die übrigen Auswirkungen des Methodenaufrufs hervorrufen wollen. Wenn Sie die `f()`-Methode wie folgt aufrufen

```
f();
```

kann der Compiler nicht ermitteln, welche „Version“ aufgerufen werden soll. Woher soll es ein Programmierer wissen, der den Quelltext liest? Dies ist der Grund, warum die Versionen einer überladenen Methoden *nicht* ausschließlich nach Rückgabetyp unterschieden werden können.

6.3 Standardkonstruktoren (parameterlose Konstruktoren)

[26] Ein Standardkonstruktor ist einer parameterloser Konstruktor und wird verwendet, um „Standardobjekte“ zu erzeugen. Wenn Sie eine Klasse schreiben und keinen Konstruktor definieren, so erzeugt der Compiler automatisch einen Standardkonstruktor für Sie. Ein Beispiel:

```
//: initialization/DefaultConstructor.java
class Bird {}

public class DefaultConstructor {
    public static void main(String[] args) {
        Bird b = new Bird(); // Default!
    }
} ///:~
```

Der Ausdruck

```
new Bird();
```

erzeugt ein neues Objekt und ruft den Standardkonstruktor auf, obwohl die Klasse keine explizite Definition eines solchen Konstruktors beinhaltet. Ohne Standardkonstruktor hätten Sie keine Methode zur Verfügung, die Sie aufrufen können, um ein Objekt zu erzeugen. Wenn Sie aber einen Konstruktor definieren (mit oder ohne Parameter), erzeugt der Compiler *keinen* weiteren Konstruktor:

```
//: initialization/NoSynthesis.java
class Bird2 {
    Bird2(int i) {}
    Bird2(double d) {}
}

public class NoSynthesis {
    public static void main(String[] args) {
        //! Bird2 b = new Bird2(); // No default
        Bird2 b2 = new Bird2(1);
        Bird2 b3 = new Bird2(1.0);
    }
} ///:~
```

Erreicht der Compiler der Ausdruck

```
new Bird2();
```

so meldet er, daß er keinen passenden Konstruktor finden kann. Wenn Sie keinen Konstruktor definieren, sagt der Compiler gewissermaßen: „Sie brauchen einen *irgendeinen* Konstruktor. Ich lege einen für Sie an.“ Wenn Sie dagegen einen Konstruktor definieren, denkt sich der Compiler: „Sie haben einen Konstruktor geschrieben, wissen also was Sie tun. Falls Sie keinen Standardkonstruktor angelegt haben, lassen Sie ihn bewußt aus.“

Übungsaufgabe 3: (1) Schreiben Sie eine Klasse mit einem Standardkonstruktor (einem parameterlosen Konstruktor), der eine Meldung ausgibt. Erzeugen Sie ein Objekt Ihrer Klasse. ■

Übungsaufgabe 4: (1) Überladen Sie den Konstruktor aus Übungsaufgabe 1 um eine Version, die ein `String`-Argument erwartet und sein Argument zusammen mit der Meldung ausgibt. ■

Übungsaufgabe 5: (2) Schreiben Sie eine Klasse `Dog` mit einer überladenen `bark()`-Methode. Die Überladung bezieht sich auf verschiedene primitive Typen und jede Version der `bark()`-Methode gibt eine Meldung mit einem anderen Hundelaut aus, zum Beispiel „Bellen“, „Jaulen“ und so weiter. Legen Sie eine `main()`-Methode an, die die verschiedenen Versionen aufruft. ■

Übungsaufgabe 6: (1) Ändern Sie Übungsaufgabe 5, so daß zwei Versionen der überladenen `bark()`-Methode zwei Parameter unterschiedlichen Typs in entgegengesetzter Reihenfolge definieren. Verifizieren Sie, daß Sie eine gültige Überladung definiert haben. ■

Übungsaufgabe 7: (1) Schreiben Sie eine Klasse ohne Konstruktor und erzeugen Sie in der `main()`-Methode ein Objekt dieser Klasse, um zu verifizieren, daß der Standardkonstruktor automatisch erzeugt wird. ■

6.4 Die Selbstreferenz this

[27] Angenommen, die beiden Referenzvariablen `a` und `b` verweisen auf zwei verschiedene Objekte der Klasse `Banana`. You might wonder how it is that you can call a method peel() for both those objects:

```
//: initialization/BananaPeel.java
class Banana { void peel(int i) { /* ... */ } }

public class BananaPeel {
    public static void main(String[] args) {
        Banana a = new Banana(),
            b = new Banana();
        a.peel(1);
        b.peel(2);
    }
} ///:~
```

Wenn es nur eine `peel()`-Methode gibt, woher „weiß“ diese Methode dann, ob sie vom Objekt `a` oder `b` aufgerufen wurde?

[28] Der Compiler ergänzt den Methodenaufruf hinter den Kulissen, um Ihnen eine komfortable objektorientierte Syntax zu ermöglichen, über die Sie „einem Objekt eine Nachricht senden“. Die `peel()`-Methode bekommt ein verborgenes erstes Argument, nämlich die Referenz auf das manipulierte Objekt. Die beiden Methodenaufrufe haben in etwa das folgende Format:

```
Banana.peel(a, 1);
Banana.peel(b, 2);
```

Diese Notation wird aber nur intern verwendet. Sie können diese Schreibweise nicht im Quelltext verwenden und den Compiler überreden, sie zu akzeptieren. Das Format vermittelt aber einen Eindruck von den Geschehnissen.

[29] Stellen Sie sich vor, daß Sie an der Implementierung einer Methode arbeiten und die Referenz auf das aktuelle Objekt brauchen. Da diese Referenz vom Compiler im Verborgenen übergeben wird, haben Sie keine entsprechende lokale Variable zur Verfügung. Zu diesem Zweck gibt es aber ein Schlüsselwort: Die Selbstreferenz `this`. Das Schlüsselwort `this` darf nur im Körper einer nicht-statischen Methode verwendet werden und liefert die Referenz das Objekt zu dem die Methode gehört. Ruft eine Methode eine andere Methode derselben Klasse auf, so ist `this` nicht erforderlich, das heißt es genügt, die zweitere Methode ohne den Präfix „`this.`“ aufzurufen. Die aktuelle Selbstreferenz `this` wird automatisch auch in der aufgerufenen Methode verwendet:

```
//: initialization/Apricot.java
public class Apricot {
    void pick() { /* ... */ }
    void pit() { pick(); /* ... */ }
} ///:~
```

[30] Sie könnten die `pick()`-Methode im Körper der `pit()`-Methode auch per `this.pick()` aufrufen, der Bezug auf `this` ist aber nicht notwendig,[1] da er vom Compiler automatisch ergänzt wird. Das Schlüsselwort `this` wird nur in Spezialfällen explizit verwendet, beispielsweise wenn Sie über eine `return`-Anweisung die Referenz auf das aktuelle Objekt zurückgeben wollen:

```
//: initialization/Leaf.java
// Simple use of the ''this'' keyword.

public class Leaf {
    int i = 0;
    Leaf increment() {
        i++;
        return this;
    }
    void print() {
        System.out.println(''i = '' + i);
    }
    public static void main(String[] args) {
        Leaf x = new Leaf();
        x.increment().increment().increment().print();
    }
} /* Output:
      i = 3
   *///:~
```

Da die Methode `increment()` per `this` die Referenz auf das aktuelle `Leaf`-Objekt zurückgibt, können mehrere Operationen auf demselben Objekt verarbeitet werden.

[31] Die Selbstreferenz `this` ist außerdem nützlich, um die Referenz auf das aktuelle Objekt einer Methode eines anderen Objektes zu übergeben:

```
//: initialization/PassingThis.java
class Person {
    public void eat(Apple apple) {
```

[1] Es gibt Programmierer, die wie besessen vor jedem Methodenaufruf und jedem Feldnamen die `this`-Referenz anbringen, weil die Syntax dadurch „klarer und expliziter wird“. Lassen Sie es sein. Wir verwenden Hochsprachen, weil sie gewisse Dinge für uns erledigen. Wenn Sie `this` an einer Stelle unnötig einsetzen, verwirren und verärgern Sie jeden, der Ihren Quelltext liest, da die Selbstreferenz im restlichen Quelltext nicht vorkommt. Die Leute erwarten, daß `this` nur auftritt, wenn es notwendig ist. Sie sparen Zeit und Geld, wenn Sie einen geradlinigen Stil einhalten.

```
        Apple peeled = apple.getPeeled();
        System.out.println("Yummy");
    }
}

class Peeler {
    static Apple peel(Apple apple) {
        // ... remove peel
        return apple; // Peeled
    }
}

class Apple {
    Apple getPeeled() { return Peeler.peel(this); }
}

public class PassingThis {
    public static void main(String[] args) {
        new Person().eat(new Apple());
    }
} /* Output:
      Yummy
   *///:~
```

[32] Die Klasse `Apple` muß die statische `Peeler`-Methode `peel()` aufrufen. `peel()` ist eine Hilfsmethode und führt eine Operation aus, die außerhalb der Klasse `Apple` definiert ist (vielleicht läßt sich die externe Methode auf viele verschiedene Klassen anwenden und Sie wollen die Anweisungen nicht kopieren). Sie müssen `this` verwenden, damit das `Apple`-Objekt die Referenz auf sich selbst an die `peel()`-Methode übergeben kann.

Übungsaufgabe 8: (1) Schreiben Sie eine Klasse mit zwei Methoden. Rufen Sie die zweite Methode aus der ersten Methode zweimal auf, einmal mit `this` und einmal ohne, nur um zu sehen, das es funktioniert. Sie sollten die Variante mit `this` in der Praxis aber nicht verwenden. ■

6.4.1 Verkettung von Konstruktoraufrufen

[33] Hat eine Klasse mehrere Konstruktoren, so kommt es gelegentlich vor, daß Sie einen Konstruktor von einem anderen aus aufrufen wollen, um das Kopieren von Quelltext zu vermeiden. Ein solcher Aufruf läßt sich per `this` bewerkstelligen.

[34–35] Im allgemeinen bedeutet `this` „dieses Objekt“ oder „das aktuelle Objekt“ und liefert die Referenz auf das aktuelle Objekt. In einem Konstruktor ändert sich die Bedeutung von `this`, wenn dem Schlüsselwort eine Liste von Argumenten folgt. Unter diesen Bedingungen bewirkt `this` den Aufruf des Konstruktors mit der entsprechenden Parameterliste. Sie haben also eine einfache Möglichkeit zur Verfügung, um andere Konstruktoren aufzurufen:

```
//: initialization/Flower.java
// Calling constructors with "this"
import static net.mindview.util.Print.*;

public class Flower {
    int petalCount = 0;
    String s = "initial value";
    Flower(int petals) {
        petalCount = petals;
        print("Constructor w/ int arg only, petalCount= " + petalCount);
    }
```

```
    Flower(String ss) {
        print("Constructor w/ String arg only, s = " + ss);
        s = ss;
    }
    Flower(String s, int petals) {
        this(petals);
        //! this(s); // Can't call two!
        this.s = s; // Another use of "this"
        print("String & int args");
    }
    Flower() {
        this("hi", 47);
        print("default constructor (no args)");
    }
    void printPetalCount() {
        //! this(11); // Not inside non-constructor!
        print("petalCount = " + petalCount + " s = "+ s);
    }
    public static void main(String[] args) {
        Flower x = new Flower();
        x.printPetalCount();
    }
} /* Output:
      Constructor w/ int arg only, petalCount= 47
      String & int args
      default constructor (no args)
      petalCount = 47 s = hi
   *///:~
```

Der Konstruktor `Flower(String s, int petals)` zeigt, daß Sie über `this` *einen* anderen Konstruktor aufrufen können, allerdings nicht mehr als einen. Außerdem muß der Aufruf eines weiteren Konstruktors die erste Anweisung im Körper eines Konstruktors sein, andernfalls gibt der Compiler eine Fehlermeldung aus.

[36–37] Das Beispiel zeigt noch eine weitere Anwendung der `this`-Referenz. Das Argument `s` des Konstruktors und das Feld `s` haben denselben Namen, `s` ist also nicht eindeutig. Das Problem läßt sich lösen, indem Sie sich per `this.s` auf das Feld beziehen. Die finden diese Notation in vielen Java-Quelltexten und vielen Beispielen in diesem Buch. In der Methode `printPetalCount()` sehen Sie,daß der Compiler den Aufruf eines Konstruktors nur aus einem Konstruktor gestattet, nicht aber aus einer gewöhnlichen Methode.

Übungsaufgabe 9: (1) Schreiben Sie eine Klasse mit zwei Konstruktoren. Rufen Sie den zweiten Konstruktor per `this` aus dem ersten Konstruktor auf. ■

6.4.2 Die Bedeutung des static-Modifikators

[38] Im Hinblick auf die Eigenschaften und Fähigkeiten der Selbstreferenz `this`, haben Sie nun ein besseres Verständnis für die Wirkung des `static`-Modifikators bei Methoden. Die `this`-Referenz steht in einer statischen Methode nicht zur Verfügung. Sie können in einer statischen Methode nicht ohne weiteres eine nicht-statische Methode aufrufen[2] (obwohl die Umkehrung möglich ist). Anderer-

[2] Das Aufrufen einer nicht-statischen Methode aus einer statischen Methode kann durch Übergabe einer Referenz auf ein entsprechendes Objekt an die statische Methode ermöglicht werden (die statische Methode könnte ein solches Objekt auch selbst erzeugen). Sie können über diese Referenz nicht-statische Methoden aufrufen und nicht-statische Felder erreichen. Andererseits definieren Sie in einem solchen Fall typischerweise eine gewöhnliche nicht-statische Methode.

seits „genügt die Klasse“, um eine statische Methode aufzurufen, das heißt Sie brauchen kein Objekt der Klasse, in der die statische Methode definiert ist. Dies ist der eigentliche Verwendungszweck einer statischen Methode. Statische Methoden sind das Äquivalent von globalen Methoden, die aber bei Java nicht erlaubt sind. Die Definition einer statischen Methode in einer Klasse gestattet dieser Klasse Zugriff auf andere statische Methoden und statische Felder.

[39] Manche Programmierer betrachten statische Methoden als nicht objektorientiert, da sie semantisch globalen Methoden entsprechen. Eine statische Methode sendet keine Nachricht an ein Objekt, da die Selbstreferenz `this` nicht vorhanden ist. Dieses Argument ist nicht von der Hand zu weisen und wenn Sie sich dabei ertappen, daß Sie viele statische Methoden angelegt haben, sollten Sie Ihren Ansatz wohl noch einmal überdenken. Andererseits sind statische Methoden anwendungsbezogen und es gibt Situationen, in denen sie tatsächlich notwendig sind. Die Frage, ob statische Methoden zur Objektorientierung passen oder nicht, sollte den Theoretikern überlassen werden.

6.5 Aufräumen: Finalisierung und automatische Speicherbereinigung

[40] Programmierer wissen um die Wichtigkeit der Initialisierung, übersehen aber häufig die Bedeutsamkeit des Aufräumens. Wer kümmert sich schließlich schon darum, ein `int`-Feld aufzuräumen? Bei Bibliotheken ist das Ignorieren eines Objektes nach Gebrauch allerdings nicht immer sicher. Java verfügt zwar über eine automatische Speicherbereinigung, die den von nicht mehr benötigten Objekten beanspruchten Arbeitsspeicher wieder zurückfordert, aber stellen Sie sich die folgende (seltene) Situation vor: Ein Objekt allokiert „besonderen“ Arbeitsspeicher, ohne den `new`-Operator zu verwenden. Die automatische Speicherbereinigung kann nur per `new` allokierten Arbeitsspeicher wieder freigeben und ist demnach nicht in der Lage, den „besonderen“ Arbeitsspeicher wieder einzufordern. Java behandelt diese Situation dadurch, daß Sie in Ihrer Klasse eine Methode namens `finalize()` definieren können. Der Finalisierungsmechanismus *soll folgendermaßen funktionieren*: Ist die automatische Speicherbereinigung bereit, um den von Ihrem Objekt beanspruchten Arbeitsspeicher freizugeben, so wird zuerst die `finalize()`-Methode aufgerufen. Erst beim nächsten Durchgang der Speicherbereinigung wird der beanspruchte Arbeitsspeicher zurückverlangt. Die `finalize()`-Methode gestattet also die Durchführung von Aufräumarbeiten *zum Zeitpunkt der automatischen Speicherbereinigung*.

[41] Dies ist eine potentielle Falle für Programmierer, vor allem für C++-Programmierer, die `finalize()` zunächst irrtümlich für das Äquivalent des Destruktors in C++ halten, also eine Funktion die beim Zerstören eines Objektes *stets* aufgerufen wird. Es ist wichtig, in diesem Punkt zwischen C++ und Java zu unterscheiden: Bei einem fehlerlosen Programm in C++ werden Objekte *stets zerstört*, während Java-Objekte *nicht immer* von der automatischen Speicherbereinigung aufgeräumt werden. Mit anderen Worten:

1. Java-Objekte werden nicht immer aufgeräumt.
2. Automatische Speicherbereinigung ist nicht gleich Objektzerstörung.

[42] Wenn Sie sich diese beiden Punkte merken, bekommen Sie keine Schwierigkeiten. Müssen Anweisungen ausgeführt werden, bevor Sie das Objekt nicht mehr brauchen, dann sind Sie selbst dafür zuständig, sich um die Verarbeitung dieser Anweisungen zu kümmern. Da Java nicht über Destruktoren oder ein ähnliches Konzept verfügt, müssen Sie zum Aufräumen eine gewöhnliche Methode anlegen. Stellen Sie beispielsweise ein Objekt vor, das bei seiner Erzeugung eine Graphik auf den Bildschirm zeichnet. Wenn Sie die Abbildung nicht explizit vom Bildschirm entfernen, wird sie eventuell überhaupt nicht gelöscht. Auch wenn Sie in der `finalize()`-Methode Löschfunktionalität

implementieren, wird die Graphik auf dem Bildschirm nur entfernt, falls das Objekt von der automatischen Speicherbereinigung aufgeräumt und seine `finalize()`-Methode aufgerufen wird (das Aufräumen des Objektes ist allerdings keinesfalls garantiert). Andernfalls bleibt die Graphik erhalten.

[43] Es ist möglich, daß der von einem Objekt beanspruchte Arbeitsspeicher überhaupt nicht freigegeben wird, weil das Programm den Punkt nicht erreicht, an dem der Arbeitsspeicher knapp wird. Ist das Programm beendet und hat die automatische Speicherbereinigung den Arbeitsspeicher keines der beteiligten Objekte freigegeben, so wird der gesamte vom Programm beanspruchte Speicher „am Stück“ an das Betriebssystem zurückgegeben. Das ist eine günstige Eigenschaft, da die Speicherbereinigung stets etwas Aufwand bedeutet. Wird die automatische Speicherbereinigung nicht aufgerufen, so müssen Sie diese Unkosten nicht tragen.

6.5.1 Die Aufgabe der finalize()-Methode

[44] Welche Aufgabe hat die `finalize()`-Methode, wenn Sie sie nicht als universelle Aufräummethode verwenden sollen? Merken Sie sich noch einen dritten Punkt:

3. Die automatische Speicherbereinigung betrifft nur den Arbeitsspeicher.

Die automatische Speicherbereinigung existiert nur aus einem einzigen Grund: Um Arbeitsspeicher freizugeben, den Ihr Programm nicht mehr braucht. Alle mit der automatischen Speicherbereinigung verbundene Aktivität, vor allem die `finalize()`-Methode darf nur den Arbeitsspeicher und seine Freigabe betreffen.

[46] Sollen Sie, wenn Ihr Objekt weitere Objekte beinhaltet (referenziert) diese Objekte per `finalize()` explizit freigeben? Nein. Die automatische Speicherbereinigung kümmert sich darum, den von sämtlichen Objekten beanspruchten Speicherplatz wieder freizugeben, unabhängig davon, wie die einzelnen Objekte erzeugt wurden. Es zeigt sich, daß die `finalize()`-Methode nur in Spezialfällen notwendig ist, bei denen ein Objekt auf eine andere Weise als die Objekterzeugung Arbeitsspeicher allokiert. Andererseits ist bei Java alles ein Objekt. Wie paßt das zusammen?

[47] Der Finalisierungsmechanismus ist auch im Hinblick auf die Möglichkeit vorhanden, daß Sie „etwas C-ähnliches tun“ und dabei auf eine Weise Arbeitsspeicher allokieren, die nicht dem Standardverfahren von Java entspricht. Dies geschieht hauptsächlich beim Aufrufen *nativer Methoden*, mit deren Hilfe Sie Code, der nicht in Java geschrieben ist, von Java aus aufrufen können. (Anhang B der zweiten Auflage der englischen Originalausgabe dieses Buches behandelt native Methoden; siehe *http://www.mindview.net*.) C und C++ sind gegenwärtig die einzigen von den nativen Methoden unterstützten Sprachen, aber da native Methoden Unterprogramme in anderen Sprachen aufrufen, können Sie effektiv alles aufrufen. In einem solchen, nicht in Java geschriebenen Code, könnten zum Beispiel die `malloc()`-Funktionen von C Speicher allokieren, der erst bei Aufrufen von `free()` wieder freigegeben wird, so daß ein Speicherleck entstehen kann. `free()` ist eine Funktion von C und C++, müßte also von der nativen Methode in `finalize()` aufgerufen werden.

[48] Nachdem Sie Abschnitt 6.5 bis zu dieser Stelle gelesen haben, sind Sie wahrscheinlich zu der Einschätzung gekommen, daß Sie `finalize()` nicht oft verwenden werden.[3] Das ist richtig. Die `finalize()`-Methode ist keine passende Stelle für normale Aufräumanweisungen. Wo also sollen solche Anweisungen platziert werden?

[3] Joshua Bloch drückt sich im Abschnitt *Avoid Finalisers*, Seite 20, seines Buches *Effective Java Language Programming Guide* Addison-Wesley (2001), noch deutlicher aus: „Finalizers are unpredictable, often dangerous, and generally unnecessary.“ Übersetzt etwa: „Die Ausführung von `finalize()`-Methoden ist nicht vorhersagbar, häufig gefährlich und im allgemeinen unnötig.“

6.5.2 Sie müssen selbst aufräumen

[49] Um ein Objekt aufzuräumen, muß der Programmierer, der das Objekt benutzt, an der Stelle an der das Aufräumen erwünscht ist, eine entsprechende Methode aufrufen. Das klingt einfach, kollidiert aber zum Teil mit dem Konzept des Destruktors aus C++. Bei C++ werden alle Objekte zerstört, oder eher, alle Objekte *sollten* zerstört werden. Wird bei C++ ein Objekt *created as a local* (das heißt auf dem *Stack/Aufrufstapel*, bei Java nicht möglich), dann wird die Zerstörung bei der schließenden geschweiften Klammer des Geltungsbereiches eingeleitet, in dem das Objekt erzeugt wurde. Wird das Objekt per `new` erzeugt, wie bei Java, so erfolgt der Destruktoraufruf, wenn der Programmier den Operator `delete` aufruft (bei Java nicht vorhanden). Versäumt der C++-Programmierer den `delete`-Aufruf, so wird der Destruktor nicht aufgerufen, es entsteht ein Speicherleck und die übrigen Teile des Objektes werden nicht aufgeräumt. Fehler dieser Art können sehr schwer zu finden sein und sind einer der unwiderlichen Anreize, um Java anstelle von C++ zu wählen.

[50] Java gestattet im Gegensatz zu C++ keine *lokalen Objekte*, sondern verlangt, daß Objekte stets per `new`-Operator erzeugt werden. Java hat keinen `delete`-Operator, weil sich die automatische Speicherbereinigung um die Freigabe des beanspruchten Arbeitsspeichers kümmert. Grob vereinfachend können Sie also sagen, daß Java aufgrund der automatischen Speicherbereinigung keine Destruktoren hat. Sie werden andererseits in diesem Buch noch erkennen, daß die Präsenz einer automatischen Speicherbereinigung den Bedarf nach beziehungsweise den Nutzen von Destruktoren keinesfalls beseitigt. (Sie sollten die `finalize()`-Methode niemals direkt aufrufen, so daß auch diese Möglichkeit ausscheidet.) Sind Aufräumarbeiten notwendig, die nichts mit der Freigabe von Arbeitsspeicher zu tun haben, müssen Sie bei Java nach wie vor eine passende Aufräummethode aufrufen, das Äquivalent des Destruktors bei C++, aber ohne den damit verbundenen Komfort.

[51] Denken Sie daran, daß weder die automatische Speicherbereinigung noch der Finalisierungsmechanismus garantiert ablaufen. Ist der Arbeitsspeicher nicht knapp, so investiert die Laufzeitumgebung keine Zeit, um per Speicherbereinigung Arbeitsspeicher freizugeben.

6.5.3 Anwendungsbeispiel für die finalize()-Methode: Die Terminierungsvoraussetzung

[52] Im allgemeinen können Sie sich nicht darauf verlassen, daß `finalize()` aufgerufen wird und müssen separate Methoden zum Aufräumen definieren und explizit aufrufen. Die `finalize()`-Methode scheint nur in undurchsichtigen Situationen bei der Freigabe von Arbeitsspeicher nützlich zu sein, die sich bei den meisten Programmen niemals einstellen. Es gibt aber einen interessanten Anwendungsfall für `finalize()`, der nicht darauf angewiesen ist, das die Methode jedesmal aufgerufen wird: Das Prüfen einer *Terminierungsvoraussetzung*[4] für Objekte.

[53] Ab dem Punkt, von dem an Sie sich nicht mehr für ein Objekt interessieren, das heißt wenn das Objekt bereit ist, um aufgeräumt zu werden, sollte es einen Zustand angenommen haben, in dem der von ihm beanspruchte Arbeitsspeicher gefahrlos wieder freigegeben werden kann. Repräsentiert das Objekt beispielsweise eine geöffnete Datei, so sollte der Programmierer diese Datei schließen, bevor das Objekt der automatischen Speicherbereinigung übergeben wird. Wird ein Teil des Objektes nicht richtig aufgeräumt, so hat das Programm einen Fehler, der unter Umständen sehr schwer zu finden ist. Die `finalize()`-Methode kann, auch wenn sie nicht zuverlässig aufgerufen wird, verwendet werden, um letztendlich festzustellen, ob diese Voraussetzung erfüllt ist.

[4] Dieser Begriff, in der englischen Ausgabe „termination condition“, stammt von Bill Venners (*http://www.artima.com*) aus einer Schulung, die wir gemeinsam abgehalten haben.

[54–55] Ein einfaches Beispiel für die Anwendung der `finalize()`-Methode, um eine Terminierungsvoraussetzung zu prüfen:

```
//: initialization/TerminationCondition.java
// Using finalize() to detect an object that
// hasn't been properly cleaned up.

class Book {
    boolean checkedOut = false;
    Book(boolean checkOut) {
        checkedOut = checkOut;
    }
    void checkIn() {
        checkedOut = false;
    }
    protected void finalize() {
        if(checkedOut)
            System.out.println("Error: checked out");
        // Normally, you'll also do this:
        // super.finalize(); // Call the base-class version
    }
}

public class TerminationCondition {
    public static void main(String[] args) {
        Book novel = new Book(true);
        // Proper cleanup:
        novel.checkIn();
        // Drop the reference, forget to clean up:
        new Book(true);
        // Force garbage collection & finalization:
        System.gc();
    }
} /* Output:
      Error: checked out
   *///:~
```

Die Terminierungsvoraussetzung lautet, daß das `checkedOut`-Feld eines `Book`-Objektes auf `true` stehen muß, bevor das Objekt der automatischen Speicherbereinigung übergeben werden kann. Durch einen Programmierfehler in der `main()`-Methode enthält das `checkedOut`-Feld eines `Book`-Objektes aber den Wert `false`. Ohne Prüfung der Terminierungsvoraussetzung über die `finalize()`-Methode, wäre dieser Fehler schwer zu entdecken.

[56] Beachten Sie den Aufruf der `System`-Methode `gc()`, um die Finalisierung zu „erzwingen“. Auch wenn die automatische Speicherbereinigung nicht aktiv wird, wird das fehlerhafte `Book`-Objekt bei wiederholtem Aufrufen des Programms letztendlich entdeckt (vorausgesetzt, daß das Programm genügend Arbeitsspeicher allokiert, um die Speicherbereinigung auszulösen).

[57] Sie sollten in der Regel davon ausgehen, daß die Basisklassenversion der `finalize()`-Methode ebenfalls eine wichtige Funktion erfüllt und sie per `super` aufrufen (siehe oben). Die Basisklassenversion erfordert das Behandeln von Ausnahmen und ist ausgekommentiert, da wir die Ausnahmebehandlung noch nicht besprochen haben.

Übungsaufgabe 10: (2) Schreiben Sie eine Klasse mit einer `finalize()`-Methode, die eine Meldung ausgibt. Erzeugen Sie in der `main()`-Methode ein Objekt Ihrer Klasse und erläutern Sie das Verhalten des Programms. ■

Übungsaufgabe 11: (4) Ändern Sie Übungsaufgabe 10, so daß die `finalize()`-Methode immer

aufgerufen wird. ■

Übungsaufgabe 12: (4) Schreiben Sie eine Klasse `Tank` mit Methoden zum Füllen und Entleeren des Behälters sowie der Terminierungsvoraussetzung, daß der Behälter entleert sein muß, bevor das Objekt aufgeräumt werden kann. Definieren Sie eine `finalize()`-Methode, die diese Terminierungsvoraussetzung prüft. Testen Sie in der `main()`-Methode die möglichen Situationen, die bei der Anwendung des Behälters eintreten können. ■

6.5.4 Die Funktionsweise der automatischen Speicherbereinigung

[58] Wenn Sie von einer Programmiersprache kommen, bei der das Allokieren von Arbeitsspeicher für Objekte auf dem Heap teuer ist, gehen Sie in natürlicher Weise davon aus, daß der Java-Ansatz, sämtlichen Arbeitsspeicher (außer bei primitiven Typen) auf dem Heap anzufordern, ebenfalls aufwändig ist. Allerdings stellt sich heraus, daß die automatische Speicherbereinigung einen deutlichen Einfluß auf die *Zunahme* der Geschwindigkeit hat, mit der Objekte erzeugt werden. Der Gedanke, daß sich die Freigabe von Arbeitsspeicher auf das Allokieren von Arbeitsspeicher auswirkt, mag auf den ersten Blick sonderbar scheinen. Manche Laufzeitumgebungen haben allerdings diese Eigenschaft und dies bedeutet, daß das Allokieren von Objekten auf dem Heap bei Java beinahe so schnell geht, als das Anfordern von Speicher für den Aufrufstapel einer Methode bei anderen Sprachen.

[59] Bei C++ können Sie sich den Heap beispielsweise wie ein Gelände vorstellen auf dem jedes Objekt sein eigenes Rasenstück absteckt. Das Grundstück kann nach einiger Zeit aufgegeben werden, wird aber wiederverwendet. Bei Java hängen die Eigenschaften des Heaps von der Laufzeitumgebung ab und unterscheiden sich deutlich von dem bei C++ gewählten Ansatz. Der Java-Heap entspricht eher einem Förderband, das jedesmal beim Erzeugen eines neuen Objektes ein Stück vorwärts bewegt wird. Das Allokieren von Arbeitsspeicher für Objekte geschieht dadurch bemerkenswert schnell. Der „Heap-Zeiger“ wird einfach vorwärts auf unberührtes Gebiet gesetzt, das heißt die Vorgehensweise entspricht effektiv dem Allokieren von Arbeitsspeicher auf dem Aufrufstapel bei C++. (Es gibt natürlich einen geringen Zusatzaufwand für die „Buchhaltung“, der aber nicht mit der Suche nach Arbeitsspeicher zu vergleichen ist.)

[60] Sie könnten einwenden, daß der Heap in Wirklichkeit kein Förderband ist und daß Sie Speicherseiten bekommen, wenn Sie ihn dennoch so behandeln und durch Auslagerung auf der Festplatte scheinbar mehr Arbeitsspeicher verwenden können, als tatsächlich zur Verfügung steht. Speicherseiten wirken sich signifikant auf die Performanz aus. Letztendlich wird der Arbeitsspeicher knapp, wenn Sie genügend Objekte erzeugen. Der Trick besteht aber darin, daß sich die automatische Speicherbereinigung einschaltet und während der Speicherfreigabe die auf dem Heap befindlichen Objekte kompaktiert, so daß der „Heap-Zeiger“ effektiv näher an den Anfang des Förderbandes und weiter von einem Seitenfehler fort bewegt wird. Die automatische Speicherbereinigung ordnet die Objekte auf dem Heap um und ermöglicht somit ein schnelles unbeschränktes Heap-Modell zum Allokieren von Arbeitsspeicher.

[61] Zum Verständnis der automatischen Speicherbereinigung bei Java ist es nützlich, zu wissen, wie ähnliche Verfahren bei anderen Sprachen funktionieren. Die Referenzzählung ist ein einfaches, aber langsames Verfahren, bei dem jedes Objekt einen Referenzzähler pflegt, der mit jedem neuen Verweis auf das Objekt inkrementiert wird. Der Zähler wird dekrementiert, wenn der Geltungsbereich einer Referenz endet oder die Referenzvariable auf `null` zurückgesetzt wird. Die Pflege des Referenzzählers ist ein geringer aber stetiger Aufwand im Lebenszyklus eines Programms. Die automatische Speicherbereinigung kontrolliert die Liste aller Objekte und gibt den beanspruchten Arbeitsspeicher frei, wenn der Zähler eines Objektes auf Null steht. (Bei der Referenzzählung wird der Arbeitsspeicher eines Objektes häufig bereits dann freigegeben, wenn sein Zähler auf Null gesetzt wird.) Ein Nachteil dieses Verfahrens zeigt sich bei verinselten zirkulär aufeinander verweisenden Objekten, die

nicht mehr verwendet werden können, aber von Null verschiedene Referenzzählerstände aufweisen. Die Referenzzählung dient häufig als Erklärungsbeispiel für die automatische Speicherbereinigung, scheint aber in keiner Implementierung der Java-Laufzeitumgebung verwendet zu werden.

[62] Schnellere Speicherbereinigungsverfahren bauen nicht auf Referenzzählung auf, sondern auf dem Ansatz, daß sich jedes „lebendige“ Objekt letztendlich bis zu einer Referenz zurückverfolgen lassen muß, die entweder auf dem Aufrufstapel oder im statischen Arbeitsspeicher liegt. Die Kette von Referenzen kann mehrere Schichten von Objekten durchdringen. Wenn Sie beim Aufrufstapel sowie im statischen Speicher beginnen und der Reihe nach allen Referenzen nachgehen, finden Sie logischerweise alle „lebendigen“ Objekte. Sie müssen jeder vorgefundenen Referenz zum zugehörigen Objekt nachgehen, alle von diesem Objekt referenzierten Objekte aufsuchen und wiederum den in diesen Objekten enthaltenen Referenzen nachgehen und so weiter bis Sie die gesamte Baumstruktur über der ursprünglichen Referenz im Aufrufstapel beziehungsweise im statischen Arbeitsspeicher erfaßt haben. Jedes Objekt auf diesem Weg muß noch „lebendig“ sein. Beachten Sie, daß abgehängte, verinselte Gruppen bei dieser Vorgehensweise unproblematisch sind, da sie einfach nicht entdeckt und somit automatisch aufgeräumt werden.

[63] Die Laufzeitumgebung implementiert bei dem in diesem Unterabschnitt beschriebenen Ansatz ein *adaptives* Speicherbereinigungsschema, das heißt die Behandlung der lokalisierten „lebendigen“ Objekte hängt vom jeweiligen Verfahren ab. Eines dieser Verfahren ist *Stop-and-Copy* und bewirkt, aus Gründen, die in Kürze verständlich werden, daß das Programm zunächst einmal angehalten wird (Stop-and-Copy ist kein im Hintergrund ablaufendes Verfahren). Anschließend wird jedes „lebendige“ Objekt vom Heap in einen zweiten Heap kopiert, so daß alle aufzuräumenden Objekte übrigbleiben. Zusätzlich werden die Objekte beim Kopieren in lückenloser Folge auf dem neuen Heap angeordnet (kompaktiert), wodurch am Ende des belegten Speicherbereichs neuer Arbeitsspeicher vergeben werden kann, wie zuvor beschrieben.

[64] Beim Verschieben eines Objektes an eine andere Speicherstelle müssen natürlich alle Referenzen auf das Objekt geändert werden. Die Referenz auf dem Heap beziehungsweise im statischen Arbeitsspeicher kann einfach aktualisiert werden, es kann aber weitere Referenzen auf das Objekt geben, die sich erst später während des Verfahrens zeigen. Diese Kandidaten werden bei Entdeckung korrigiert (Sie können sich eine Tabelle vorstellen, die die alten Speicheradressen auf die neuen abbildet).

[65] Zwei Aspekte machen Stop-and-Copy ineffizient. Einerseits benötigt das Verfahren zwei Heaps, deren Inhalte hin- und herkopiert werden und beansprucht dadurch doppelt soviel Arbeitsspeicher als Sie eigentlich brauchen. Manche Laufzeitumgebungen begegnen dieser Eigenschaft, indem Sie den Heap bedarfsorientiert blockweise allokieren und einfach einen Block in den nächsten kopieren.

[66] Der andere Aspekt betrifft den Kopiervorgang selbst. Nachdem Ihr Programm stabil geworden ist, erzeugt es womöglich nur wenig oder gar keinen „Abfall“. Dennoch kopiert Stop-and-Copy in verschwenderischer Weise den gesamten von Ihrem Programm beanspruchten Arbeitsspeicher von einer Stelle an eine andere. Um dies zu vermeiden, sind einige Laufzeitumgebungen in der Lage, zu erkennen, daß kein neuer „Abfall“ erzeugt wurde und auf ein anderes Verfahren umzuschalten (dies ist der adaptive Teil). Das alternative Verfahren heißt *Mark-and-Sweep* und wurde bei den früheren Versionen der Laufzeitumgebung von Sun Microsystems verwendet. Mark-and-Sweep ist für die generelle Verwendung zu langsam, aber wenn Sie wissen, daß nur wenig oder überhaupt kein Arbeitsspeicher freigegeben werden muß, ist es ein schneller Ansatz.

[67] Auch Mark-and-Sweep beginnt bei Aufrufstapel und im statischen Arbeitsspeicher und geht sämtlichen Referenzen nach, um die „lebendigen“ Objekte zu finden. Die „lebendigen“ Objekte werden aber noch nicht gesammelt, sondern nur durch ein Flag markiert. Der Aufräumvorgang beginnt erst, nachdem der Markierungsvorgang beendet ist. Während des Aufräumens wird der von den

unmarkierten Objekten beanspruchte Arbeitsspeicher freigegeben. Mark-and-Sweep beinhaltet keinen Kopiervorgang. Das Verfahren ordnet die Objekt ledig neu an, wenn ein fragmentierter Heap kompaktiert werden muß.

[68] Stop-and-Copy zieht in Betracht, daß diese Form der Speicherbereinigung nicht im Hintergrund stattfindet, sondern das Programm angehalten wird, um die Speicherbereinigung aufzurufen. Sie finden in der von Sun Microsystems herausgegebenen Literatur viele Hinweise auf Speicherbereinigung als Hintergrundprozeß mit geringer Priorität. Allerdings stellt sich heraus, daß die automatische Speicherbereinigung in den frühen Versionen der Laufzeitumgebung von Sun Microsystems nicht auf diese Weise implementiert war. Statt dessen hielt die Speicherbereinigung von Sun Microsystems das Programm an, wenn der Arbeitsspeicher knapp wurde. Auch Mark-und-Sweep setzt voraus, daß das Programm angehalten wird.

[69] Die hier beschriebene Laufzeitumgebung allokiert Arbeitsspeicher in großen Blöcken, wie zuvor bereits erwähnt. Wenn Sie Speicher für ein großes Objekt allokieren, bekommt das Objekt einen eigenen Block. Strenges Stop-and-Copy verlangt, daß jedes „lebendige“ Objekt vom Quell-Heap auf einen neuen Heap kopiert wird, bevor der ursprüngliche Heap aufgelöst werden kann, so daß viel Speicherinhalt übertragen werden muß. Bei Blöcken kann die Speicherbereinigung typischerweise Objekte während des Verfahrens in „tote“ Blöcke kopieren. Jeder Block verfügt über einen Generationszähler, der angibt, ob der Block noch „lebendig“ ist. Im Normalfall werden nur die seit dem letzten Durchgang der Speicherbereinigung erzeugten Blöcke kompaktiert, während die Generationszähler der übrigen Blöcke inkrementiert werden, falls es noch Verweise in diese Blöcke gibt. Dieses Verhalten behandelt den Normalfall vieler temporärer Objekte von kurzer Lebensdauer. In periodischen Abständen erfolgt ein vollständiger Bereinigungsdurchgang, wobei große Objekte noch immer nicht kopiert werden (nur ihre Generationszähler werden inkrementiert), während Blöcke mit kleinen Objekt kopiert und kompaktiert werden. Die Laufzeitumgebung beobachtet die Effizienz der Speicherbereinigung und schaltet auf Mark-and-Sweep um, wenn die Speicherbereinigung aufgrund vieler Objekte mit langer Lebensdauer zur Zeitverschwendung wird. Die Laufzeitumgebung beobachtet analog den Erfolg von Mark-and-Sweep und schaltet auf Stop-and-Copy um, wenn der Heap zu fragmentieren beginnt. Hier kommt die Adaptivität ins Spiel, [Lücke: ein halber Satz fehlt.]

[70] Es gibt mehrere Möglichkeiten für die Laufzeitumgebung, die Speicherbereinigung zu beschleunigen. Eine besonders wichtige hat mit dem Klassenlader und sogenannten *Just-in-Time* (JIT) Compilern zu tun. Ein JIT-Compiler übersetzt ein Programm teilweise oder vollständig in nativen Maschinencode, so daß es nicht von der Laufzeitumgebung interpretiert zu werden braucht und daher erheblich schneller ist. Muß eine Klasse geladen werden (typischerweise, wenn Sie das erste Objekt dieser Klasse erzeugen), so wird die Klassendatei (*.class* Datei) lokalisiert und der Bytecode der Klasse in den Arbeitsspeicher übertragen. Eine Beschleunigungsmöglichkeit besteht darin, den Bytecode mit einem JIT-Compiler zu übersetzen, hat allerdings zwei Nachteile: Der Ansatz braucht etwas mehr Zeit, die sich über den Lebenszyklus des Programms hinweg summiert und vergrößert das Volumen der ausführbaren Datei (Bytecode ist deutlich kompakter als expandierter JIT-Code). Letzteres kann zu Paging („Kachelverwaltung“) führen, wodurch das Programm definitiv verlangsamt wird. Alternativ kann der Bytecode erst bei Bedarf per JIT-Compiler übersetzt werden. Niemals ausgeführter Bytecode würde also auch nicht übersetzt werden. Die HotSpot-Technologie von Java folgt einem ähnlichen Ansatz, wobei ein Stück Bytecode schrittweise bei jeder Ausführung verbessert wird. Der Bytecode wird umso schneller, je öfter er ausgeführt wird.

6.6 Initialisierung von Feldern und lokalen Variablen

[71] Java scheut keine Mühe, um zu garantieren, daß Felder und lokale Variablen vor ihrer Verwendung korrekt initialisiert sind. Bei lokalen Variablen in Methoden drückt sich diese Garantie als

Fehlermeldung zur Übersetzungszeit aus. Das Beispiel

```
void f() {
    int i;
    i++; // Error -- i not initialized
}
```

ruft eine Fehlermeldung hervor, die besagt, daß `i` eventuell noch nicht initialisiert ist. Der Compiler hätte `i` natürlich einen Standardwert zuweisen können. Aber eine uninitialisierte lokale Variable ist wahrscheinlich ein Programmierfehler und die Zuweisung eines Standardwertes würde den Fehler überdecken. Indem der Programmierer zur Initialisierung gezwungen wird, wird der Fehler wahrscheinlicher bemerkt und behoben.

[72–74] Bei Feldern primitiven Typs liegen die Dinge dagegen anders. Wie Sie in Kapitel 3 gesehen haben, wird jedem Feld primitiven Typs ein typspezifischer Initialwert zugewiesen. Das folgende Programm verizifiert diese Aussage und zeigt die Initialwerte:

```
//: initialization/InitialValues.java
// Shows default initial values.
import static net.mindview.util.Print.*;

public class InitialValues {
    boolean t;
    char c;
    byte b;
    short s;
    int i;
    long l;
    float f;
    double d;
    InitialValues reference;
    void printInitialValues() {
        print("Data type Initial value");
        print("boolean " + t);
        print("char ['' + c + ''] ");
        print("byte " + b);
        print("short " + s);
        print("int " + i);
        print("long " + l);
        print("float " + f);
        print("double " + d);
        print("reference " + reference);
    }
    public static void main(String[] args) {
        //InitialValues iv = new InitialValues();
        //iv.printInitialValues();
        // You could also say:
        new InitialValues().printInitialValues();
        //
    }
} /* Output:
      Data type Initial value
      boolean false
      char [ ]
      byte 0
      short 0
      int 0
      long 0
```

```
        float 0.0
        double 0.0
        reference null
    *///:~
```

Obwohl keines der Felder explizit bewertet wurde, sind sie doch automatisch initialisiert (der Initialwert des Typs `char` ist Null, also ein Leerzeichen). Es besteht zumindest keine Gefahr, mit uninitialisierten Feldern zu arbeiten. Wenn Sie ein Feld nicht primitiven Typs deklarieren, ohne es mit einer Objektreferenz zu bewerten, wird das Feld mit dem speziellen Wert `null` initialisiert.

6.6.1 Statische Initialisierung von Feldern

[75–76] Welche Möglichkeiten gibt es, um ein Feld mit einem Initialwert zu versehen? Sie können den Initialwert eines Feldes einfach an der Stelle der Deklaration zuweisen. (Beachten Sie, daß dies bei C++ nicht möglich ist, obwohl Neulinge es stets versuchen.) Im folgenden Beispiel wurden die Felddeklarationen in der Klasse `InitialValues` geändert, so daß die Initialwerte gleich zugewiesen werden:

```
//: initialization/InitialValues2.java
// Providing explicit initial values.

public class InitialValues2 {
    boolean bool = true;
    char ch = 'x';
    byte b = 47;
    short s = 0xff;
    int i = 999;
    long lng = 1;
    float f = 3.14f;
    double d = 3.14159;
} ///:~
```

Auch Felder nicht primitiven Typs können auf diese Weise initialisiert werden:

```
//: initialization/Measurement.java
class Depth {}

public class Measurement {
    Depth d = new Depth();
    // ...
} ///:~
```

Ist `d` nicht initialisiert, so ist die Verwendung von `d` ein Laufzeitfehler, eine sogenannte *Ausnahme* (siehe Kapitel 13).

[77] Sie können auch den Rückgabewert einer Methode als Initialwert zuweisen:

```
//: initialization/MethodInit.java
public class MethodInit {
    int i = f();
    int f() { return 11; }
} ///:~
```

Eine solche Methode kann Argumente haben, wobei natürlich keine noch uninitialisierten Felder übergeben werden können. Erlaubt ist:

```
//: initialization/MethodInit2.java
public class MethodInit2 {
    int i = f();
```

```
        int j = g(i);
        int f() { return 11; }
        int g(int n) { return n * 10; }
    } ///:~
```

Dagegen ist nicht möglich:

```
    //: initialization/MethodInit3.java
    public class MethodInit3 {
        //! int j = g(i); // Illegal forward reference
        int i = f();
        int f() { return 11; }
        int g(int n) { return n * 10; }
    } ///:~
```

An dieser Stelle beschwert sich der Compiler zurecht über Vorwärtsreferenzierung (*forward referencing*). Dieser Fehler betrifft die Initialisierungsreihenfolge und nicht das Übersetzen des Programms.

[78] Diese Form der Initialisierung ist einfach und unkompliziert, ist aber in der Hinsicht einschränkend, daß jedes Objekt vom Typ `InitialValues` dieselben Initialwerte hat. Das entspricht gelegentlich Ihren Anforderungen, es gibt aber auch Fälle, in denen Sie mehr Flexibilität brauchen.

6.7 Dynamische Initialisierung von Feldern per Konstruktor

[79] Der Konstruktor kann zur Initialisierung von Feldern verwendet werden. Die Programmierung wird flexibler, da Sie zur Laufzeit Methoden aufrufen und Abläufe bewirken können, um die Initialwerte zu ermitteln. Merken Sie sich aber: Sie können die automatische Initialisierung nicht umgehen, die vor der Verarbeitung des Konstruktors abläuft. Im Beispiel

```
    //: initialization/Counter.java
    public class Counter {
        int i;
        Counter() { i = 7; }
        // ...
    } ///:~
```

wird das Feld `i` zunächst mit 0 vorinitialisiert und anschließend mit 7 bewertet. Dies gilt für alle Felder von primitivem Typ sowie bei Feldern für Objektreferenzen, darunter auch die Felder, die bei ihrer Deklaration initialisiert werden. Aus diesem Grund zwingt der Compiler Sie nicht, Felder im Konstruktor oder an einer anderen Stelle vor ihrer ersten Verwendung zu initialisieren. Die Initialisierung ist bereits garantiert.

6.7.1 Die Initialisierungsreihenfolge

[80–81] Die Initialisierungsreihenfolge der Felder ist einer Klasse durch die Reihenfolge ihrer Deklaration gegeben. Die Felddeklarationen können über die ganze Klasse hinweg verstreut sein, auch zwischen Methodendefinitionen, aber die Initialisierung der Felder erfolgt, bevor die erste Methode aufgerufen wird, darunter auch der Konstruktor. Ein Beispiel:

```
    //: initialization/OrderOfInitialization.java
    // Demonstrates initialization order.
    import static net.mindview.util.Print.*;

    // When the constructor is called to create a
    // Window object, you'll see a message:
```

```
class Window {
    Window(int marker) { print("Window(" + marker + ")"); }
}

class House {
    Window w1 = new Window(1); // Before constructor
    House() {
        // Show that we're in the constructor:
        print("House()");
        w3 = new Window(33); // Reinitialize w3
    }
    Window w2 = new Window(2); // After constructor
    void f() { print("f()"); }
    Window w3 = new Window(3); // At end
}

public class OrderOfInitialization {
    public static void main(String[] args) {
        House h = new House();
        h.f(); // Shows that construction is done
    }
} /* Output:
      Window(1)
      Window(2)
      Window(3)
      House()
      Window(33)
      f()
   *///:~
```

Die Deklarationen und Initialisierungen der `Window`-Felder sind absichtlich über die gesamte Klasse `House` verteilt, um zu zeigen, daß alle Felder initialisiert werden, bevor der Konstruktor aufgerufen wird oder sonst etwas geschehen kann. Das Feld `w3` wird im Konstruktor erneut initialisiert.

[82] Sie sehen an der Ausgabe, daß das `w3`-Feld zweimal initialisiert wird: einmal vor und einmal während des Konstruktoraufrufs. (Das erste `Window`-Objekt wird verworfen, so daß es später von der automatischen Speicherbereinigung aufgeräumt werden kann.) Das wirkt auf den ersten Blick nicht effizient, garantiert aber korrekte Initialisierung. Was würde geschehen, wenn eine Version eines überladenen Konstruktors das `w3`-Feld nicht initialisieren würde und keine Standardinitialisierung an der Stelle der Deklaration ausführt würde?

6.7.2 Initialisierung statischer Felder

[83–84] Zu jedem statischen Feld existiert nur eine einzige Stelle im Arbeitsspeicher, unabhängig von der Anzahl erzeugter Objekte. Der Modifikator `static` ist bei lokalen Variablen nicht erlaubt und daher nur auf Felder anwendbar. Ein statisches Feld primitiven Typs erhält seinen typspezifischen Initialwert, wenn Sie das Feld nicht bewerten. Ein statisches Feld nicht primitiven Typs wird mit dem Standardwert `null` initialisiert. Die Initialisierung eines statisches Feldes an der Stelle seiner Deklaration unterscheidet sich syntaktisch nicht von der nämlichen Situation bei einem nicht-statischen Feld.

[85] Das folgende Beispiel zeigt, *wann* ein statisches Feld initialisiert wird:

```
//: initialization/StaticInitialization.java
// Specifying initial values in a class definition.
import static net.mindview.util.Print.*;
```

```
class Bowl {
    Bowl(int marker) {
        print("Bowl(" + marker + ")");
    }
    void f1(int marker) {
        print("f1(" + marker + ")");
    }
}

class Table {
    static Bowl bowl1 = new Bowl(1);
    Table() {
        print("Table()");
        bowl2.f1(1);
    }
    void f2(int marker) {
        print("f2(" + marker + ")");
    }
    static Bowl bowl2 = new Bowl(2);
}

class Cupboard {
    Bowl bowl3 = new Bowl(3);
    static Bowl bowl4 = new Bowl(4);
    Cupboard() {
        print("Cupboard()");
        bowl4.f1(2);
    }
    void f3(int marker) {
        print("f3(" + marker + ")");
    }
    static Bowl bowl5 = new Bowl(5);
}

public class StaticInitialization {
    public static void main(String[] args) {
        print("Creating new Cupboard() in main");
        new Cupboard();
        print("Creating new Cupboard() in main");
        new Cupboard();
        table.f2(1);
        cupboard.f3(1);
    }
    static Table table = new Table();
    static Cupboard cupboard = new Cupboard();
} /* Output:
      Bowl(1)
      Bowl(2)
      Table()
      f1(1)
      Bowl(4)
      Bowl(5)
      Bowl(3)
      Cupboard()
      f1(2)
      Creating new Cupboard() in main
      Bowl(3)
      Cupboard()
      f1(2)
```

```
    Creating new Cupboard() in main
    Bowl(3)
    Cupboard()
    f1(2)
    f2(1)
    f3(1)
*///:~
```

Sie können das Erzeugen eines Objektes an der Klasse `Bowl` nachvollziehen. Die Klassen `Table` und `Cupboard` haben mehrere statische `Bowl`-Felder, verstreut über die gesamte Klassendefinition. Beachten Sie in der Klasse `Cupboard` das nicht-statische `Bowl`-Feld `bowl3` vor den statischen `Bowl`-Feldern.

[86] Sie können an der Ausgabe erkennen, daß die statischen Felder nur bei Bedarf initialisiert werden. Wenn Sie kein `Table`-Objekt erzeugen und sich nirgends auf die Felder `Table.bowl1` und `Table.bowl2` beziehen, werden die von `bowl1` und `bowl2` referenzierten `Bowl`-Objekt nicht erzeugt. Die Felder werden nur beim Erzeugen des *ersten* `Table`-Objektes (beziehungsweise beim ersten Zugriff auf eine statische Komponente dieser Klasse) initialisiert. Danach werden die statischen Felder nicht mehr reinitialisiert.

[87] In der Initialisierungsreihenfolge stehen die statischen Felder vor den nicht-statischen Feldern, falls die statischen Felder nicht bereits im Rahmen eines zuvor erzeugten Objektes bewertet wurden (siehe Ausgabe). Die Ausführung der `main()`-Methode (einer statischen Methode) setzt voraus, daß die Klasse `StaticInitialization` geladen und ihre statischen Felder `table` und `cupboard` initialisiert sind. Dies verursacht das Laden der Klassen `Table` und `Cupboard`. Da beide Klassen über statische Felder `Bowl`-Objekte referenzieren, wird auch die Klasse `Bowl` geladen. In diesem Programm werden also alle Klassen geladen, bevor die Verarbeitung der `main()`-Methode beginnt. Dies ist allerdings nicht der Regelfall. In einem typischen Programm sind nicht alle Bestandteile über statische Felder miteinander verknüpft, wie in diesem Beispiel.

[88] Wir fassen den *Vorgang der Objekterzeugung* am Beispiel einer Klasse namens `Dog` zusammen:

1. Der Konstruktor ist eine statische Methode, obwohl der `static`-Modifikator nicht explizit auftritt. Beim ersten erzeugten `Dog`-Objekt oder dem ersten Zugriff auf eine statische Methode beziehungsweise ein statisches Feld der Klasse `Dog`, durchsucht die Laufzeitumgebung den Klassenpfad nach der Klassendatei *Dog.class*.

2. Nach dem Laden der Klassendatei und dem Erzeugen des Klassenobjektes (mehr darüber in Abschnitt 15.2) werden alle statischen Felder der Klasse `Dog` initialisiert. Die statische Initialisierung findet also nur einmal statt, wenn das Klassenobjekt erstmals geladen wird.

3. Beim Aufrufen des Konstruktors der Klasse `Dog` allokiert der Objekterzeugungsvorgang zuerst ausreichenden Speicherplatz für ein `Dog`-Objekt auf dem Heap.

4. Der zugeteilte Speicherbereich wird auf binär Null zurückgesetzt. Anschließend wird jedem Feld primitiven Typs der Klasse `Dog` sein typspezifischer Initialwert zugewiesen: Bei Zahlen Null, bei `boolean` `false` und bei `char` `\u0000` (ein Leerzeichen). Felder nicht primitiven Typs werden mit `null` initialisiert.

5. Initialisierungen bei Felddeklaration werden ausgeführt.

6. Konstruktoren werden verarbeitet. In Kapitel 8 lernen Sie, daß dieser Schritt einige Aktivität mit sich bringen kann, besonders wenn Ableitung im Spiel ist.

6.7.3 Statische Initialisierungsblöcke

[89] Java gestattet die Zusammenfassung von Initialisierungsanweisungen in einem *statischen Initialisierungsblock*, zum Beispiel:

```
//: initialization/Spoon.java
public class Spoon {
    static int i;
    static {
        i = 47;
    }
} ///:~
```

[90] Der statische Initialisierungsblock ähnelt einer Methode, besteht aber lediglich aus dem Schlüsselwort `static`, gefolgt von einem Block von Anweisungen. Wie alle statischen Initialisierungen wird auch ein statischer Initialisierungsblock nur einmal ausgeführt, nämlich beim Erzeugen des ersten Objektes beziehungsweise beim ersten Zugriff auf eine statische Komponente der entsprechenden Klasse (selbst wenn Sie kein Objekt dieser Klasse erzeugen), zum Beispiel:

```
//: initialization/ExplicitStatic.java
// Explicit static initialization with the ''static'' clause.
import static net.mindview.util.Print.*;

class Cup {
    Cup(int marker) {
        print(''Cup('' + marker + '')'');
    }
    void f(int marker) {
        print(''f('' + marker + '')'');
    }
}

class Cups {
    static Cup cup1;
    static Cup cup2;
    static {
        cup1 = new Cup(1);
        cup2 = new Cup(2);
    }
    Cups() {
        print(''Cups()'');
    }
}

public class ExplicitStatic {
    public static void main(String[] args) {
        print(''Inside main()'');
        Cups.cup1.f(99); // (1)
    }
    //static Cups cups1 = new Cups(); // (2)
    //static Cups cups2 = new Cups(); // (2)
} /* Output:
      Inside main()
      Cup(1)
      Cup(2)
      f(99)
   *///:~
```

[91] Der statische Initialisierungsblock der Klasse `Cups` wird entweder beim Zugriff auf das von dem statischen Feld `cup1` referenzierte Objekt in Zeile (1) oder beim Auskommentieren von Zeile (1) und Entfernen der Kommentarzeichen vor den beiden mit (2) bezeichneten Zeilen ausgeführt. Sind (1) und (2) auskommentiert, unterbleibt die Verarbeitung des statischen Initialisierungsblocks (beachten Sie Übungsaufgabe 13). Es kommt auch nicht darauf an, ob die Kommentarzeichen nur vor einer oder beiden mit (2) bezeichneten Zeilen entfernt werden. Die statischen Felder werden nur einmal initialisiert.

Übungsaufgabe 13: (1) Verifizieren Sie die Aussagen im vorigen Absatz. ■

Übungsaufgabe 14: (1) Schreiben Sie eine Klasse mit einem statischen Feld vom Typ `String`, das bei seiner Deklaration initialisiert wird und einem weiteren `String`-Feld, das über einen statischen Initialisierungsblock bewertet wird. Legen Sie eine statische Methode an, die den Inhalt beider Felder ausgibt und zeigen Sie, daß beide Felder vor ihrer Verwendung initialisiert werden. ■

6.7.4 Dynamische Initialisierungsblöcke

[92–93] Java gestattet auch die Zusammenfassung von Initialisierungsanweisungen für die nicht statischen Felder eines Objektes in einem *dynamischen Initialisierungsblock*, zum Beispiel:

```
//: initialization/Mugs.java
// Java ''Instance Initialization.''
import static net.mindview.util.Print.*;

class Mug {
    Mug(int marker) {
        print(''Mug('' + marker + '')'');
    }
    void f(int marker) {
        print(''f('' + marker + '')'');
    }
}

public class Mugs {
    Mug mug1;
    Mug mug2;
    {
        mug1 = new Mug(1);
        mug2 = new Mug(2);
        print(''mug1 & mug2 initialized'');
    }
    Mugs() {
        print(''Mugs()'');
    }
    Mugs(int i) {
        print(''Mugs(int)'');
    }
    public static void main(String[] args) {
        print(''Inside main()'');
        new Mugs();
        print(''new Mugs() completed'');
        new Mugs(1);
        print(''new Mugs(1) completed'');
    }
} /* Output:
      Inside main()
```

```
        Mug(1)
        Mug(2)
        mug1 & mug2 initialized
        Mugs()
        new Mugs() completed
        Mug(1)
        Mug(2)
        mug1 & mug2 initialized
        Mugs(int)
        new Mugs(1) completed
    *///:~
```

Der dynamische Initialisierungsblock

```
{
    mug1 = new Mug(1);
    mug2 = new Mug(2);
    print("mug1 & mug2 initialized");
}
```

gleicht dem statischen mit Ausnahme des fehlenden `static`-Modifikators. Diese Syntax wird zur Initialisierung *anonymer innerer Klassen* (siehe Kapitel 11) benötigt, gestattet aber auch die Verarbeitung bestimmter Operationen unabhängig davon, welcher Konstruktor aufgerufen wird. Die Ausgabe zeigt, daß der dynamische Initialisierungsblock vor den Konstruktoren verarbeitet wird.

Übungsaufgabe 15: (1) Schreiben Sie eine Klasse mit einem Feld vom Typ `String`, das über einen dynamischen Initialisierungsausdruck bewertet wird. ■

6.8 Initialisierung von Arrays

[94] Ein Array ist nichts weiter, als eine unter einem Namen zusammengefaßte Folge von Objektreferenzen oder Werten primitiven Typs, die demselben Typ angehören. Arrays werden mit Hilfe des *Indizierungsoperators* `[]` definiert und verwendet. Ein Feld oder eine lokale Variable zur Aufnahme einer Arrayreferenz wird dadurch deklariert, daß dem Typ der Elemente ein Paar eckiger Klammern folgt:

```
int[] a1;
```

Das Paar eckiger Klammern darf auch nach dem Namen anstelle des Typs der Elemente notiert werden (die Wirkung ist äquivalent):

```
int a1[];
```

Die zweite Variante entspricht den Erwartungen von C- und C++-Programmieren. Die erste Variante ist wohl sinnvoller, da sie besagt, daß die deklarierte Referenzvariable auf ein `int`-Array verweist. In diesem Buch wird die erste Variante verwendet.

[95] Der Compiler erlaubt bei der Deklaration der Referenzvariablen keine Angabe zur Länge des Arrays. Damit sind wir wieder beim Thema „Referenzvariablen“. Sie haben bis jetzt lediglich ein Feld beziehungsweise eine lokale Variable *zur Aufnahme einer Referenz auf ein Arrayobjekt* deklariert, das heißt es wird genügend Speicherplatz allokiert, um eine Arrayreferenz speichern zu können. Es wurde allerdings noch kein Arbeitsspeicher für das Arrayobjekt selbst allokiert. Sie müssen einen Initialisierungsausdruck angeben, um Speicher für das eigentliche Array anzufordern. Die Initialisierung einer Referenzvariablen für ein Array kann an einer beliebigen Stelle geschehen. Bei Initialisierung an der Stelle der Deklaration können Sie eine besondere Art von Initialisierungsausdruck wählen, nämlich die **aggregierte Initialisierung**, das heißt, eine kommaseparierte Liste

von Werten zwischen geschweiften Klammern. In diesem Fall kümmert sich der Compiler um das Allokieren des Arbeitsspeichers (äquivalent zur Verwendung des `new`-Operators), zum Beispiel:

```
int[] a1 = { 1, 2, 3, 4, 5 };
```

Aus welchem Anlaß könnten Sie je eine Array-Referenzvariable ohne Array deklarieren?

```
int[] a2;
```

Java gestattet die Zuweisung einer Array-Referenzvariablen zu einer anderen:

```
a2 = a1;
```

Dabei wird der Wert von `a1` (eine Referenz auf ein Array) in `a2` kopiert, wie das folgende Beispiel zeigt:

```
//: initialization/ArraysOfPrimitives.java
import static net.mindview.util.Print.*;

public class ArraysOfPrimitives {
    public static void main(String[] args) {
        int[] a1 = { 1, 2, 3, 4, 5 };
        int[] a2;
        a2 = a1;
        for(int i = 0; i < a2.length; i++)
            a2[i] = a2[i] + 1;
        for(int i = 0; i < a1.length; i++)
            print("a1[" + i + "] = " + a1[i]);
    }
} /* Output:
      a1[0] = 2
      a1[1] = 3
      a1[2] = 4
      a1[3] = 5
      a1[4] = 6
   *///:~
```

Die lokale Variable `a1` wird mit einer Referenz auf ein Array initialisiert, nicht aber `a2`. Eine Zeile danach wird `a2` der Inhalt von `a1` zugewiesen. Da beide Variablen dasselbe Array referenzieren, können Sie die über `a2` bewirkten Änderungen anhand von `a1` nachvollziehen.

[96] Alle Arrays haben, unabhängig davon, ob sie Objektreferenzen oder Werte primitiven Typs enthalten, eine Eigenschaft, die Sie abfragen aber nicht ändern können, nämlich die Anzahl ihrer Elemente (`length`). Da die Arrayelemente bei Java, analog zu C und C++, von Null an indiziert sind, hat das letzte Element den Index `length - 1`. C und C++ akzeptieren das Verletzen von Arraygrenzen stillschweigend und erlauben Ihnen, durch Ihren gesamten Speicher zu trampeln. Dies ist die Ursache vieler unrühmlicher Fehler. Java schützt Sie vor solchen Problemen, in dem zur Laufzeit ein Fehler gemeldet (eine Ausnahme ausgeworfen) wird, wenn Sie die Grenzen eines Arrays verletzen.[5]

[97–98] Was können Sie tun, wenn Sie während der Arbeit an einem Programm nicht vorhersehen können, wieviele Elemente Ihr Array beinhalten wird? Der `new`-Operator gestattet, das Arrayobjekt

[5] Die Prüfung jedes einzelnen Zugriffs auf ein Array kostet Zeit und Anweisungen und es gibt keine Möglichkeit diese Prüfung abzuschalten. Arrayzugriffe können in kritischen Momenten die Effizienz eines Programms beeinträchtigen. Die Designer von Java haben diesen Nachteil zugunsten der Sicherheit in Bezug auf das Internet und die Produktivität der Programmierer in Kauf genommen. Auch wenn Sie sich versucht fühlen, den Zugriff auf ein Array nach Ihrem Empfinden programmatisch effizienter zu machen, verschwenden Sie Ihre Zeit, da Arrayzugriffe sowohl zur Übersetzungs- als auch zur Laufzeit durch automatische Optimierungen beschleunigt werden.

mit einer zur Laufzeit festgelegten Anzahl von Elementen zu erzeugen. Der `new`-Operator funktioniert, obwohl wir ein Array von Werten primitiven Typs erzeugen (`new` kann aber keine einzelnen Werte primitiven Typs erzeugen):

```
//: initialization/ArrayNew.java
// Creating arrays with new.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayNew {
    public static void main(String[] args) {
        int[] a;
        Random rand = new Random(47);
        a = new int[rand.nextInt(20)];
        print("length of a = " + a.length);
        print(Arrays.toString(a));
    }
} /* Output:
      length of a = 18
      [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
   *///:~
```

Die Länge des Arrays wird über die `Random`-Methode `nextInt()` zufällig bestimmt, die einen Wert zwischen Null und dem Vorgänger ihres Argumentes liefert (einschließlich dieser Randpunkte). Aufgrund der nicht vorhersehbaren Länge des Arrays ist klar, daß das Array tatsächlich erst zur Laufzeit erzeugt wird. Darüber hinaus zeigt dieses Beispiel, daß Arrayelemente primitiven Typs automatisch mit typspezifischen Initialwerten versehen werden: Bei Zahlen Null, bei `boolean false` und bei `char` \u0000 (ein Leerzeichen). Die statische `Arrays`-Methode `toString()`, die zur Bibliothek im Package `java.util` gehört, liefert eine ausgabefreundliche Version für ein eindimensionales Array.

[99] Das Arrayobjekt hätte im obigen Beispiel auch in einer Zeile zusammen mit der Deklaration erzeugt werden können:

```
int[] a = new int[rand.nextInt(20)];
```

Dies ist, sofern möglich, die bevorzugte Arbeitsweise. Ein Array von Elementen nicht primitiven Typs ist ein Array von Objektreferenzen. Das folgende Beispiel verwendet den Wrappertyp `Integer`, also eine Klasse anstelle eines primitiven Typs:

```
//: initialization/ArrayClassObj.java
// Creating an array of nonprimitive objects.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayClassObj {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] a = new Integer[rand.nextInt(20)];
        print("length of a = " + a.length);
        for(int i = 0; i < a.length; i++)
            a[i] = rand.nextInt(500); // Autoboxing
        print(Arrays.toString(a));
    }
} /* Output: (Sample)
      length of a = 18
      [55, 193, 361, 461, 429, 368, 200, 22, 207, 288, 128, 51, 89, 309, 278, 498,
361, 20]
   *///:~
```

Der `new`-Operator

```
Integer[] a = new Integer[rand.nextInt(20)];
```

erzeugt lediglich ein Array zur Aufnahme von Referenzen auf `Integer`-Objekte. Die Initialisierung des Arrays ist nicht beendet, bis jede Speicherstelle mit der Referenz auf ein `Integer`-Objekt bewertet wurde (hier mittels Autoboxing):

```
a[i] = rand.nextInt(500);
```

Wenn Sie eine Speicherstelle des Arrays beim Initialisieren auslassen, wird beim Zugriff auf dieses Element zur Laufzeit eine Ausnahme ausgeworfen.

[100] Ein Array von Objektreferenzen kann auch über eine kommaseparierte Liste zwischen geschweiften Klammern initialisiert werden. Es gibt zwei Varianten:

```
//: initialization/ArrayInit.java
// Array initialization.
import java.util.*;

public class ArrayInit {
    public static void main(String[] args) {
        Integer[] a = {
            new Integer(1),
            new Integer(2),
            3, // Autoboxing
        };
        Integer[] b = new Integer[] {
            new Integer(1),
            new Integer(2),
            3, // Autoboxing
        };
        System.out.println(Arrays.toString(a));
        System.out.println(Arrays.toString(b));
    }
} /* Output:
      [1, 2, 3]
      [1, 2, 3]
   *///:~
```

Das letzte Komma in der Initialisierungsliste ist in beiden Fällen optional. (Diese Eigenschaft zieht die Pflege längerer Listen in Betracht.)

[101] Die erste Variante ist zwar nützlich, aber in ihrer Anwendung eingeschränkt, da sie nur bei der Deklaration der Referenzvariablen verwendet werden kann. Die zweite Variante kann überall verwendet werden, sogar in der Argumentliste eines Methodenaufrufs. Das folgende Beispiel zeigt, wie Sie der `main()`-Methode einer anderen Klasse ein Array von `String`-Objekten als alternative Liste von Kommandozeilenargumenten übergeben können:

```
//: initialization/DynamicArray.java
// Array initialization.

public class DynamicArray {
    public static void main(String[] args) {
        Other.main(new String[] { ''fiddle'', ''de'', ''dum'' });
    }
}

class Other {
    public static void main(String[] args) {
        for(String s : args)
```

```
            System.out.print(s + " ");
        }
    } /* Output:
          fiddle de dum
       *///:~
```

Das als Argument der `main()`-Methode der Klasse `Other` übergebene Array wird beim Methodenaufruf erzeugt, so daß Sie zum Zeitpunkt des Aufrufs andere Argumente übergeben können.

Übungsaufgabe 16: (1) Erzeugen Sie ein `String`-Array und weisen Sie jedem Speicherplatz ein `String`-Objekt zu. Geben Sie das Array mittels einer `for`-Schleife aus. ■

Übungsaufgabe 17: (2) Schreiben Sie eine Klasse mit einem Konstruktor, der ein Argument vom Typ `String` erwartet. Geben Sie das Argument während der Objekterzeugung aus. Deklarieren Sie eine Referenzvariable für ein Array von Objekten Ihrer Klasse, aber initialisieren Sie das Array nicht. Achten Sie beim Aufrufen des Programms darauf, ob die Initialisierungsmeldung des Konstruktors ausgegeben wird. ■

Übungsaufgabe 18: (1) Vervollständigen Sie Übungsaufgabe 17, indem Sie jedem Speicherplatz des Arrays eine Objektreferenz zuweisen. ■

6.8.1 Argumentlisten variabler Länge

[102–103] Die zweite Variante liefert eine komfortable Syntax zum Definieren und Aufrufen von Methoden mit einer Argumentliste variabler Länge (in der englischsprachigen Literatur auch als „varargs“ bezeichnet). Eine solche Liste kann eine unbekannte Anzahl von Elementen eines einheitlichen unbekannten Typs haben. Da alle Klassen letztendlich von der Wurzelklasse `Object` abgeleitet sind (ein Thema über das Sie in diesem Buch noch mehr lernen werden), können Sie eine Methode definieren, die ein Array von Referenzen vom Typ `Object` erwartet:

```
//: initialization/VarArgs.java
// Using array syntax to create variable argument lists.

class A {}

public class VarArgs {
    static void printArray(Object[] args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        printArray(new Object[] {
            new Integer(47), new Float(3.14), new Double(11.11)
        });
        printArray(new Object[] { "one", "two", "three" });
        printArray(new Object[] { new A(), new A(), new A() });
    }
} /* Output: (Sample)
      47 3.14 11.11
      one two three
      A@1a46e30 A@3e25a5 A@19821f
   *///:~
```

Die Methode `printArray()` erwartet ein Array von `Object`-Referenzen, wählt über die erweiterte `for`-Schleife ein Arrayelement nach dem anderen aus und gibt es aus. Die Klassen der Standardbibliothek von Java liefern eine sinnvollere Ausgabe. Die Objekte der Klasse `A` werden nur

als Klassenname, gefolgt von einem `@`-Zeichen und einigen hexadezimalen Ziffern ausgegeben. Das Standardverhalten (falls Sie die `toString()`-Methode nicht überschreiben, ~~siehe später in diesem Buch~~) besteht in der Ausgabe des Klassennamens sowie der Speicheradresse des Objektes.

[104] Sie finden die obige Syntax für Argumentlisten variabler Länge in älteren Quelltexten vor Version 5 der Java Standard Edition (SE 5). Die SE 5 führt schließlich die Auslassungspunkte ein, eine lange erwartete Eigenschaft, die Sie in der `printArray()`-Methode im folgenden Beispiel sehen:

```
//: initialization/NewVarArgs.java
// Using array syntax to create variable argument lists.

public class NewVarArgs {
    static void printArray(Object... args) {
        for(Object obj : args)
            System.out.print(obj + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        // Can take individual elements:
        printArray(new Integer(47), new Float(3.14), new Double(11.11));
        printArray(47, 3.14F, 11.11);
        printArray("one", "two", "three");
        printArray(new A(), new A(), new A());
        // Or an array:
        printArray((Object[])new Integer[] { 1, 2, 3, 4 });
        printArray(); // Empty list is OK
    }
} /* Output: (75% match)
      47 3.14 11.11
      47 3.14 11.11
      one two three
      A@1bab50a A@c3c749 A@150bd4d
      1 2 3 4
   *///:~
```

[105] Durch die Auslassungspunkte brauchen Sie die Arraysyntax nicht mehr explizit auszuschreiben. Der Compiler setzt diese Syntax für Sie ein, wenn Sie eine Argumentliste variabler Länge definieren. Sie bekommen nach wie vor ein Array, so daß Sie in der `printArray()`-Methode die erweiterte `for`-Schleife anwenden können. Die Auslassungspunkte gestatten noch mehr Flexibilität, als nur die automatische Umwandlung einer Liste von Elementen in ein Array. Beachten Sie die zweitletzte Zeile, in der ein Array von Referenzen auf `Integer`-Objekte (initialisiert per Autoboxing) in ein Array von Referenzen vom Typ `Object` umgewandelt (um eine Warnung des Compilers zu beseitigen) und der `printArray()`-Methode übergeben wird. Der Compiler erkennt, daß das übergebene Argument bereits ein Array ist *und verzichtet auf die Umwandlung*. Sie können also einerseits mehrere Argumente als Liste übergeben und andererseits akzeptiert die Syntax auch ein bereits vorhandenes Array als Argumentliste variabler Länge.

[106] Die letzte Zeile zeigt, daß eine Methode, die eine Argumentliste variabler Länge erwartet, auch ohne Argumente aufgerufen werden kann. Das ist praktisch bei optionalen nachlaufenden Argumenten:

```
//: initialization/OptionalTrailingArguments.java
public class OptionalTrailingArguments {
   static void f(int required, String... trailing) {
       System.out.print("required: " + required + " ");
       for(String s : trailing)
           System.out.print(s + " ");
```

```
        System.out.println();
    }
    public static void main(String[] args) {
        f(1, ''one'');
        f(2, ''two'', ''three'');
        f(0);
    }
} /* Output:
      required: 1 one
      required: 2 two three
      required: 0
   *///:~
```

[107] Dieses Beispiel zeigt nebenbei, wie Sie eine Argumentliste variabler Länge mit einem anderen Typ als `Object` verwenden können. Hier sind im variablen Teil der Argumentliste nur Argument vom Typ `String` erlaubt. Eine Argumentliste variabler Länge kann für jeden beliebigen Typ definiert werden, insbesondere für jeden primitiven Typ. Das folgende Beispiel zeigt zweierlei: Eine Argumentliste variabler Länge wird in ein Array umgewandelt und eine leere Argumentliste in ein Array der Länge Null.

```
//: initialization/VarargType.java
public class VarargType {
    static void f(Character... args) {
        System.out.print(args.getClass());
        System.out.println('' length '' + args.length);
    }
    static void g(int... args) {
        System.out.print(args.getClass());
        System.out.println('' length '' + args.length);
    }
    public static void main(String[] args) {
        f('a');
        f();
        g(1);
        g();
        System.out.println(''int[]: '' + new int[0].getClass());
    }
} /* Output:
      class [Ljava.lang.Character; length 1
      class [Ljava.lang.Character; length 0
      class [I length 1
      class [I length 0
      int[]: class [I
   *///:~
```

Die Methode `getClass()` ist in der Klasse `Object` definiert und wird in Kapitel 15 ausführlich erklärt. `getClass()` gibt eine Referenz auf das sogenannte Klassenobjekt (`Class`-Objekt) einer Klasse zurück. Die Ausgabe eines Klassenobjekt liefert eine „Zeichenkettendarstellung“. Die führende eckige Klammer (`[`) zeigt ein Array an, gefolgt vom Typ der Arrayelemente. Das Zeichen „`I`“ zeigt den primitiven Typ `int` an. Die letzte Zeile gibt zur genauen Überprüfung das Klassenobjekt eines Arrays von `int`-Werten aus. Dies zeigt, daß eine Argumentliste variabler Länge nicht vom Autoboxing abhängt, sondern in Wirklichkeit die primitiven Typen verwendet.

[108] Dennoch harmonieren Argumentlisten variabler Länge und Autoboxing, zum Beispiel:

```
//: initialization/AutoboxingVarargs.java
public class AutoboxingVarargs {
```

```
    public static void f(Integer... args) {
        for(Integer i : args)
            System.out.print(i + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        f(new Integer(1), new Integer(2));
        f(4, 5, 6, 7, 8, 9);
        f(10, new Integer(11), 12);
    }
} /* Output:
      1 2
      4 5 6 7 8 9
      10 11 12
   *///:~
```

Werte primitiven Typs und Objekte der Wrapperklassen können gemeinsam in einer Argumentliste auftreten und Autoboxing wandelt selektiv die `int`-Werte in `Integer`-Objekte um.

[109] Argumentlisten variabler Länger verkomplizieren das Überladen von Methoden:

```
//: initialization/OverloadingVarargs.java
public class OverloadingVarargs {
    static void f(Character... args) {
        System.out.print("first");
        for(Character c : args)
            System.out.print(" " + c);
        System.out.println();
    }
    static void f(Integer... args) {
        System.out.print("second");
        for(Integer i : args)
            System.out.print(" " + i);
        System.out.println();
    }
    static void f(Long... args) {
        System.out.println("third");
    }
    public static void main(String[] args) {
        f('a', 'b', 'c');
        f(1);
        f(2, 1);
        f(0);
        f(0L);
        //! f(); // Won't compile - ambiguous
    }
} /* Output:
      first a b c
      second 1
      second 2 1
      second 0
      third
   *///:~
```

Der Compiler wendet in allen drei Fällen Autoboxing an und wählt die am besten passende Methode.

[110] Der Compiler kann beim Aufrufen der Methode `f()` ohne Argumente nicht entscheiden, welche Version gemeint ist. Der Fehler ist einerseits einleuchtend, wird den Clientprogrammierer aber

wahrscheinlich überraschen.

[111] Der Versuch, das Problem durch Hinzufügen eines zusätzlichen Parameters vor der Argumentliste variabler Länge zu lösen, scheitert:

```
//: initialization/OverloadingVarargs2.java
// {CompileTimeError} (Won't compile)

public class OverloadingVarargs2 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(Character... args) {
        System.out.print("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} ///:~
```

Der Kommentar `{CompileTimeError}` schließt die Datei *OverloadingVarargs2.java* vom Ant-Skript zur Quelltextdistribution dieses Buches aus. Wenn Sie das Programm manuell übersetzen, bekommen Sie die folgende Fehlermeldung:

```
OverloadingVarargs2.java:13: reference to f is ambiguous, \
    both method f(float,java.lang.Character...) in OverloadingVarargs2 \
    and method f(java.lang.Character...) in OverloadingVarargs2 match
        f('a', 'b');
        ^
1 error
```

Wenn Sie in beiden Versionen der Methode `f()` einen zusätzliche Parameter definieren, läßt sich das Programm übersetzen:

```
//: initialization/OverloadingVarargs3.java
public class OverloadingVarargs3 {
    static void f(float i, Character... args) {
        System.out.println("first");
    }
    static void f(char c, Character... args) {
        System.out.println("second");
    }
    public static void main(String[] args) {
        f(1, 'a');
        f('a', 'b');
    }
} /* Output:
      first
      second
   *////:~
```

In der Regel sollten Sie eine Argumentliste variabler Länge nur bei einer Version einer überladenen Methoden verwenden, oder in Betracht ziehen, ganz darauf zu verzichten.

Übungsaufgabe 19: (2) Definieren Sie eine Methode, die ein Array von Referenzen auf `String`-Objekte als Argumentliste variabler Länge erwartet. Verifizieren Sie, daß Sie sowohl eine kommaseparierte Liste von `String`-Referenzen als auch ein `String`-Array übergeben können. ■

Übungsaufgabe 20: (1) Definieren Sie eine `main()`-Methode, die statt der gewöhnlichen Syntax eine Argumentliste variabler Länge definiert. Geben Sie alle Elemente des resultierenden Arrays (`args`) aus. Testen Sie die `main()`-Methode mit unterschiedlich vielen Kommandozeilenargumenten. ■

6.9 Aufzählungstypen

[112] Das Schlüsselwort `enum` ist eine scheinbar geringfügige Erweiterung in der SE 5 und erleichtert Ihnen die Arbeit, wenn Sie eine Anzahl von Werten zu einem Aufzählungstyp zusammenfassen wollen. Zuvor mußten Sie eine Menge ganzzahliger Konstanten deklarieren, die allerdings nicht in natürlicher Weise auf ihre Element beschränkt und daher riskanter und schwieriger anzuwenden war. Aufzählungstypen sind häufig genug benötigte Komponenten und waren bei C, C++ sowie vielen anderen Sprachen schon immer vorhanden. Vor der SE 5 mußten Java-Programmierer viel wissen und sorgfältig vorgehen, um den Effekt eines Aufzählungstyps korrekt zu implementieren. Nun verfügt auch Java über Aufzählungstypen und ihre Eigenschaften und Fähigkeiten gehen weit über die Funktionalität ihrer Äquivalente in C und C++ hinaus. Ein einfaches Beispiel:

```
//: initialization/Spiciness.java
public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~
```

Diese Zeilen definieren den Aufzählungstyp `Spiciness` mit fünf benannten Werten. Die Elemente eines Aufzählungstyps sind Konstanten und werden daher konventionsgemäß durchgängig mit Großbuchstaben bezeichnet (besteht der Name einer Konstanten aus mehreren Worten, so werden diese mit Unterstrichen getrennt). Sie verwenden eine Konstante eines Aufzählungstyps, indem Sie eine Referenzvariable vom Typ des Aufzählungstyps deklarieren und ihr eine Konstante zuweisen:

```
//: initialization/SimpleEnumUse.java
public class SimpleEnumUse {
    public static void main(String[] args) {
        Spiciness howHot = Spiciness.MEDIUM;
        System.out.println(howHot);
    }
} /* Output:
      MEDIUM
   *////:~
```

[113–114] Der Compiler stattet die Konstanten eines Aufzählungstyps automatisch mit nützlichen Eigenschaften und Fähigkeiten aus. Beispielsweise bekommt jede Konstante eine `toString()`-Methode, so daß Sie ihren Namen ausgeben können (siehe `println()`-Anweisung im obigen Beispiel). Der Aufzählungstyp verfügt außerdem über eine `ordinal()`-Methode, die die Position einer Konstanten in der Deklarationsreihenfolge angibt sowie eine statische `values()`-Methode, die eine Referenz auf ein Array mit den Konstanten des Aufzählungstyps in der Reihenfolge ihrer Deklaration zurückgibt:

```
//: initialization/EnumOrder.java
public class EnumOrder {
    public static void main(String[] args) {
        for(Spiciness s : Spiciness.values())
            System.out.println(s + ", ordinal " + s.ordinal());
    }
} /* Output:
      NOT, ordinal 0
```

```
      MILD, ordinal 1
      MEDIUM, ordinal 2
      HOT, ordinal 3
      FLAMING, ordinal 4
   *///:~
```

Obwohl ein Aufzählungstyp scheinbar ein neuer Datentyp ist, bewirkt das Schlüsselwort `enum` einige Compileraktivität, während für den Aufzählungstyp eine Klasse generiert wird, so daß Sie einen Aufzählungstyp wie jede andere Klasse behandeln können. Aufzählungstypen *sind* in Wirklichkeit Klassen und haben eigene Methoden.

[115] Die Kombination eines Aufzählungstyps mit einer `switch`-Anweisung ist eine besonders erfreuliche Anwendung:

```
//: initialization/Burrito.java
public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree;}
    public void describe() {
        System.out.print("This burrito is ");
        switch(degree) {
        case NOT: System.out.println("not spicy at all.");
            break;
        case MILD:
        case MEDIUM: System.out.println("a little hot.");
            break;
        case HOT:
        case FLAMING:
        default: System.out.println("maybe too hot.");
        }
    }
    public static void main(String[] args) {
        Burrito
            plain = new Burrito(Spiciness.NOT),
            greenChile = new Burrito(Spiciness.MEDIUM),
            jalapeno = new Burrito(Spiciness.HOT);
        plain.describe();
        greenChile.describe();
        jalapeno.describe();
    }
} /* Output:
      This burrito is not spicy at all.
      This burrito is a little hot.
      This burrito is maybe too hot.
   *///:~
```

[116] Die `switch`-Anweisung ist zur Auswahl aus einer begrenzten Menge von Möglichkeiten gedacht und paßt daher ideal zu Aufzählungstypen. Beachten Sie, wie die Namen der Konstanten viel deutlicher anzeigen, wie sich das Programm verhält.

[117] Im allgemeinen können Sie das Schlüsselwort `enum` als eine weitere Möglichkeit zur Definition eines Datentyps betrachten und anschließend einfach mit den definierten Konstanten programmieren. Dies ist die Kernaussage, das heißt Sie sich brauchen nicht allzu intensiv mit Aufzählungstypen auseinanderzusetzen. Vor der Einführung des Schlüsselwortes `enum` in der SE 5 mußte einige Mühe investiert werden, um einen gleichwertigen, sicher anzuwenden Aufzählungstyp zu definieren.

[118] Das genügt, damit Sie die Grundzüge der Verwendung von Aufzählungstypen verstehen, aber wir kehren später in diesem Buch noch einmal zu diesem Thema zurück. Die Aufzählungstypen

haben ein eigenes Kapitel: Kapitel 20.

Übungsaufgabe 21: (1) Definieren Sie einen Aufzählungstyp aus den sechs kleinsten Banknoten. Wählen Sie mittels einer Schleife jede Konstante einmal aus (`values()`) und geben Sie ihre Position in der Deklarationsreihenfolge aus (`ordinal()`). ■

Übungsaufgabe 22: (2) Legen Sie eine `switch`-Anweisung für den Aufzählungstyp von Übungsaufgabe 21 an. Geben Sie in jedem `case`-Zweig eine Beschreibung der jeweiligen Banknote aus. ■

6.10 Zusammenfassung

[119] Das Konzept des Konstruktors, dieses offensichtlich sorgfältig durchdachten Initialisierungsmechanismus', sollte für Sie ein deutlicher Hinweis auf das Gewicht sein, das der Initialisierung in dieser Sprache zugemessen wurde. Eine der ersten Beobachtungen im Hinblick auf die Produktivität von C-Programmieren, die Bjarne Stroustrup, dem Erfinder von C++, beim Entwurf von C++ auffiel, bestand darin, daß falsch oder gar nicht initialisierte Variablen für einen beträchtlichen Anteil der Programmierprobleme verantwortlich waren. Derartige Fehler sind schwierig zu finden und dasselbe gilt für unsauberes Aufräumen. Nachdem Konstruktoren dem Programmierer gestatten, korrekte Initialisierung und Aufräumen zu *garantieren* (der Compiler erlaubt keine Objekterzeugung ohne Aufruf des entsprechenden Konstruktors), haben Sie vollständige Kontrolle und Sicherheit.

[120] Bei C++ ist die Zerstörung von Objekten wesentlich, da jedes per `new` erzeugte Objekt explizit zerstört werden muß. Bei Java gibt die automatische Speicherbereinigung den von sämtlichen Objekten beanspruchten Arbeitsspeicher wieder frei, so daß zumeist keine äquivalente Aufräummethode erforderlich ist (ist Aufräumen notwendig, so sind Sie selbst dafür zuständig). Die automatische Speicherbereinigung von Java vereinfacht das Programmieren erheblich und erweitert die Speicherverwaltung um dringend benötigte Sicherheitsaspekte, sofern Sie kein destruktorartiges Verhalten brauchen. Es gibt Speicherbereinigungen, die sogar Resourcen wie Bildschirminhalte und Dateihandles aufräumen können. Andererseits verursacht die automatische Speicherbereinigung zur Laufzeit zusätzliche Unkosten, deren Ausmaß aufgrund der historischen Langsamkeit der Java-Interpreter schwierig zu relativieren ist. Obwohl Java im Laufe der Zeit deutliche Performanzverbesserungen erfahren hat, hat das Geschwindigkeitsproblem hinsichtlich der Akzeptanz der Sprache bei bestimmten Programmierproblemen seinen Tribut gefordert.

[121] Die Garantie, daß jedes Objekt per Konstruktor erzeugt wird, bedeutet, daß Konstruktoren mehr Eigenschaften und Fähigkeiten haben, als in diesem Kapitel besprochen wurden. Die Konstruktorgarantie besteht auch, wenn Sie mit Hilfe von *Komposition* oder *Ableitung* neue Klassen entwickeln und es ist weitere Syntax erforderlich, um dies zu unterstützen. In den folgenden Kapiteln lernen Sie Komposition, Ableitung und die jeweiligen Auswirkungen auf Konstruktoren kennen.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 7

Zugriffskontrolle

Inhaltsübersicht

[0] Zugriffskontrolle (beziehungsweise das Verbergen der Implementierung) hat damit zu tun, „es beim ersten Mal noch nicht richtig hinzubekommen“.

[1] Jeder gute Autor, darunter auch die Autoren von Software, weiß, daß ein Text/Quelltext nicht gut ist, bevor er wieder und wieder neu geschrieben wurde, oft viele Male. Wenn Sie ein Stück Quelltext für einige Zeit in der Schublade liegen lassen und es sich dann wieder vornehmen, erkennen Sie unter Umständen eine viel bessere Möglichkeit, um das Problem zu lösen. Dies ist eine der wichtigsten Motivationen zur Refaktorierung, einem Verfahren, bei dem Sie einen bereits funktionierenden Quelltext überarbeiten, um ihn leichter lesbar und verständlicher machen und somit letztendlich seine Pflege zu erleichtern.[1]

[2] Der Wunsch, einen Quelltext zu überarbeiten und zu verbessern, verursacht allerdings auch Reibung. Es gibt in der Regel Clientprogrammierer (Konsumenten), die sich darauf verlassen, daß

[1] Siehe Martin Fowler et al.: *Improving the Design of Existing Code*, Addison-Wesley (1999). Gelegentlich wird gegen Refaktorierung argumentiert, daß ein funktionstüchtiger Quelltext perfekt und eine Refaktorierung Zeitverschwendung sei. Das Problem an dieser Perspektive besteht darin, daß der größte Teil der Zeit und des Geldes bei einem Projekt nicht im Erstellen der ersten lauffähigen Version eines Quelltextes steckt, sondern in dessen Pflege. Eine Investition in die Verständlichkeit des Quelltextes schlägt sich maßgeblich in gesparten Dollars (Euros) nieder.

einige Aspekte Ihres Quelltextes unverändert bleiben. Während Sie eine Änderungen implementieren möchten, verlangen die Clientprogrammierer, daß der Quelltext nicht geändert wird. Eine primäre Frage beim objektorientierten Design betrifft daher die Trennung der veränderlichen von den unveränderlichen Dingen.

[3] Dies ist bei Bibliotheken besonders wichtig. Die Clientprogrammierer einer Bibliothek müssen sich auf den Teil verlassen können, den sie anwenden und wissen, daß sie ihre Quelltexte nicht zu ändern brauchen, wenn eine neue Version der Bibliothek herausgegeben wird. Andererseits muß der Autor einer Bibliothek die Freiheit haben, Änderungen und Verbesserungen in der Gewißheit zu implementieren, daß die Quelltexte der Clientprogrammierer nicht funktionsuntüchtig werden.

[4] Diese Sicherheit *könnte* durch Übereinkünfte erreicht werden. Der Autor einer Bibliothek muß sich einverstanden erklären, bei Änderungen an einer Klasse seiner Bibliothek keine existierenden Methoden zu entfernen, da hierdurch Quelltexte von Clientprogrammierern funktionsuntüchtig werden können. Die umgekehrte Situation ist allerdings noch erheblich komplizierter. Wie kann der Autor einer Bibliothek wissen, welche Felder die Clientprogrammierer zum Beispiel verwenden? Dasselbe gilt für Methoden, die lediglich zur internen Implementierung einer Klasse gehören und nicht für die direkte Anwendung durch die Clientprogrammierer gedacht sind. Was ist, wenn der Autor einer Bibliothek eine ältere Implementierung durch eine neue ersetzen will? Jede geänderte Komponente kann sich nachteilig auf Quelltexte der Clientprogrammierer auswirken. Der Autor einer Bibliothek steckt also in einer Zwangsjacke und kann nichts ändern.

[5] Java verfügt zur Lösung dieses Problems über *Zugriffsmodifikatoren*, mit deren Hilfe der Autor einer Bibliothek deklarieren kann, welche Komponenten den Clientprogrammierern zur Verfügung stehen und welche nicht. Die Zugriffseinschränkungen werden in der Reihenfolge `public`, `protected`, Packagezugriff (ohne Schlüsselwort) und `private` immer enger. Nach dem vorigen Absatz haben Sie vermutlich den Eindruck, daß Sie beim Design einer Bibliothek soviele Komponenten wie möglich so privat wie möglich deklarieren und nur solche Methoden exponieren sollten, die die Clientprogrammierer verwenden dürfen. Dieser Gedanke ist völlig richtig, auch wenn er Programmierern nicht eingängig erscheint, die von anderen Sprachen kommen (vor allem C) und es gewohnt sind, ohne Einschränkung auf jede Komponente zugreifen zu können. Bis zum Ende dieses Kapitels sollten Sie vom Wert der Zugriffskontrolle bei Java überzeugt sein.

[6] Das Konzept der Bibliothek von Komponenten mit Kontrolle darüber, wer auf welche Komponente zugreifen darf, ist aber noch unvollständig. Es bleibt noch die Frage, wie die Komponenten zu einer in sich geschlossenen Einheit innerhalb der Bibliothek zusammengefaßt werden können. Das Schlüsselwort `package` liefert diesen fehlenden Baustein. Die Zugriffsmodifikatoren werden davon beeinflußt, ob eine Klasse im selben oder einem anderen Package liegt. Das Kapitel beginnt daher mit der Zuordnung von Bibliothekskomponenten zu Packages. Anschließend sind Sie vorbereitet, um die Bedeutung der Zugriffsmodifikatoren insgesamt zu verstehen.

7.1 Packages: Abgeschlossene Bibliothekseinheiten

[7–8] Ein Package beinhaltet mehrere Klassen und ordnet sie einem gemeinsamen Namensraum zu. Die Standardbibliothek von Java enthält beispielsweise eine Bibliothek von Hilfsklassen unter dem Namensraum `java.util`. Eine der Klassen in `java.util` ist `ArrayList`. Eine Möglichkeit, sich auf `ArrayList` zu beziehen, ist die Angabe des *voll qualifizierten Klassennamens* `java.util.ArrayList`:

```
//: access/FullQualification.java
public class FullQualification {
    public static void main(String[] args) {
        java.util.ArrayList list = new java.util.ArrayList();
```

```
    }
} ///:~
```

[9] Die Angabe des voll qualifizierten Namens wird schnell lästig. Die `import`-Anweisung leistet Abhilfe. Die `import`-Anweisung im folgenden Beispiel importiert eine einzelne Klasse:

```
//: access/SingleImport.java
import java.util.ArrayList;

public class SingleImport {
    public static void main(String[] args) {
        ArrayList list = new java.util.ArrayList();
    }
} ///:~
```

Die `import`-Anweisung gestattet, den Namen `ArrayList` ohne Qualifizierung zu verwenden. Die übrigen Klassen im Package `java.util` wurden aber nicht importiert. Die folgende `*`-Notation importiert alle Namen eines Packages und wird von nun an bis zum Endes des Buches verwendet:

```
import java.util.*;
```

[10] Das Importieren von Namen liefert einen Mechanismus zur Verwaltung von Namensräumen. Die Namen der Komponenten einer Klasse sind gegen die Namen der Komponenten einer anderen Klasse abgeschirmt. Die Methode `f()` von Klasse `A` kollidiert nicht mit der signaturidentischen Methode `f()` aus Klasse `B`. Wie steht es aber mit den Klassennamen? Angenommen, Sie schreiben eine Klasse namens `Stack`, die in einer Umgebung installiert wird, die bereits eine gleichnamige Klasse eines anderen Programmierers enthält? Aufgrund dieser Kollisionsmöglichkeit ist es wichtig, bei Java umfassende Kontrolle über die Namensräume zu haben und jeder Klasse einen eindeutigen Namen zu geben.

[11] Die meisten Beispiele in den vorigen Kapiteln bestanden nur aus einer einzelnen Datei und waren zum lokalen Aufruf gedacht, so daß keine *expliziten* Packagenamen nötig waren. Die Beispiele gehören nämlich *implizit* doch einem Package an, nämlich dem unbenannten **Standardpackage**. Die Zuordnung zum Standardpackage ist eine zulässige Option und wird, der Einfachheit halber, bis zum Ende des Buches so oft als möglich verwendet. Wenn Sie allerdings eine Bibliothek oder ein Programm entwickeln, das sich den anderen Java-Bibliotheken auf demselben Rechner gegenüber „kollegial“ verhalten soll, müssen Sie sich über das Vermeiden von Namenskollisionen Gedanken machen.

[12] Quelltextdateien werden häufig als *Übersetzungseinheiten* bezeichnet. Jede Übersetzungseinheit muß bei Java die Endung *.java* haben und eine öffentliche, also als `public` deklarierte Klasse enthalten, deren Name mit dem Dateinamen übereinstimmt (inklusive Groß-/Kleinschreibung und ohne die Endung *.java*). Pro Übersetzungseinheit ist höchstens eine öffentliche Klasse erlaubt, andernfalls beschwert sich der Compiler. Weitere Klassen derselben Übersetzungseinheit sind als nicht öffentliche Klassen vor der Außenwelt verborgen und dienen als „Unterstützungsklassen“ der öffentlichen Klasse.

7.1.1 Zuordnung von Klassen zu Namensräumen

[13] Beim Übersetzen einer *.java* Datei erhalten Sie zu jeder in dieser Datei definierten Klasse eine Ausgabedatei. Die Ausgabedatei hat den Namen der Klasse in der *.java* Datei und die Endung *.class* („Klassendatei“). Das Übersetzen einiger *.java* Dateien kann zu einer beträchtlichen Anzahl von Klassendateien führen. Wenn Sie bereits Erfahrung mit übersetzten Programmiersprachen haben, sind Sie eventuell daran gewöhnt, daß der Compiler ein intermediäres Dateiformat liefert (in der Regel eine *.obj* Datei), welches mit Hilfe eines Linkers in eine ausführbare Datei beziehungsweise

per librarian in eine Bibliothek umgewandelt wird. Java funktioniert anders. Ein funktionstüchtiges Programm besteht aus mehreren Klassendateien, die mit dem Hilfsprogramm `jar` zu einer *.jar* Datei („Java Archive“) verpackt und komprimiert werden können. Die Laufzeitumgebung ist dafür zuständig, die Klassendateien zu lokalisieren, zu laden und zu interpretieren.[2]

[14] Eine Bibliothek besteht aus mehreren Klassendateien. In der Regel enthält jede *.java* Datei eine öffentliche Klasse sowie beliebig viele nicht-öffentliche Klassen, das heißt es gibt eine öffentliche Komponente pro *.java* Datei. Das Schlüsselwort `package` kommt ins Spiel, wenn Sie ausdrücken wollen, daß alle diese Komponenten zusammengehören.

[15] Die `package`-Anweisung *muß*, falls sie existiert, die erste Zeile der Datei sein, die keine Kommentarzeile ist. Die Anweisung

```
package access;
```

definiert, daß diese Übersetzungseinheit Teil einer Bibliothek namens `access` ist. Anders ausgedrückt, besagt diese `package`-Anweisung, daß die öffentliche Klasse dieser Übersetzungseinheit unter dem Schutz des Namens `access` steht und jeder, der sich auf die öffentliche Klasse beziehen möchte, entweder den voll qualifizierten Namen angeben oder eine `import`-Anweisung für das `access`-Package anlegen muß, wie zuvor beschrieben. (Beachten Sie, daß Packagenamen konventionsgemäß durchgängig in Kleinbuchstaben geschrieben werden, auch wenn der Name aus mehreren Worten besteht.)

[16] Lautet der Dateiname beispielsweise *MyClass.java*, so darf die Datei nur eine einzige öffentliche Klasse enthalten, deren Name `MyClass` sein muß (inklusive Groß-/Kleinschreibung):

```
//: access/mypackage/MyClass.java
package access.mypackage;

public class MyClass {
    // ...
} ///:~
```

Ein Programmierer, der die Klasse `MyClass`, oder eine der anderen öffentlichen Klassen im Package `access` verwenden möchte, muß den beziehungsweise die Namen in `access` mit Hilfe einer `import`-Anweisung verfügbar machen oder den voll qualifizierte Namen angegeben:

```
//: access/QualifiedMyClass.java
public class QualifiedMyClass {
    public static void main(String[] args) {
        access.mypackage.MyClass m = new access.mypackage.MyClass();
    }
} ///:~
```

Die `import`-Anweisung macht den Quelltext übersichtlicher:

```
//: access/ImportedMyClass.java
import access.mypackage.*;

public class ImportedMyClass {
    public static void main(String[] args) {
        MyClass m = new MyClass();
    }
} ///:~
```

[17] Die Schlüsselwörter `package` und `import` gestatten dem Designer einer Bibliothek, den einen globalen Namensraum aufzuteilen, so daß sich keine Namenskollisionen ergeben können, gleichgültig

[2] Java erzwingt keineswegs die Verwendung einer Laufzeitgebung (eines Interpreters). Es gibt Java-Compiler, die eine einzelne native ausführbare Datei erzeugen.

wieviele Programmierer im Internet unterwegs sind und Java-Klassen schreiben.

7.1.2 Eindeutigkeit von Packagenamen

[18] Da ein Package niemals wirklich zu einer einzelnen Datei zusammengepackt wird und aus vielen Klassendateien bestehen kann, besteht die Gefahr, daß die Menge von Klassendateien unübersichtlich wird. Eine logische Gegenmaßnahme besteht darin, alle Klassendateien, die einem bestimmten Package zugeordnet sind, in einem einzigen Verzeichnis unterzubringen, also die vom Betriebssystem bewirkte hierarchische Struktur des Dateisystems zu Ihrem Vorteil zu nutzen. Dies ist eine Möglichkeit, mit der Java das Ordnungsproblem löst. Sie lernen später eine weitere Möglichkeit kennen, wenn wir das Hilfsprogramm `jar` besprechen.

[19] Das Sammeln aller Klassendateien eines Packages in einem Unterverzeichnis löst zwei weitere Probleme: Die Wahl eines eindeutigen Packagenamens und das Lokalisieren dieser Klassendateien, die tief in der Verzeichnisstruktur verborgen sein können. Der Packagename gibt den Pfad zur Klassendatei an. Konventionsgemäß beginnt der Packagename mit dem rückwärts geschriebenen Namen der Internetdomain des Autors der Klasse. Wenn Sie diese Konvention einhalten, bekommen Sie einen eindeutigen Packagenamen und erleiden niemals eine Namenskollision, da die Domainnamen im Internet garantiert eindeutig sind. (Solange, bis Sie Ihren Domainnamen an jemand anderes abtreten, der in Java programmiert und dieselben Pfadnamen wählt wie Sie.) Falls Sie keine eigene Domain besitzen, sollten Sie sich ein Anfangsstück für Ihre Packagenamen überlegen, das wahrscheinlich kein zweites Mal gewählt wird, zum Beispiel Ihren Vor- und Namennamen. Wenn Sie Ihren Java-Code öffentlich zur Verfügung stellen möchten, lohnt sich der relativ geringe Aufwand, eine eigene Domain zu registrieren.

[20] Der zweite Teil des Tricks besteht in der Abbildung des Packagenamens auf ein Verzeichnis Ihrer Festplatte, damit Compiler und Laufzeitumgebung das Verzeichnis finden können, in dem die Klassendatei deponiert wurde.

[21] Die Laufzeitumgebung geht folgendermaßen vor: Zuerst wird die Umgebungsvariable `$CLASSPATH`,[3] der sogenannte **Klassenpfad** ausgewertet. (Der Klassenpfad wird vom Betriebssystem deklariert und zuweilen von Installationsprogrammen aktualisiert, die Java oder Java-basierte Hilfsprogramme auf Ihrem Rechner installieren.) Der Klassenpfad besteht aus einem oder mehreren Verzeichnissen, die als Wurzelverzeichnisse bei der Suche nach Klassendateien dienen. Die Laufzeitumgebung ersetzt jeden Punkt im Namen eines Packages durch ein Pfadtrennungszeichen (je nach Betriebssystem wird der Packagename `foo.bar.baz` also in *foo/bar/baz*, *foo\bar\baz* oder noch eine andere Variante umgewandelt), kombiniert dieses Endstück mit den im Klassenpfad deklarierten Anfangsstücken und sucht in diesen Verzeichnissen nach der Klassendatei zur der Klasse, von der Sie ein Objekt erzeugen wollen. (Die Laufzeitumgebung durchsucht außerdem einige standardisierte Verzeichnisse in Ihrer Java-Installation.)

[22–23] Nehmen Sie meinen Domainnamen als Beispiel: *MindView.net*. Durch Umkehren der Reihenfolge der Worte und Ändern aller Groß- in Kleinbuchstaben erhalten wir `net.mindview`, meinen weltweit eindeutigen Präfix für meine Klassen. (Die Endungen *com*, *edu*, *org* und so weiter wurden früher in Java-Packages großgeschrieben, aber diese Konvention wurde mit Java 2 geändert, so daß nun der gesamte Packagename aus Kleinbuchstaben besteht.) Sie können Ihren Namensraum weiter unterteilen, beispielsweise in eine Teilbibliothek namens `simple`. Der Packagename lautet dann:

```
package net.mindview.simple;
```

Wir ordnen dem Namensraum `net.mindview.simple` nun zwei Klassen `Vector` und `List` zu:

[3] Die Namen von Umgebungsvariablen werden in der Regel durchgängig in Großbuchstaben wiedergegeben.

```
//: net/mindview/simple/Vector.java
// Creating a package.
package net.mindview.simple;

public class Vector {
    public Vector() {
        System.out.println("net.mindview.simple.Vector");
    }
} ///:~
```

Die `package`-Anweisung muß, wie zuvor bereits erwähnt, falls sie existiert, die erste Zeile der Datei sein, die keine Kommentarzeile ist. Die zweite Klasse ist der ersten sehr ähnlich:

```
//: net/mindview/simple/List.java
// Creating a package.
package net.mindview.simple;

public class List {
    public List() {
        System.out.println("net.mindview.simple.List");
    }
} ///:~
```

Beide Datei liegen im selben Unterverzeichnis, bei mir

```
C:\DOC\JavaT\net\mindview\simple
```

(Die erste Kommentarzeile jeder Datei in diesem Buch gibt den Pfad zu dieser Datei in der Quelltextdistribution zu diesem Buch an.)

[24] Wenn Sie den Pfad von rechts nach links lesen, finden Sie den Packagenamen `net.mindview.simple`, aber was ist mit dem Anfangsstück des Pfades? Das Anfangsstück ist im Klassenpfad deklariert, auf meinem Rechner:

```
CLASSPATH=.;D:\JAVA\LIB;C:\DOC\JavaT
```

Wie Sie sehen, kann der Klassenpfad mehrere alternative Suchpfade beinhalten.

[25–26] Die Angabe von *.jar* Dateien im Klassenpfad hat eine Besonderheit: Sie müssen den Namen der *.jar* Datei in den Klassenpfad eintragen, nicht nur den Pfad zu dem Verzeichnis, das die *.jar* Datei enthält. Der folgende Klassenpfad beinhaltet zum Beispiel auch die Datei *grape.jar*:

```
CLASSPATH=.; ...
```

Ist der Klassenpfad korrekt deklariert, so können Sie das folgende Beispiel in einem beliebigen Verzeichnis ablegen:

```
//: access/LibTest.java
// Uses the library.
import net.mindview.simple.*;

public class LibTest {
    public static void main(String[] args) {
        Vector v = new Vector();
        List l = new List();
    }
} /* Output:
      net.mindview.simple.Vector
      net.mindview.simple.List
   *///:~
```

Trifft der Compiler auf die `import`-Anweisung für das Package `net.mindview.simple`, so beginnt er, die im Klassenpfad deklarierten Verzeichnisse nach einem Unterverzeichnis *net/mindview/simple* zu durchsuchen und dieses nach Klassendateien mit passendem Namen (*Vector.class* für `Vector` beziehungsweise *List.class* für `List`). Beachten Sie, daß sowohl die beiden Klassen als auch die gewünschten Methoden (hier die Konstruktoren) in `Vector` und `List` öffentlich sein müssen.

[27] Die Deklaration des Klassenpfades war für Java-Neulinge eine solche Strapaze (für mich jedenfalls, als ich anfing), daß Sun Microsystems das JDK bei den späteren Java-Versionen etwas intelligenter gestaltet hat. Sie können nach der Installation einfache Java-Programme übersetzen und ausführen, auch wenn Sie den Klassenpfad nicht deklarieren. Sie müssen allerdings das Wurzelverzeichnis der Quelltextdistribution (erhältlich bei *http://www.mindview.net* zum Herunterladen) zu diesem Buch in Ihren Klassenpfad aufnehmen, wenn Sie die Beispiele übersetzen und laufen lassen wollen.

Übungsaufgabe 1: (1) Schreiben Sie eine Klasse und ordnen Sie sie einem Package zu. Erzeugen Sie außerhalb des Packages ein Objekt Ihrer Klasse. ■

7.1.2.1 Namenskollisionen

[28] Was geschieht, wenn zwei Packages per `*`-Notation importiert werden, die eine gleichnamige Komponente enthalten, zum Beispiel:

```
import net.mindview.simple.*;
import java.util.*;
```

Da auch das Package `java.util` eine Klasse `Vector` enthält, kann sich eine Namenskollision ergeben. Solange Sie sich allerdings nicht auf die gleichnamige Komponente beziehen, ist alles in Ordnung und das ist gut so, weil Sie andernfalls viel zu schreiben hätten, um Kollisionen zu vermeiden, die nicht eintreten.

[29] Die Kollision tritt allerdings ein, wenn Sie versuchen, ein `Vector`-Objekt zu erzeugen:

```
Vector v = new Vector();
```

Welche `Vector`-Klasse ist gemeint? Weder Compiler noch der Betrachter des Quelltext kann es wissen. Der Compiler gibt daher eine Fehlermeldung aus und verlangt, daß Sie sich unmißverständlich ausdrücken. Die folgende Zeile erzeugt ein Objekt der Klasse `java.util.Vector` aus der Standardbibliothek:

```
java.util.Vector v = new java.util.Vector();
```

Da diese Notation, zusammen mit dem Klassenpfad, den Ort der Klasse `Vector` vollständig beschreibt, ist keine `import`-Anweisung für `java.util.*` erforderlich, solange ich keinen anderen Namen aus diesem Package verwende.

[30] Altenativ können Sie die Namenskollision auch durch Importieren einer einzelnen Klasse vermeiden, solange Sie nicht beide kollidierenden Namen im selben Programm verwenden (in diesem Fall müssen Sie wieder auf voll qualifizierte Namen zurückgreifen).

Übungsaufgabe 2: (1) Setzen Sie die Fragmente aus Unterunterabschnitt 7.1.2.1 in ein Programm ein und verifizieren Sie, daß die Kollisionen tatsächlich auftreten. ■

7.1.3 Die Hilfsbibliothek im Package net.mindview.util

[31] Mit diesem Wissen können Sie nun Ihre eigenen Bibliotheken mit Hilfsklassen und -methoden anlegen, um weniger oder überhaupt keinen Quelltext zu kopieren. Nehmen Sie zum Beispiel die Pseudonyme für `System.out.println()`, um Schreibarbeit einzusparen. Die Klasse `Print` faßt diese pseudonymen Methode zusammen und kann augenfreundlich statisch importiert werden:

```
//: net/mindview/util/Print.java
// Print methods that can be used without
// qualifiers, using Java SE5 static imports:
package net.mindview.util;
import java.io.*;

public class Print {
    // Print with a newline:
    public static void print(Object obj) {
        System.out.println(obj);
    }
    // Print a newline by itself:
    public static void print() {
        System.out.println();
    }
    // Print with no line break:
    public static void printnb(Object obj) {
        System.out.print(obj);
    }
    // The new Java SE5 printf() (from C):
    public static PrintStream
        printf(String format, Object... args) {
        return System.out.printf(format, args);
    }
} ///:~
```

Sie können beliebige Werte ausgeben, entweder mit anschließendem Zeilenumbruch (`print()`) oder ohne (`printnb()`).

[32] Sie haben sicherlich erraten, daß diese Datei in einem Verzeichnis stehen muß, dessen Pfad mit einem im Klassenpfad deklarierten Anfangsstück beginnt und mit *net/mindview/util* weitergeht. Nach dem Übersetzen der Klasse `Print`, können die statischen Methoden `print()` und `printnb()` in jeder Übersetzungseinheit aufgerufen werden, die `Print` statisch importiert:

```
//: access/PrintTest.java
// Uses the static printing methods in Print.java.
import static net.mindview.util.Print.*;

public class PrintTest {
    public static void main(String[] args) {
        print("Available from now on!");
        print(100);
        print(100L);
        print(3.14159);
    }
} /* Output:
      Available from now on!
      100
      100
      3.14159
   *///:~
```

[33] Die in Abschnitt 5.4 vorgestellte Klasse `Range` mit ihren `range()`-Methoden ist eine weitere Komponente der im Package `net.mindview.util` definierten Bibliothek. Die Methoden liefern ganzzahlige Wertebereiche zur leichteren Verwendung der erweiterten `for`-Schleife:

```
//: net/mindview/util/Range.java
// Array creation methods that can be used without
// qualifiers, using Java SE5 static imports:
package net.mindview.util;

public class Range {
    // Produce a sequence [0..n)
    public static int[] range(int n) {
        int[] result = new int[n];
        for(int i = 0; i < n; i++)
            result[i] = i;
        return result;
    }
    // Produce a sequence [start..end)
    public static int[] range(int start, int end) {
        int sz = end - start;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + i;
        return result;
    }
    // Produce a sequence [start..end) incrementing by step
    public static int[] range(int start, int end, int step) {
        int sz = (end - start)/step;
        int[] result = new int[sz];
        for(int i = 0; i < sz; i++)
            result[i] = start + (i * step);
        return result;
    }
} ///:~
```

Sie können nun jede nützliche Funktionalität, die Sie entwickeln, in einer solchen Bibliothek unterbringen. Die `net.mindview.util`-Bibliothek wird im Verlauf dieses Buches immer wieder ergänzt.

7.1.4 Bedingte Übersetzung mit Hilfe von Packages

[34] Die Fähigkeit der *bedingten Übersetzung* von C ist bei Java nicht gegeben. Die bedingte Übersetzung gestattet das Verhalten des übersetzten Quelltextes zu beeinflussen, ohne den Quelltext selbst ändern zu müssen. Eine solche Fähigkeit wurde bei Java vermutlich deshalb nicht berücksichtigt, weil sie bei C am häufigsten zur Lösung plattformübergreifender Probleme verwendet wurde: Verschiedene Teile des Quelltextes mußten abhängig von der Plattform übersetzt werden. Aufgrund der angestrebten Plattformunabhängigkeit von Java war eine solche Fähigkeit nicht erforderlich.

[35] Es gibt aber noch andere nützliche Anwendungsmöglichkeiten für die bedingte Übersetzung, beispielsweise die Fehlersuche. Die entsprechenden Eigenschaften und Fähigkeiten werden während der Entwicklung aktiviert und vor der Auslieferung wieder deaktiviert. Sie können diesen Effekt bewirken, in dem Sie das importierte Package, mit den auf die Fehlersuche ausgerichteten Anweisungen durch das Package für die produktive Version Ihres Programms ersetzen. Diese Vorgehensweise kann bei jeder Art von bedingungsabhängigen Quelltext angewendet werden.

Übungsaufgabe 3: (2) Legen Sie zwei Packages `debug` und `debugoff` an, die identische Klassen mit einer `debug()`-Methode enthalten. Die Methode im Package `debug` gibt ihr `String`-Argument

auf der Konsole aus, die Methode im Package `debugoff` liefert keine Ausgabe. Verwenden Sie eine statische `import`-Anweisung, um die Klasse in ein Testprogramm zu importieren und demonstrieren Sie den Effekt der bedingten Übersetzung. ■

7.1.5 Ursache der Packagewarnungen zur Laufzeit

[36] Denken Sie daran, daß Sie jedesmal beim Anlegen eines Packages mit der Wahl des Packagenamens implizit einen Verzeichnisnamen festlegen. Die Elemente des Packages *müssen* in dem durch den Packagenamen gegebenen Verzeichnis liegen, welches sich unmittelbar an eines der im Klassenpfad deklarierten Verzeichnisse anschließen muß. Das Experimentieren mit dem Schlüsselwort `package` kann anfangs ein wenig frustrierend sein. Wenn Sie sich nicht an die Regel „Packagename als Verzeichnisname“ halten, bekommen Sie zur Laufzeit viele seltsame Fehlermeldungen dahingehend, daß eine bestimmte Klasse nicht gefunden werden kann, obwohl sie im Arbeitsverzeichnis liegt. Wenn Sie eine solche Meldung sehen, versuchen Sie es noch einmal mit auskommentierter `package`-Anweisung. Wenn es dann funktioniert, wissen Sie wo das Problem liegt.

[37] Beachten Sie, daß die übersetzten Klassendateien häufig in einem anderen Verzeichnis stehen, als der Quelltext. Die Laufzeitumgebung muß dennoch in der Lage sein, die Klassendateien über den Klassenpfad zu finden.

7.2 Zugriffsmodifikatoren

[38] Die Java-Zugriffsmodifikatoren `public`, `protected` und `private` werden vor der Definition beziehungsweise Deklaration einer Methode oder eines Feldes in einer Klasse notiert. Der Modifikator gilt nur für den Zugriff auf diese Definition beziehungsweise Deklaration.

[39] Wenn Sie keinen Zugriffsmodifikator angeben, steht die entsprechende Komponente unter „Packagezugriff“. Somit unterliegt jede Komponente einer Klasse der Zugriffskontrolle. In den folgenden vier Unterabschnitten lernen Sie die vier verschiedenen Zugriffstypen kennen.

7.2.1 Packagezugriff (Standardzugriff)

[40] Bei den Beispielen bis zu diesem Kapitel wurden keine Zugriffsmodifikatoren deklariert. Der **Standardzugriff** hat kein Schlüsselwort und wird häufig als **Packagezugriff** bezeichnet. Alle übrigen Klassen im aktuellen Package haben Zugriff auf eine Komponente unter Packagezugriff, während die Komponente allen Klassen außerhalb des aktuellen Packages gegenüber als `private` deklariert zu sein scheint. Da eine Übersetzungseinheit, also eine Datei, nur einem einzigen Package angehören kann, stehen sich alle Klassen einer Übersetzungseinheit unter Packagezugriff automatisch gegenseitig zur Verfügung.

[41] Der Packagezugriff gestattet die Zusammenfassung verwandter Klasse zu einem Package, so daß sie mühelos miteinander interagieren können. Wenn Sie Klassen zu einem Package zusammenfassen, den Komponenten unter Packagezugriff also gegenseitige Erreichbarkeit gewähren, „gehört“ Ihnen der Inhalt dieses Packages. Es ist sinnvoll, nur Klassen die Ihnen gehören, Packagezugriff auf weitere Klassen zu gewähren, die ebenfalls Ihnen gehören. Der Packagezugriff liefert gewissermaßen eine Bedeutung oder einen Grund zur Zusammenfassung von Klassen zu einem Package. Viele Programmiersprachen erlauben Ihnen, Ihre Klassendefinitionen beliebig auf Dateien zu verteilen, aber bei Java müssen Sie Ihre Klassen sinnvoll organisieren. Außerdem möchten Sie eventuell Klassen vom Zugriff auf die dem aktuellen Package zugeordneten Klassen ausschließen.

[42] Eine Klasse kontrolliert, „wer“ Zugriff auf ihre Komponenten erhält. Keine Methode oder Anweisung aus einem anderen Package erhält Zugriff auf die als `protected` oder `private` deklarierten beziehungsweise und Packagezugriff stehenden Komponenten einer Klasse. Es gibt nur die folgenden Möglichkeiten, um eine Komponente erreichbar zu machen:

- Deklarieren Sie die Komponente als `public`. In diesem Fall ist die Komponente von überall erreichbar.
- Stellen Sie die Komponente unter Packagezugriff, indem Sie keinen Zugriffsmodifikator deklarieren und ordnen Sie die anderen Klassen demselben Package zu. Alle Klassen in diesem Package haben nun Zugriff auf die Komponente.
- Sie lernen in Kapitel 8, wenn der Ableitungsmechanismus eingeführt wird, daß eine abgeleitete Klasse Zugriff auf die als `protected` oder `public` deklarierten Komponenten ihrer Basisklasse hat (nicht aber auf deren als `private` deklarierte Komponenten). Eine abgeleitete Klasse kann die Komponenten ihrer Basisklasse, die unter Packagezugriff stehen, nur erreichen, wenn beide Klassen im selben Package liegen. Machen Sie sich aber im Augenblick noch keine Gedanken über Ableitung und das Schlüsselwort `protected`.
- Definieren Sie Abfrage- und Änderungsmethoden (`getXxx()`- und `setXxx()`-Methoden), um den Inhalt des Feldes `Xxx` auszulesen beziehungsweise zu setzen. Im Sinne der objektorientierten Programmierung ist dies der beste Ansatz und außerdem grundlegend für JavaBeans (siehe Abschnitt 23.11).

7.2.2 Der public-Modifikator (öffentliche Schnittstelle einer Klasse)

[43] Das Schlüsselwort `public` bewirkt, daß die unmittelbar folgende Komponente öffentlich und für jedermann verfügbar ist, insbesondere für Clientprogrammierer die diese Klasse verwenden. Angenommen, das Package `access.dessert` beinhaltet die folgende Übersetzungseinheit:

```
//: access/dessert/Cookie.java
// Creates a library.
package access.dessert;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    void bite() { System.out.println("bite"); }
} ///:~
```

[44] Denken Sie daran, daß die vom Compiler aus *Cookie.java* erzeugte Klassendatei in einem Unterverzeichnis namens *dessert* eines Verzeichnisses namens *access* (für Kapitel 7 „Zugriffskontrolle“) liegen muß, welches sich unmittelbar an eines der im Klassenpfad deklarierten Verzeichnisse anschließt. Die Annahme, daß Compiler und Laufzeitumgebung das aktuelle Arbeitsverzeichnis in die Suche einbeziehen, ist falsch. Wenn der Klassenpfad keinen Punkt (`.`) enthält, wird das aktuelle Verzeichnis nicht berücksichtigt.

[45] Das folgende Programm verwendet die Klasse `Cookie`:

```
//: access/Dinner.java
// Uses the library.
import access.dessert.*;

public class Dinner {
    public static void main(String[] args) {
        Cookie x = new Cookie();
```

```
            //! x.bite(); // Can't access
        }
    } /* Output:
          Cookie constructor
       *///:~
```

Das Erzeugen eines `Cookie`-Objektes ist möglich, da sowohl der Konstruktor als auch auch Klasse als `public` deklariert sind. (Das Konzept der öffentlichen Klasse wird später noch genauer behandelt.) Die Methode `bite()` ist dagegen aus der Klasse `Dinner` nicht erreichbar, da sie unter Packagezugriff steht und `Dinner` nicht zum Package `access.dessert` gehört. Der Compiler verhindert, daß Sie die `bite()`-Methode aufrufen.

7.2.2.1 Das Standardpackage

[46] Eventuell werden Sie überrascht sein, daß sich die folgende Klasse `Cake` übersetzen läßt, obwohl die Regeln scheinbar verletzt werden:

```
//: access/Cake.java
// Accesses a class in a separate compilation unit.

class Cake {
    public static void main(String[] args) {
        Pie x = new Pie();
        x.f();
    }
} /* Output:
      Pie.f()
   *///:~
```

Die Klasse `Pie` ist in einer zweiten Datei im selben Verzeichnis definiert:

```
//: access/Pie.java
// The other class.

class Pie {
    void f() { System.out.println("Pie.f()"); }
} ///:~
```

Auf den ersten Blick scheinen diese beiden Dateien voneinander völlig unabhängig zu sein. Dennoch ist die Klasse `Cake` in der Lage, ein `Pie`-Objekt zu erzeugen und seine `f()`-Methode aufzurufen. (Beachten Sie, daß Ihr Klassenpfad einen Punkt beinhalten muß, damit sich die beiden Dateien übersetzen lassen.) Sie nehmen typischerweise an, daß `Pie` und `f()` unter Packagezugriff stehen und `Cake` somit nicht zur Verfügung stehen. Es ist korrekt, daß `Pie` und `f()` unter Packagezugriff stehen. Beide Namen sind aber in der Klasse `Cake` verfügbar, weil die Dateien *Cake.java* und *Pie.java* im selben Verzeichnis stehen und nicht explizit einem Package zugeordnet sind. Java ordnet den Inhalt einer solchen Übersetzungseinheit implizit dem *Standardpackage* für ihr Verzeichnis zu, so daß ihre Klassen und Komponenten gegenüber allen anderen Übersetzungseinheiten in diesem Verzeichnis unter Packagezugriff stehen.

7.2.3 Der private-Modifikator (interne Funktionalität einer Klasse)

[47] Die Deklaration einer Komponente als `private` bewirkt, daß nur die Komponenten der Klasse, zu der die private Komponente gehört, Zugriff auf diese Komponente erhalten. Auch die übrigen Klassen im selben Package haben keinen Zugriff auf die privaten Komponenten einer Klasse. Sie schützen die Klasse gewissermaßen vor sich selbst (Ihnen, dem Programmierer). Es kommt andererseits vor,

daß mehrere zusammenarbeitende Programmierer ein Package zusammenstellen. In diesem Fall gestattet Ihnen die Deklaration privater Komponenten, diese Komponenten zu ändern, ohne sich mit Auswirkungen auf andere Klassen im selben Package auseinandersetzen zu müssen.

[48] Das Standardpackage liefert häufig ausreichende Verdeckung, da Clientprogrammierer, die eine öffentliche Klasse verwenden, die Komponenten dieser Klasse, die unter Packagezugriff stehen, nicht erreichen können. Das ist eine günstige Eigenschaft, da die in der Regel den Standardzugriff verwenden (beziehungsweise bekommen, wenn Sie den Zugriffsmodifikator vergessen). Sie befassen sich typischerweise mit der Frage, welche Komponenten Sie für die Clientprogrammierer explizit als `public` deklarieren sollen und gehen in der Anfangszeit davon aus, daß Sie das Schlüsselwort `private` nur selten anwenden werden, da es vertretbar ist, darauf zu verzichten. Bei Anwesenheit mehrerer Threads stellt sich hingegen heraus, daß der konsistente Gebrauch des `private`-Modifikators sehr wichtig ist (siehe Kapitel 22).

[49–50] Ein Beispiel für die Verwendung des Modifikators `private`:

```
//: access/IceCream.java
// Demonstrates ''private'' keyword.

class Sundae {
    private Sundae() {}
    static Sundae makeASundae() {
        return new Sundae();
    }
}

public class IceCream {
    public static void main(String[] args) {
        //! Sundae x = new Sundae();
        Sundae x = Sundae.makeASundae();
    }
} ///:~
```

Dieses Beispiel zeigt einen praktischen Anwendungsfall für den `private`-Modifikator: Gelegentlich wollen Sie kontrollieren, auf welche Weise ein Objekt erzeugt wird und die Clientprogrammierer daran hindern, einen bestimmten Konstruktor (oder auch alle Konstruktoren) direkt aufzurufen. Im obigen Beispiel können Objekte der Klasse `Sundae` nicht direkt per Konstruktor erzeugt werden, sondern Sie müssen die dazu die statische Methode `makeASundae()` aufrufen.[4]

[51] Jede Methode einer Klasse, die unzweifelhaft nur eine „Hilfsmethode“ innerhalb der Klasse ist, kann als private Methode deklariert werden, um zu garantieren, daß Sie sie nicht versehentlich irgendwo im Package aufrufen und sich damit selbst am Ändern oder Entfernen der Methode hindern. Die Deklaration einer Methode als privat garantiert, daß diese Option erhalten bleibt.

[52] Dasselbe gilt für die Felder einer Klasse. Sie sollten Felder stets als privat deklarieren, solange Sie nicht gezwungen sind, diesen Teil der Implementierung Ihrer Klasse offenzulegen (dieser Schritt ist seltener erforderlich, als Sie vielleicht annehmen). Daß ein Objekt von einem privaten Feld referenziert wird, bedeutet allerdings nicht, daß eine andere Klasse dasselbe Objekt nicht zugleich über ein öffentliches Feld referenzieren kann. (Beachten Sie zu diesem Thema die Online-Anhänge zu diesem Buch unter der Webadresse *http://www.mindview.net*.)

[4] Der private Konstruktor der Klasse `Sundae` hat noch eine andere Auswirkung: Da außer dem *privaten* Standardkonstruktor kein weiterer Konstruktor definiert ist, kann keine weitere Klasse von `Sundae` abgeleitet werden: Die Fehlermeldung des Compilers lautet `Sundae() has private access in Sundae`. (Das Konzept der Ableitung wird im nächsten Kapitel vorgestellt.)

7.2.4 Der protected-Modifikator (Zugriff unter Ableitung)

[53] Das Verständnis des Zugriffsmodifikators `protected` erfordert einen Schritt vorwärts. Sie brauchen diesen Unterabschnitt bis zur Einführung des Ableitungskonzeptes im nächsten Kapitel noch nicht verinnerlicht zu haben. Der Vollständigkeit halber liefert dieser Unterabschnitt dennoch eine kurze Beschreibung und ein Beispiel zur Verwendung des `protected`-Modifikators.

[54] Das Schlüsselwort `protected` hängt mit dem Konzept der *Ableitung* (auch Vererbung) zusammen, welches einer sogenannten *Basisklasse* neue Komponenten hinzufügt, ohne die bereits existierende Basisklasse zu berühren. Es ist aber auch möglich, das Verhalten bereits vorhandener Komponenten der Basisklasse zu ändern. Die folgende Zeile definiert, daß die Klasse `Bar` von der Basisklasse `Foo` abgeleitet ist:

```
class Foo extends Bar {
```

Die restliche Definition der Klasse `Foo` ist wie gewohnt.

[55] Wenn Sie in einem neuen Package eine Klasse anlegen und von einer Basisklasse ableiten, die einem anderen Package angehört, haben Sie lediglich Zugriff auf die öffentlichen Komponenten der Basisklasse. (Liegt die abgeleitete Klasse mit der Basisklasse in einem gemeinsamen Package, so erstreckt sich Ihr Zugriff natürlich auf alle Komponenten unter Packagezugriff.) Gelegentlich möchte der Autor der Basisklasse den abgeleiteten Klassen, nicht aber aller Welt im allgemeinen, Zugriff auf eine bestimmte Komponente gewähren. Diese Funktion erfüllt der `protected`-Modifikator. `protected` deklariert auch Packagezugriff, das heißt Klassen im selben Package haben Zugriff auf als `protected` deklarierte Komponenten.

[56] Die Klasse `ChocolateChip` im folgenden Beispiel darf die unter Packagezugriff stehende `bite()`-Methode der Klasse `Cookie` von Seite 181 nicht aufrufen:

```
//: access/ChocolateChip.java
// Can't use package-access member from another package.
import access.dessert.*;

public class ChocolateChip extends Cookie {
    public ChocolateChip() {
        System.out.println("ChocolateChip constructor");
    }
    public void chomp() {
        //! bite(); // Can't access bite
    }
    public static void main(String[] args) {
        ChocolateChip x = new ChocolateChip();
        x.chomp();
    }
} /* Output:
      Cookie constructor
      ChocolateChip constructor
   *///:~
```

[57] Es ist eine interessante Eigenschaft der Ableitung, daß die `bite()`-Methode der Klasse `Cookie` zwar auch in jeder von `Cookie` abgeleiteten Klasse existiert, aber nicht aus einer abgeleiteten Klasse aufgerufen werden kann, die einem anderen Package als `access.dessert` angehört, da die `bite()`-Methode unter Packagezugriff steht. Die `bite()`-Methode steht also nicht zu unserer Verfügung. Sie könnten die `bite()`-Methode natürlich als öffentliche Methode umdeklarieren, aber dann wäre sie für jedermann erreichbar und dies liegt unter Umständen nicht in Ihrer Absicht. Wenn Sie die Klasse `Cookie` aber folgendermaßen ändern:

```
//: access/cookie2/Cookie.java
package access.cookie2;

public class Cookie {
    public Cookie() {
        System.out.println("Cookie constructor");
    }
    protected void bite() {
        System.out.println("bite");
    }
} ///:~
```

kann die `bite()`-Methode in jeder von `Cookie` abgeleiteten Klassen aufgerufen werden:

```
//: access/ChocolateChip2.java
import access.cookie2.*;

public class ChocolateChip2 extends Cookie {
    public ChocolateChip2() {
        System.out.println("ChocolateChip2 constructor");
    }
    public void chomp() { bite(); } // Protected method
    public static void main(String[] args) {
        ChocolateChip2 x = new ChocolateChip2();
        x.chomp();
    }
} /* Output:
      Cookie constructor
      ChocolateChip2 constructor
      bite
   *////:~
```

Beachten Sie, daß `bite()` zwar auch unter Packagezugriff steht, aber nicht öffentlich ist.

Übungsaufgabe 4: (2) Zeigen Sie, daß eine als `protected` deklarierte Methode zugleich unter Packagezugriff, nicht aber unter öffentlichem Zugriff steht. ■

Übungsaufgabe 5: (2) Schreiben Sie eine Klasse mit Komponenten unter allen vier Zugriffsmodi: `public`, `private`, `protected` und Packagezugriff. Erzeugen Sie ein Objekt dieser Klasse und beobachten Sie die vom Compiler ausgegebenen Meldungen beim Zugriffsversuch auf die einzelnen Komponenten. Beachten Sie, daß Klassen im selben Verzeichnis zum Standardpackage gehören. ■

Übungsaufgabe 6: (1) Schreiben Sie eine Klasse mit einem als `protected` deklarierten Feld. Legen Sie in derselben Datei eine zweite Klasse mit einer Methode an, die den Inhalt des `protected`-Feld der ersten Klasse ändert. ■

7.3 Trennung von Schnittstelle und Implementierung

[58] Die Zugriffskontrolle wird häufig auch als *Verbergen der Implementierung* bezeichnet. Das Verpacken von Feldern und Methoden in einer Klasse in Kombination mit dem Verbergen der Implementierung wird häufig auch als *Kapselung*[5] bezeichnet. Das Ergebnis ist ein Datentyp mit Eigenschaften und Verhalten.

[59] Die Zugriffskontrolle implementiert aus zwei Gründen Abgrenzungen innerhalb eines Datentyps. Erstens, um zu bezeichnen, welche Komponenten die Clientprogrammierer verwenden dürfen und

[5] Teilweise wird auch das Verbergen der Implementierung alleine als Kapselung bezeichnet.

welche nicht. Sie können interne Funktionalität in Ihrer Klasse anlegen, ohne sich Sorgen darüber machen zu müssen, daß die Clientprogrammierer versehentlich eine interne Komponente als Teil der öffentlichen Schnittstelle mißverstehen, die sie eigentlich bedienen sollen.

[60] Dies führt direkt zum zweiten Grund, nämlich der Trennung von Schnittstelle und Implementierung. Wird eine Klasse in mehreren Programmen verwendet, wobei Clientprogrammierer lediglich Methoden der öffentlichen Schnittstelle aufrufen können, so können Sie ohne weiteres jede nicht öffentliche Komponente (das heißt Komponenten unter Packagezugriff sowie als `protected` oder `private` deklarierten Komponenten) ändern, ohne die Funktionstüchtigkeit des abhängigen Programms zu gefährden.

[61] Der Übersichtlichkeit halber können Sie sich der Richtlinie anschließen, in der Definition einer Klasse die öffentlichen Komponenten zuerst anzulegen, danach die `protected`-, Package- beziehungsweise `private`-Komponenten. Diese Vorgehensweise hat für den Leser des Quelltextes den Vorteil, beim Lesen von oben nach unten schnell zu erkennen, was wichtig ist (die öffentlichen Komponenten, da sie außerhalb der Datei erreichbar sind) und bei den ersten nicht öffentlichen Komponenten aufzuhören, da diese zur internen Funktionalität gehören:

```
//: access/OrganizedByAccess.java
public class OrganizedByAccess {
    public void pub1() { /* ... */ }
    public void pub2() { /* ... */ }
    public void pub3() { /* ... */ }
    private void priv1() { /* ... */ }
    private void priv2() { /* ... */ }
    private void priv3() { /* ... */ }
    private int i;
    // ...
} ///:~
```

[62] Das Verständnis der Funktionalität einer Klasse wird hiermit nur teilweise erleichtert, da Schnittstelle und Implementierung noch immer kombiniert sind, schließlich gehört der Quelltext (die Implementierung) zur Definition der Klasse. Die Dokumentation im Javadoc-Format schränkt die Wichtigkeit der Lesbarkeit des Quelltextes für die Clientprogrammierer etwas ein. Das Anzeigen der Schnittstelle einer Klasse für den Konsumenten ist eigentlich die Aufgabe eines Klassenbrowsers, also eines Werkzeuges, das die verfügbaren Klassen untersucht und auf eine nützliche Weise anzeigt, was Sie damit tun können (das heißt welche Komponenten verfügbar sind). Das Betrachten der API-Dokumentation des Java Development Kits im Webbrowser liefert denselben Effekt als ein Klassenbrowser.

7.4 Zugriffsmodifikatoren bei Klassen

[63] Die Zugriffsmodifikatoren können auch verwendet werden, um die Verfügbarkeit einer Klasse in einer Bibliothek den Clientprogrammierern gegenüber zu deklarieren. Soll den Clientprogrammierern eine Klasse zur Verfügung stehen, so stellen Sie der Klassendefinition den Modifikator `public` voran. Damit kontrollieren Sie insbesondere, ob die Clientprogrammierer ein Objekt dieser Klasse erzeugen können.

[64] Der Modifikator muß vor dem Schlüsselwort `class` erscheinen, um den Zugriff auf die Klasse zu kontrollieren:

```
public class Widget {
```

Liegt die Klasse `Widget` im Package `access`, so können die Clientprogrammierer `Widget` durch

```
import access.Widget;
```

oder

```
import access.*;
```

importieren.

[65] Folgende Regeln sind einzuhalten:

- Pro Übersetzungseinheit (*.java* Datei) ist höchstens eine öffentliche Klasse erlaubt. Der Leitgedanke lautet, daß jede Übersetzungseinheit genau eine öffentliche Schnittstelle hat, die durch diese öffentliche Klasse repräsentiert wird. Darüber hinaus kann die Übersetzungseinheit beliebig viele Klassen unter Packagezugriff beinhalten. Umfaßt eine Übersetzungseinheit mehr als eine öffentliche Klasse, so gibt der Compiler eine Fehlermeldung aus.
- Der Name der öffentlichen Klasse muß exakt mit dem Namen der Datei übereinstimmen, die die Übersetzungseinheit enthält, einschließlich Groß-/Kleinschreibung. Die Datei, welche die Klasse `Widget` definiert, muß also *Widget.java* heißen, nicht *widget.java* oder *WIDGET.java*. Wiederum wird zur Übersetzungszeit ein Fehler gemeldet, wenn die Namen nicht übereinstimmen.
- Es ist zulässig, aber untypisch, daß eine Übersetzungseinheit keine öffentliche Klasse beinhaltet. In diesem Fall unterliegt der Dateiname keiner Einschränkung (obwohl eine völlig beliebige Benennung die Leute verwirrt, die den Quelltext lesen und pflegen müssen).

[66] Wie verfahren Sie mit einer Klasse im Package `access`, die Sie nur brauchen, um die Funktionalität der Klasse `Widget` oder einer anderen öffentlichen Klasse in `access` zu unterstützen? Sie wollen keine Dokumentation für Clientprogrammierer schreiben und erwägen, die Klasse später einmal durch eine andere Klasse zu ersetzen. Diese Flexibilität setzt voraus, daß die Clientprogrammierer nicht von Ihren in `access` verborgenen Implementierungsdetails abhängig werden. Sie lassen dazu das Schlüsselwort `public` vor der Klasse fort, stellen die Klasse also unter Packagezugriff. (Die Klasse kann dann nur noch innerhalb des `access`-Packages verwendet werden.)

Übungsaufgabe 7: (1) Legen Sie, den Fragementen zum Importieren des `access`-Packages und der Klasse `Widget` entsprechend, eine Bibliothek an. Erzeugen Sie ein `Widget`-Objekt in einer Klasse, die nicht zum Package `access` gehört. ■

[67] Wenn Sie eine Klasse unter Packagezugriff stellen, ist es nach wie vor sinnvoll, die Felder als privat zu deklarieren (Felder sollten nach Möglichkeit immer privat sein), während die Methoden ebenfalls unter Packagezugriff stehen sollten. Da eine Klasse unter Packagezugriff in der Regel nur innerhalb ihres Packages verwendet wird, müssen Sie die Methoden einer solchen Klassen nur dann als öffentlich deklarieren, wenn es sich nicht verhindern läßt. Der Compiler zeigt diese Fälle an.

[68] Beachten Sie, daß eine Klasse weder als `private` noch als `protected` deklariert werden kann,[6] das heißt Sie haben nur die Wahl zwischen einer öffentlichen Klasse und einer Klasse unter Packagezugriff. Soll niemand außer Ihnen Zugriff auf die Klasse haben, so können Sie alle Konstruktoren als privat deklarieren, damit nur Sie allein über eine statische Methode oder ein statisches Feld ein Objekt dieser Klasse erzeugen können, zum Beispiel:

```
//: access/Lunch.java
// Demonstrates class access specifiers. Make a class
// effectively private with private constructors:

class Soup1 {
```

[6] Innere Klassen können auch als `private` oder `protected` deklariert werden, aber dies sind Speziallfälle. Innere Klassen werden in Kapitel 11 vorgestellt.

```
    private Soup1() {}
    // (1) Allow creation via static method:
    public static Soup1 makeSoup() {
        return new Soup1();
    }
}

class Soup2 {
    private Soup2() {}
    // (2) Create a static object and return a reference
    // upon request.(The ''Singleton'' pattern):
    private static Soup2 ps1 = new Soup2();
    public static Soup2 access() {
        return ps1;
    }
    public void f() {}
}

// Only one public class allowed per file:
public class Lunch {
    void testPrivate() {
        // Can't do this! Private constructor:
        //! Soup1 soup = new Soup1();
    }
    void testStatic() {
        Soup1 soup = Soup1.makeSoup();
    }
    void testSingleton() {
        Soup2.access().f();
    }
} ///:~
```

Die meisten Methoden haben bis jetzt entweder `void` oder einen Wert primitiven Typs zurückgegeben, so daß die Definition

```
public static Soup1 makeSoup() {
    return new Soup1();
}
```

auf den ersten Blick ungewöhnlich wirkt. Das Wort „`Soup1`“ vor dem Methodennamen (`makeSoup()`) gibt an, was die Methode zurückgibt. Der Rückgabetyp der meisten Methoden im Buch bis hier war `void`, das heißt die Methode gibt nichts zurück. Eine Methode kann aber auch eine Objektreferenz liefern. Die `makeSoup()`-Methode gibt eine Referenz auf ein `Soup1`-Objekt zurück.

[69] Die Klassen `Soup1` und `Soup2` zeigen, wie sich die direkte Objekterzeugung durch ausschließlich private Konstruktoren verhindern läßt. Sie erinnern sich, daß der Standardkonstruktor (parameterloser Konstruktor) automatisch erzeugt wird, falls Sie nicht selbst wenigstens einen Konstruktor explizit definieren. Wenn Sie den Standardkonstruktor selbst anlegen, entfällt die automatische Erzeugung. Indem Sie den Konstruktor als privat deklarieren, kann niemand ein Objekt der jeweiligen Klasse erzeugen. Aber wie läßt sich eine solche Klasse verwenden? Das obige Beispiel zeigt zwei Möglichkeiten. Die Klasse `Soup1` definiert eine statische Methode, die ein `Soup1`-Objekt erzeugt und eine Referenz darauf zurückgibt. Dieser Ansatz ist nützlich, wenn Sie einige Operationen auf dem `Soup1`-Objekt veranlassen wollen, bevor die Referenz zurückgegeben wird oder wenn Sie die Anzahl der erzeugten `Soup1`-Objekte kontrollieren möchten (etwa um die Größe der Population zu begrenzen).

[70] Die Klasse `Soup2` implementiert ein sogenanntes *Entwurfsmuster*, das in *Thinking in Patterns, Problem-Solving Techniques using Java* beschrieben wird. Das hier gewählte Entwurfsmuster heißt

Singleton und gestattet nur das Erzeugen eines einzigen Objektes. Das `Soup2`-Objekt wird von einem statischen privaten Feld der Klasse `Soup2` referenziert, so daß nur ein Exemplar existiert. Sie erhalten nur auf einem Weg eine Referenz auf dieses Objekt, nämlich über die öffentliche Methode `access()`.

[71] Eine Klasse ohne Zugriffsmodifikator steht unter Packagezugriff, wie bereits beschrieben wurde. Somit kann jede Klasse, die demselben Package angehört, ein Objekt der ersteren Klasse erzeugen, nicht aber eine Klasse, die außerhalb des Packages liegt. (Denken Sie daran, daß jede Datei ohne explizite `package`-Anweisung im selben Verzeichnis implizit zum Standardpackage dieses Verzeichnisses gehört.) Beinhaltet eine Klasse ohne Zugriffsmodifikator aber eine statische öffentliche Komponente, so haben die Clientprogrammierer Zugriff auf diese Komponente, auch wenn sie kein Objekt der Klasse erzeugen können.

Übungsaufgabe 8: (4) Schreiben Sie eine Klasse `ConnectionManager`, die ein Array von `Connection`-Objekten enthält. Orientieren Sie sich am Beispiel *Lunch.java* (Seite 187). Die Clientprogrammierer dürfen nicht befähigt werden, `Connection`-Objekte explizit zu erzeugen, sondern kann Objektreferenzen nur über eine statische Methode der Klasse `ConnectionManager` anfordern. Hat `ConnectionManager` jedes `Connection`-Objekt aus seinem Array einmal zurückgegeben, so liefert die statische Methode den Wert `null`. Testen Sie die Klassen in der `main()`-Methode. ■

Übungsaufgabe 9: Legen Sie im Verzeichnis *access/local* (voraussichtlich bereits im Klassenpfad deklariert) die folgende Datei an:

```
// access/local/PackagedClass.java
package access.local;

class PackagedClass {
    public PackagedClass() {
        System.out.println("Creating a packaged class");
    }
}
```

Legen Sie die folgende Datei in einem anderen Verzeichnis als *access/local* an:

```
// access/foreign/Foreign.java
package access.foreign;
import access.local.*;

public class Foreign {
    public static void main(String[] args) {
        PackagedClass pc = new PackagedClass();
    }
}
```

Erläutern Sie, warum der Compiler eine Fehlermeldung ausgibt. Würde sich etwas ändern, wenn die Klasse `Foreign` dem Package `access.local` zugeordnet wäre. ■

7.5 Zusammenfassung

[72] In jeder Beziehung sind Grenzen wichtig, die von allen Beteiligten respektiert werden. Beim Anlegen einer Bibliothek entsteht eine Beziehung zu den Clientprogrammierern, die die Bibliothek nutzen. Die Anwender einer Bibliothek sind selbst Programmierer und verwenden die Bibliothek, um eine Anwendung oder eine noch umfangreicherere Bibliothek zu entwickeln.

[73] Ohne Regeln können die Clientprogrammierer mit jeder Komponente einer Klasse tun was sie wollen, auch wenn Sie es vorziehen, daß ein Teil der Komponenten nicht manipuliert wird.

[74] In diesem Kapitel haben wir Eigenschaften von Klassen betrachtet, die in einer Bibliothek von Bedeutung sind: Einerseits die Zusammenfassung von Klassen zu Packages und andererseits die Art und Weise, wie eine Klasse den Zugriff auf ihre Komponenten steuert.

[75] Es gibt Schätzungen dahingehend, daß ein Softwareprojekt in C irgendwo zwischen 50.000 und 100.000 Quelltextzeilen Schiffbruch erleidet, da C nur einen einzigen Namensraum hat, so daß es zu Namenskollisionen kommt, die zusätzlichen Verwaltungsaufwand bedeuten. Bei Java gestatten das Schlüsselwort `package`, die Richtlinie zur Benennung von Packages und die `import`-Anweisung vollständige Kontrolle über die Namen, so daß sich das Problem der Namenskollision mühelos vermeiden läßt.

[76] Die Kontrolle des Zugriffs auf die Komponenten einer Klasse hat zwei Gründe: Einerseits sollen die Clientprogrammierer die Finger von den Dingen lassen, die sie nicht berühren sollen. Diese Komponenten sind zur internen Funktionalität der Klasse erforderlich, gehören aber nicht zur Schnittstelle, die die Clientprogrammierer bedienen. Die Kennzeichnung von Feldern und Methoden als private Komponenten kommt den Clientprogrammierern entgegen, da sie leicht erkennen können, welche Teile einer Klasse relevant sind und welche sie nicht zu beachten brauchen. Die Kennzeichnung vereinfacht das Verständnis der Klasse.

[77] Der zweite und wichtigste Grund für die Zugriffskontrolle besteht darin, dem Designer einer Bibliothek zu gestatten, Änderungen an der internen Struktur seiner Klasse vorzunehmen, ohne sich über Auswirkungen für die Clientprogrammierer Gedanken machen zu müssen. Vielleicht erkennen Sie beispielsweise erst nach der Fertigstellung der ersten Version einer Klasse, daß eine Umstrukturierung Ihres Quelltextes die Verarbeitung erheblich beschleunigen würde. Sind Schnittstelle und Implementierung klar von einander getrennt und vor einander geschützt, so können Sie die Änderung implementieren, ohne die Clientprogrammierer zu zwingen, ihre Implementierungen ebenfalls zu ändern. Zugriffskontrolle gewährleistet, daß die Clientprogrammierer nicht von einem Teil der einer Klasse unterliegenden Implementierung abhängig werden.

[78] Wenn Sie in der Lage sind, die einer Klasse unterliegende Implementierung zu ändern, haben Sie nicht nur die Freiheit, Ihr Design zu verbessern, sondern auch die Freiheit, Fehler zu machen. Fehler lassen sich nicht vermeiden, gleichgültig wieviel Sorgfalt Sie in Planung und Design investieren. Zu wissen, daß Sie solche Fehler relativ sicher machen können, gibt Ihnen mehr experimentellen Freiraum. Sie lernen schneller und schließen Ihre Projekte früher ab.

[79] Die öffentliche Schnittstelle einer Klasse ist der von den Clientprogrammierern wahrgenommene Teil und somit der wichtigste Teil der Klasse, der während der Untersuchungs- und Designphase „richtig“ werden muß. Aber selbst hier haben Sie noch Spielraum. Falls Ihnen die Schnittstelle beim ersten Entwurf nicht gelingt, können Sie zusätzliche Methoden definieren, solange Sie keine Methoden entfernen, die Clientprogrammierer eventuell bereits verwendet haben.

[80] Beachten Sie, daß sich die Zugriffskontrolle auf eine Beziehung (und eine Art von Kommunikation) zwischen dem Autor und den externen Anwendungen einer Bibliothek konzentriert. Es gibt viele Situationen, in denen diese Art zu kommunizieren nicht in Betracht gezogen wird, beispielsweise wenn Sie ein Programm nur für Ihren persönlichen Gebrauch schreiben oder eng mit einem kleinen Team zusammenarbeiten und alle Klassen demselben Package zugeordnet werden. In einer solchen Situation findet eine andere Art von Kommunikation statt und steifes Festhalten an Zugriffsregeln ist unter Umständen nicht optimal. Der Standardzugriff (Packagezugriff) kann völlig ausreichen.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 8

Wiederwendung bereits existierender Klassen

Inhaltsübersicht

[0] Die Wiederverwendung bestehender Klassen ist eine der unwiderstehlichsten Eigenschaften von Java. Revolutionär zu sein, erfordert aber erheblich mehr als das Kopieren und Ändern von Quelltext.

[1] Dieses „Verfahren“ stammt von prozeduralen Sprachen wie C und hat nicht gut funktioniert. Die Lösung basiert auf dem Konzept der Klasse, wie alles in Java. Die Wiederverwendung von Quelltext ereignet sich beim Schreiben neuer Klassen, indem Sie eine neue Klasse nicht von Anfang an selbst entwickeln, sondern eine existierende Klasse „einbinden“, die bereits geschrieben und auf Fehler untersucht worden ist.

[2] Das Ziel besteht darin, existierende Klassen zu nutzen, ohne den bestehenden Quelltext zu verunreinigen. In diesem Kapitel werden zwei Wege vorgestellt, um dieses Ziel zu erreichen. Der erste Ansatz ist unkompliziert: Die neue Klasse referenziert Objekte existierender Klassen. Diese Vorgehensweise wird als *Komposition* bezeichnet, da die neue Klasse aus Objekten anderer Klassen zusammengesetzt („komponiert“) ist. Sie verwenden die Funktionalität der Klasse, nicht aber ihre Form.

[3] Der zweite Ansatz ist ausgetüftelter und erzeugt eine Klasse als Untertyp der existierenden Klasse. Sie übernehmen buchstäblich die Form der existierenden Klasse und ergänzen die Implementierung um weitere Methoden, ohne die existierende Klasse zu verändern. Dieses Konzept wird als *Ableitung* oder auch *Vererbung* bezeichnet und der Compiler verrichtet den größten Teil der Arbeit. Die Ableitung ist eine tragende Säule der objektorientierten Programmierung und hat weitere Auswirkungen, die in Kapitel 9 untersucht werden.

[4] Es wird sich herausstellen, daß Syntax und Verhalten bei Komposition und Ableitung ähnlich sind (das ergibt Sinn, da beide Ansätze dazu dienen, aus existierenden Typen neue Typen zu gewinnen). In diesem Kapitel lernen Sie diese beiden Wiederverwendungsmechanismen kennen.

8.1 Komposition

[5] Der Kompositionansatz ist bis zu dieser Stelle des Buches schon häufig zum Einsatz gekommen. Sie legen in einer neuen Klasse einfach Felder an, die Objekte anderer Klassen referenzieren. Stellen Sie zum Beispiel ein Objekt vor, das einige `String`-Objekte, mehrere Werte primitiven Typs und eine Referenz auf ein Objekt einer beliebigen sonstigen Klasse enthält. Felder nicht primitiven Typs werden mit Objektreferenzen bewertet, Felder primitiven Typs dagegen direkt:

```
//: reusing/SprinklerSystem.java
// Composition for code reuse.

class WaterSource {
    private String s;
    WaterSource() {
        System.out.println("WaterSource()");
        s = "Constructed";
    }
    public String toString() { return s; }
}

public class SprinklerSystem {
    private String valve1, valve2, valve3, valve4;
    private WaterSource source = new WaterSource();
    private int i;
    private float f;
    public String toString() {
        return
            "valve1 = " + valve1 + " " +
            "valve2 = " + valve2 + " " +
            "valve3 = " + valve3 + " " +
            "valve4 = " + valve4 + "\n" +
            "i = " + i + " " + "f = " + f + " " +
            "source = " + source;
    }
    public static void main(String[] args) {
        SprinklerSystem sprinklers = new SprinklerSystem();
        System.out.println(sprinklers);
```

```
    }
} /* Output:
      WaterSource()
      valve1 = null valve2 = null valve3 = null valve4 = null
      i = 0 f = 0.0 source = Constructed
   *///:~
```

[6] Die in beiden Klassen definierte Methode `toString()` hat eine besondere Aufgabe: Diese Methode ist bei jedem Objekt vorhanden und wird aufgerufen, wenn der Compiler ein Objekt zur Verfügung hat, aber ein `String`-Objekt braucht. Der Compiler erkennt beispielsweise in der `toString()`-Methode der Klasse `SprinklerSystem`, daß Sie versuchen, mittels

```
''source = '' + source;
```

ein `String`-Objekt (`''source = ''`) mit einem `WaterSource`-Objekt zu verknüpfen. Da Sie an ein `String`-Objekt nur wiederum ein `String`-Objekt anhängen können, „wandelt“ der Compiler das von `source` referenzierte Objekt durch Aufrufen von dessen `toString()`-Methode in ein `String`-Objekt um. Anschließend können die beiden `String`-Objekte zusammengefaßt und das resultierende `String`-Objekt der Methode `System.out.println()` (oder einer der äquivalenten statischen Hilfsmethoden `print()` oder `printnb()` der Klasse `Print`, siehe Unterabschnitt 7.1.3) übergeben werden. Wenn Sie eine Klasse mit diesem Verhalten ausstatten wollen, genügt es, eine `toString()`-Methode zu definieren.

[7] Felder primitiven Typs werden, wie in Unterunterabschnitt 3.4.1.1 beschrieben, automatisch mit ihrem typspezifischen Initialwert belegt. Felder nicht primitiven Typs werden dagegen mit `null` initialisiert und beim Versuch über eine „frisch initialisierte“ Referenzvariable eine Methode aufzurufen, wird eine Ausnahme ausgeworfen, das heißt ein Laufzeitfehler gemeldet.

[8–9] Es ist sinnvoll, daß der Compiler nicht einfach zu jedem Feld nicht primitiven Typs ein Standardobjekt erzeugt, da dieses Verhalten oft unnötigen Aufwand bedeuten würde. Sie können ein Feld nicht primitiven Typs auf die folgenden vier Arten initialisieren:

- Direkt bei der Felddeklaration. Das Feld wird vor dem Aufruf des Konstruktors initialisiert.
- Im Konstruktor der Klasse zu der das Feld gehört.
- Unmittelbar vor der tatsächlichen Verwendung des Objektes (verzögerte Initialisierung). Diese Variante reduziert die Unkosten in Situationen, in denen die Objekterzeugung teuer ist und nicht immer ein Objekt erzeugt werden muß.
- Mit Hilfe eines dynamischen Initialisierungsblocks.

Das folgende Beispiel zeigt alle vier Möglichkeiten:

```
//: reusing/Bath.java
// Constructor initialization with composition.
import static net.mindview.util.Print.*;

class Soap {
    private String s;
    Soap() {
        print(''Soap()'');
        s = ''Constructed'';
    }
    public String toString() { return s; }
}

public class Bath {
    private String // Initializing at point of definition:
```

```
        s1 = ''Happy'',
        s2 = ''Happy'',
        s3, s4;
    private Soap castille;
    private int i;
    private float toy;
    public Bath() {
        print(''Inside Bath()'');
        s3 = ''Joy'';
        toy = 3.14f;
        castille = new Soap();
    }
    // Instance initialization:
    { i = 47; }
    public String toString() {
        if(s4 == null) // Delayed initialization:
            s4 = ''Joy'';
        return
            ''s1 = '' + s1 + ''\n'' +
            ''s2 = '' + s2 + ''\n'' +
            ''s3 = '' + s3 + ''\n'' +
            ''s4 = '' + s4 + ''\n'' +
            ''i = '' + i + ''\n'' +
            ''toy = '' + toy + ''\n'' +
            ''castille = '' + castille;
    }
    public static void main(String[] args) {
        Bath b = new Bath();
        print(b);
    }
} /* Output:
      Inside Bath()
      Soap()
      s1 = Happy
      s2 = Happy
      s3 = Joy
      s4 = Joy
      i = 47
      toy = 3.14
      castille = Constructed
   *///:~
```

Beachten Sie, daß im Konstruktor der Klasse `Bath` eine `print()`-Anweisung verarbeitet wird, bevor die erste Initialisierung stattfindet. Sofern Sie ein Feld nicht bei seiner Deklaration initialisieren, gibt es (mit Ausnahme der unvermeidlichen Ausnahme zur Laufzeit) keine weitere Garantie dafür, daß Sie sich um die Initialisierung kümmern, bevor Sie die versuchen, die erste Methode aufrufen.

[10] Die `toString()`-Methode bewertet `s4` wenn sie aufgerufen wird, so daß alle Felder zum Zeitpunkt ihrer Verwendung korrekt intialisiert sind.

Übungsaufgabe 1: (2) Schreiben Sie eine einfache Klasse. Deklarieren Sie in einer weiteren Klasse ein Feld zur Aufnahme einer Referenz auf ein Objekt der ersten Klasse. Erzeugen Sie mittels verzögerter Initialisierung ein Objekt dieser Klasse. ■

8.2 Ableitung (Vererbung)

[11] Das Ableitungskonzept ist ein fester Bestandteil von Java und allen anderen objektorientierten Programmiersprachen. Im Grunde genommen ist jedesmal Ableitung im Spiel, wenn Sie eine neue Klasse schreiben, da jede Klasse implizit direkt von der Java-Wurzelklasse `Object` abgeleitet ist, falls Sie Ihre Klasse nicht explizit von einer anderen Klasse ableiten.

[12–13] Die Kompositionsyntax ist offensichtlich, während die Ableitungssyntax eine deutlich andere Form hat. Die Ableitung einer neuen Klasse von einer existierenden Klasse bedeutet, daß die neue Klasse in gewisser Hinsicht der alten ähnelt. Die Beziehung zwischen der neuen und der alten Klasse wird durch das Schlüsselwort `extends` und den Namen der sogenannten *Basisklasse* vor der öffnenden geschweiften Klammer ausgedrückt, welche die Definition der neuen Klasse einleitet. Durch Ableitung gehen alle Felder und Methoden der Basisklasse an die abgeleitete Klasse über. Ein Beispiel:

```
//: reusing/Detergent.java
// Inheritance syntax & properties.
import static net.mindview.util.Print.*;

class Cleanser {
    private String s = ''Cleanser'';
    public void append(String a) { s += a; }
    public void dilute() { append('' dilute()''); }
    public void apply() { append('' apply()''); }
    public void scrub() { append('' scrub()''); }
    public String toString() { return s; }
    public static void main(String[] args) {
        Cleanser x = new Cleanser();
        x.dilute(); x.apply(); x.scrub();
        print(x);
    }
}

public class Detergent extends Cleanser {
    // Change a method:
    public void scrub() {
        append('' Detergent.scrub()'');
        super.scrub(); // Call base-class version
    }
    // Add methods to the interface:
    public void foam() { append('' foam()''); }
    // Test the new class:
    public static void main(String[] args) {
        Detergent x = new Detergent();
        x.dilute();
        x.apply();
        x.scrub();
        x.foam();
        print(x);
        print(''Testing base class:'');
        Cleanser.main(args);
    }
} /* Output:
      Cleanser dilute() apply() Detergent.scrub() scrub() foam()
      Testing base class:
      Cleanser dilute() apply() scrub()
   *///:~
```

Dieses Beispiel demonstriert eine Reihe von Eigenschaften des Ableitungsmechanismus'. Erstens werden in der `append()`-Methode der Klasse `Cleanser` `String`-Objekte mit Hilfe des Operators `+=` verknüpft („konkateniert"), einem der beiden Operatoren den die Designer von Java zur Anwendung auf `String`-Objekte „überladen" haben (der andere Operator ist `+`).

[14] Zweitens definieren sowohl die Klasse `Cleanser` als auch die Klasse `Detergent` je eine `main()`-Methode. Sie können in jeder Ihrer Klassen eine `main()`-Methode anlegen. Dies ist ein praktisches Verfahren zum Testen jeder einzelnen Klasse. Diese `main()`-Methoden müssen nicht entfernt werden, wenn die Arbeit an einer Klasse beendet ist, sondern kann für spätere Tests aufbewahrt werden.

[15] Auch wenn ein Programm aus vielen Klassen besteht, wird nur die `main()`-Methode der Klasse aufgerufen, die auf der Kommandozeile angegeben wird. Das Kommando `java Detergent` bewirkt zum Beispiel, daß die `main()`-Methode der Klasse `Detergent` aufgerufen wird. Das Kommando `java Cleanser` ruft dagegen die `main()`-Methode der Klasse `Cleanser` auf, obwohl `Cleanser` keine öffentliche Klasse ist. Selbst wenn eine Klasse unter Packagezugriff steht, ist eine öffentliche `main()`-Methode erreichbar.

[16] In diesem Beispiel ruft die `main()`-Methode der Klasse `Detergent` explizit die `main()`-Methode der Klasse `Cleanser` auf und übergibt dabei die auf der Kommandozeile übergebenen Argumente (Sie können aber auch ein beliebiges Array von `String`-Objekten übergeben).

[17] Es ist wichtig, daß alle Methoden der Klasse `Cleanser` öffentlich sind. Denken Sie daran, das eine Komponente ohne Zugriffsmodifikatoren unter Packagezugriff steht, also nur Komponenten aus demselben Package Zugriff erlaubt. *Innerhalb des Packages* könnte also „jedermann" diese Methoden aufrufen, wenn sie ohne Zugriffsmodifikatoren definiert wären. Das wäre beispielsweise für `Detergent` kein Problem. Würde aber eine Klasse in einem anderen Package von `Cleanser` abgeleitet, so könnte sie nur die öffentlichen Komponenten von `Cleanser` erreichen. Im allgemeinen sollten Sie, um Ableitung zu berücksichtigen, alle Felder als privat und alle Methoden als öffentlich deklarieren. (Auch als `protected` deklarierte Felder gestatten Zugriff durch abgeleitete Klassen, siehe Abschnitt 8.6.) In einzelnen Klassen können Anpassungen notwendig sein, aber dies ist eine nützliche Richtlinie.

[18] Die Schnittstelle der Klasse `Cleanser` besteht aus fünf Methoden: `append()`, `dilute()`, `apply()`, `scrub()` und `toString()`. Da `Detergent` von `Cleanser` abgeleitet ist (durch das Schlüsselwort `extends`), beinhaltet auch die Schnittstelle von `Detergent` diese fünf Methoden, obwohl sie nicht explizit in `Detergent` definiert sind. Die Ableitung ist in diesem Sinne eine Wiederverwendung der Basisklasse.

[19] Wie Sie an der Methode `scrub()` sehen, kann eine in der Basisklasse definierte Methode in der abgeleiteten Klasse modifiziert werden. Hier soll die `scrub()`-Methode aus der Basisklasse in der Version von `scrub()` in der abgeleiteten Klasse aufgerufen werden. Sie können allerdings in `scrub()` nicht einfach `scrub()` aufrufen, da dies einen rekursiven Methodenaufruf bewirken würde, nicht aber den eigentlich erwünschten Effekt. Java hält zur Lösung dieses Problems das Schlüsselwort `super` bereit, welches die (direkte) Basisklasse der aktuellen abgeleiteten Klasse referenziert. Der Ausdruck `super.scrub()` ruft also die Basisklassenversion der `scrub()`-Methode auf.

[20–21] Sie sind beim Ableiten einer Klasse nicht auf die Verwendung der Methoden aus der Basisklasse beschränkt, sondern können in der abgeleiteten Klasse neue Methoden definieren, wie in jeder anderen Klasse. Die Methode `foam()` ist ein Beispiel hierfür. Die `main()`-Methode der Klasse `Detergent` zeigt, daß `Detergent` sowohl alle Methoden der Klasse `Cleanser` als auch alle Methoden der Klasse `Detergent` aufrufen kann (das heißt `foam()`).

Übungsaufgabe 2: (2) Leiten Sie eine neue Klasse von `Detergent` ab. Überschreiben Sie `scrub()` und definieren Sie eine zusätzliche Methode `sterilize()`. ■

8.2.1 Initialisierung des „Unterobjektes“ der Basisklasse

[22] Da nun statt einer Klasse, zwei Klassen im Spiel sind, nämlich die Basisklasse und die abgeleitete Klasse, kann der Versuch, sich ein Objekt der abgeleiteten Klasse vorzustellen, einen leichten Eindruck von Unübersichtlichkeit bewirken. Von außen betrachtet, haben die neue Klasse und ihre Basisklasse dieselbe Schnittstelle, wobei die abgeleitete Klasse eventuell über einige zusätzliche Methoden und Felder verfügt. Ein Objekt der abgeleiteten Klasse beinhaltet ein *Unterobjekt* der Basisklasse, welches einem gewöhnlichen Objekt der Basisklasse selbst entspricht. Von außen betrachtet ist das Objekt der Basisklasse im Objekt der abgeleiten Klasse verpackt.

[23–24] Es ist natürlich wichtig, daß das Unterobjekt der Basisklasse korrekt initialisiert wird und es gibt nur eine Möglichkeit, diese Initialisierung zu garantieren: Bewerkstelligen Sie die Initialisierung im Konstruktor, indem Sie den Konstruktor der Basisklasse aufrufen, der über die erforderlichen „Kenntnisse“ und Berechtigungen verfügt, um das Unterobjekt der Basisklasse zu initialisieren. Der Compiler setzt automatisch einen Aufruf des Basisklassenkonstruktors in den Konstruktor der abgeleiteten Klasse ein. Das folgende Beispiel hat drei Ableitungsebenen:

```
//: reusing/Cartoon.java
// Constructor calls during inheritance.
import static net.mindview.util.Print.*;

class Art {
    Art() { print(''Art constructor''); }
}

class Drawing extends Art {
    Drawing() { print(''Drawing constructor''); }
}

public class Cartoon extends Drawing {
    public Cartoon() { print(''Cartoon constructor''); }
    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
} /* Output:
      Art constructor
      Drawing constructor
      Cartoon constructor
   *///:~
```

Sie können erkennen, daß die Objekterzeugung von der Basisklasse an „nach außen“ gerichtet ist, so daß das Unterobjekt der Basisklasse initialisiert ist, bevor der Konstruktor der abgeleiteten Klasse Zugriff darauf erhält. Auch wenn Sie in der Klasse `Cartoon` keinen Konstruktor definieren, erzeugt der Compiler einen Standardkonstruktor, der den entsprechenden Konstruktor der Basisklasse aufruft.

Übungsaufgabe 3: (2) Beweisen Sie den letzten Satz im vorigen Abschnitt. ■

Übungsaufgabe 4: (2) Zeigen Sie, daß der Konstruktor der Basisklasse immer aufgerufen wird und daß der Aufruf vor dem Konstruktor der abgeleiteten Klasse erfolgt. ■

Übungsaufgabe 5: (1) Schreiben Sie zwei Klassen `A` und `B` mit Standardkonstruktoren (parameterlose Konstruktoren), die sich jeweils durch eine Textmeldung bemerkbar machen. Leiten Sie eine neue Klasse `C` von `A` ab und legen Sie in `C` ein Feld an, welches ein `B`-Objekt referenziert. Definieren Sie für die Klasse `C` keinen Konstruktor. Erzeugen Sie ein Objekt der Klasse `C` und beobachten wie die Ergebnisse. ■

8.2.1.1 Parameterbehaftete Konstruktoren

[25] Das vorige Beispiel verwendet Standardkonstruktoren, das heißt Konstruktoren ohne Parameter. Das Aufrufen des Basisklassenkonstruktors ist für den Compiler kein Problem, da keine Argumente übergeben werden müssen. Hat eine Basisklasse aber keinen Standardkonstruktor oder wollen Sie einen parameterbehafteten Basisklassenkonstruktor aufrufen, so müssen Sie den Aufruf des Konstruktors der Basisklasse mit Hilfe des Schlüsselwortes `super` explizit bewerkstelligen und eine passende Argumentliste übergeben:

```
//: reusing/Chess.java
// Inheritance, constructors and arguments.
import static net.mindview.util.Print.*;

class Game {
    Game(int i) {
        print("Game constructor");
    }
}

class BoardGame extends Game {
    BoardGame(int i) {
        super(i);
        print("BoardGame constructor");
    }
}

public class Chess extends BoardGame {
    Chess() {
        super(11);
        print("Chess constructor");
    }
    public static void main(String[] args) {
        Chess x = new Chess();
    }
} /* Output:
      Game constructor
      BoardGame constructor
      Chess constructor
   *///:~
```

[26] Wenn sie den Basisklassenkonstruktor im Konstruktor der Klasse `BoardGame` nicht explizit aufrufen, beschwert sich der Compiler, daß er keinen Konstruktor der Form `Game()` finden kann. Der explizite Aufruf des Basisklassenkonstruktors *muß*, falls vorhanden, die erste Anweisung des Konstruktors der abgeleiteten Klasse sein. (Andernfalls gibt der Compiler eine entsprechende Fehlermeldung aus.)

Übungsaufgabe 6: (1) Beweisen Sie die Aussagen im vorigen Absatz am Beispiel *Chess.java*. ■

Übungsaufgabe 7: (1) Ändern Sie Übungsaufgabe 5 (Seite 197), so daß die Klassen `A` und `B` anstelle von Standardkonstruktoren Konstruktoren mit Argumenten haben. Legen Sie in der Klasse `C` einen Konstruktor an und führen Sie alle Initialisierungen in diesem Konstruktor aus. ■

Übungsaufgabe 8: (1) Schreiben Sie eine Basisklasse, die nur einen Nicht-Standardkonstruktor hat und eine abgeleitete Klasse, die sowohl einen Standard- als auch einen Nicht-Standardkonstruktor hat. Rufen Sie aus beiden Konstruktoren der abgeleiteten Klasse den Konstruktor der Basisklasse auf. ■

Übungsaufgabe 9: (2) Schreiben Sie eine Klasse `Root`, die je ein Objekt der Klassen `Component1`, `Component2` und `Component3` referenziert (legen Sie auch diese drei Klassen an). Leiten Sie eine Klasse `Stem` von `Root` ab, die ebenfalls je ein Objekt der Klassen `Component1`, `Component2` und `Component3` referenziert. Jede Klasse hat einen Standardkonstruktor, der eine Meldung darüber ausgibt, zu welche Klasse er gehört. ■

Übungsaufgabe 10: (1) Ändern Sie Übungsaufgabe 9, so daß jede Klasse nur einen Nicht-Standardkonstruktor hat. ■

8.3 Delegation

[27] Es gibt noch einen dritten Wiederverwendungsansatz, der von Java nicht direkt unterstützt wird, nämlich die Delegation. Die Delegation liegt in der Mitte zwischen Ableitung und Komposition, indem Sie in der neuen Klasse ein Objekt einer anderen Klasse referenzieren (wie bei der Komposition) und die neue Klasse zugleich alle Methoden des referenzierten Objektes exponiert (wie bei der Ableitung). Das folgende Beispiel zeigt die Steuereinheit eines Raumschiffes:

```
//: reusing/SpaceShipControls.java
public class SpaceShipControls {
    void up(int velocity) {}
    void down(int velocity) {}
    void left(int velocity) {}
    void right(int velocity) {}
    void forward(int velocity) {}
    void back(int velocity) {}
    void turboBoost() {}
} ///:~
```

Ein Möglichkeit die Steuereinheit in ein Raumschiff einbauen ist die Ableitung:

```
//: reusing/SpaceShip.java
public class SpaceShip extends SpaceShipControls {
    private String name;
    public SpaceShip(String name) { this.name = name; }
    public String toString() { return name; }
    public static void main(String[] args) {
        SpaceShip protector = new SpaceShip("NSEA Protector");
        protector.forward(100);
    }
} ///:~
```

[28–29] Aber eigentlich ist ein Raumschiff selbst keine Steuereinheit, auch wenn Sie dem Raumschiff die „Anweisung erteilen“ können, sich vorwärts zu bewegen. Es ist präziser, zu sagen, daß ein Raumschiff eine Steuereinheit *enthält* und das Raumschiff gleichzeitig Aufrufe aller Methoden der Steuereinheit entgegennimmt. Das Problem läßt sich per Delegation lösen:

```
//: reusing/SpaceShipDelegation.java
public class SpaceShipDelegation {
    private String name;
    private SpaceShipControls controls =
        new SpaceShipControls();
    public SpaceShipDelegation(String name) {
        this.name = name;
    }
    // Delegated methods:
```

```
    public void back(int velocity) {
        controls.back(velocity);
    }
    public void down(int velocity) {
        controls.down(velocity);
    }
    public void forward(int velocity) {
        controls.forward(velocity);
    }
    public void left(int velocity) {
        controls.left(velocity);
    }
    public void right(int velocity) {
        controls.right(velocity);
    }
    public void turboBoost() {
        controls.turboBoost();
    }
    public void up(int velocity) {
        controls.up(velocity);
    }
    public static void main(String[] args) {
        SpaceShipDelegation protector =
            new SpaceShipDelegation(''NSEA Protector'');
        protector.forward(100);
    }
} ///:~
```

Die Steuerungsmethoden werden an das unterliegende `SpaceShipControls`-Objekt übertragen und die Schnittstelle der Klasse `SpaceShip` entspricht der Schnittstelle bei Ableitung. Der Delegationsansatz gestattet Ihnen aber mehr Einflußnahme, da Sie zum Beispiel nur einen Teil der Methoden des Komponentenobjektes über die Schnittstelle der äußeren Klasse anbieten können.

[30] Obwohl Java den Delegationsansatz nicht unterstützt, finden Sie diese Unterstützung bei Entwicklungswerkzeugen. Das obige Beispiel wurde mit der Entwicklungsumgebung IntelliJ IDEA von JetBrains automatisch generiert.

Übungsaufgabe 11: (3) Ändern Sie das Beispiel *Detergent.java* (Seite 195), so daß es den Delegationsansatz implementiert. ■

8.4 Kombination von Komposition und Ableitung

[31–32] Komposition und Ableitung werden häufig gemeinsam verwendet. Das folgende Beispiel zeigt die Kombination von Komposition und Ableitung, zusammen mit der erforderlichen Initialisierung durch die Konstruktoren, an einem etwas komplexeren Beispiel:

```
//: reusing/PlaceSetting.java
// Combining composition & inheritance.
import static net.mindview.util.Print.*;

class Plate {
    Plate(int i) {
        print(''Plate constructor'');
    }
}
```

```
class DinnerPlate extends Plate {
    DinnerPlate(int i) {
        super(i);
        print("DinnerPlate constructor");
    }
}

class Utensil {
    Utensil(int i) {
        print("Utensil constructor");
    }
}

class Spoon extends Utensil {
    Spoon(int i) {
        super(i);
        print("Spoon constructor");
    }
}

class Fork extends Utensil {
    Fork(int i) {
        super(i);
        print("Fork constructor");
    }
}

class Knife extends Utensil {
    Knife(int i) {
        super(i);
        print("Knife constructor");
    }
}

// A cultural way of doing something:
class Custom {
    Custom(int i) {
        print("Custom constructor");
    }
}

public class PlaceSetting extends Custom {
    private Spoon sp;
    private Fork frk;
    private Knife kn;
    private DinnerPlate pl;
    public PlaceSetting(int i) {
        super(i + 1);
        sp = new Spoon(i + 2);
        frk = new Fork(i + 3);
        kn = new Knife(i + 4);
        pl = new DinnerPlate(i + 5);
        print("PlaceSetting constructor");
    }
    public static void main(String[] args) {
        PlaceSetting x = new PlaceSetting(9);
    }
} /* Output:
      Custom constructor
      Utensil constructor
```

```
    Spoon constructor
    Utensil constructor
    Fork constructor
    Utensil constructor
    Knife constructor
    Plate constructor
    DinnerPlate constructor
    PlaceSetting constructor
*///:~
```

Der Compiler erzwingt zwar die Initialisierung des Unterobjektes der Basisklasse unmittelbar am Anfang des Konstruktors der abgeleiteten Klasse, wacht andererseits aber nicht darüber, daß die Kompositionsfelder mit Referenzen auf die Komponentenobjekte bewertet sind, das heißt Sie müssen selbst darauf achten.

[33] Die saubere Trennung der Klassen ist verblüffend. Sie brauchen nicht einmal den Quelltext der Methoden, um sie wiederverwenden zu können. Sie müssen höchstens ein Package importieren. (Dies gilt sowohl für die Ableitung als auch für die Komposition.)

8.4.1 Sauberes Aufräumen garantieren

[34] Java hat das Konzept des Destruktors, das heißt einer Methode, die beim Zerstören eines Objektes automatisch aufgerufen wird, nicht von C++ übernommen. Das liegt wahrscheinlich an der Praxis, Objekte bei Java einfach zu vergessen statt zu zerstören und der automatischen Speicherbereinigung zu gestatten, den beanspruchten Arbeitsspeicher bei Bedarf freizugeben.

[35] Diese Vorgehensweise ist in der Regel in Ordnung, aber es gibt Situationen, in denen eine Klasse während ihres Lebenszyklus' Operationen ausführt, die Aufräumarbeiten erforderlich machen. Wie Sie in Abschnitt 6.5 gelernt haben, können Sie nicht wissen, wann die automatische Speicherbereinigung aufgerufen wird beziehungsweise ob der Aufruf überhaupt erfolgt. Sind bei einer Klasse Aufräumarbeiten notwendig, so müssen Sie explizit eine entsprechende Methode definieren und die Clientprogrammierer darüber informieren, daß diese Methode aufgerufen werden muß. Darüber hinaus müssen Sie solche Aufräumarbeiten, wie in Abschnitt 13.8 beschrieben, in eine `finally`-Klausel setzen, um ihre Ausführung zu garantieren.

[36–37] Das folgende Beispiel zeigt ein CAD-System (Computer-Aided Design), welches geometrische Figuren auf dem Bildschirm darstellt:

```
//: reusing/CADSystem.java
// Ensuring proper cleanup.
package reusing;
import static net.mindview.util.Print.*;

class Shape {
    Shape(int i) { print("Shape constructor"); }
    void dispose() { print("Shape dispose"); }
}

class Circle extends Shape {
    Circle(int i) {
        super(i);
        print("Drawing Circle");
    }
    void dispose() {
        print("Erasing Circle");
        super.dispose();
```

```
    }
}

class Triangle extends Shape {
    Triangle(int i) {
        super(i);
        print(''Drawing Triangle'');
    }
    void dispose() {
        print(''Erasing Triangle'');
        super.dispose();
    }
}

class Line extends Shape {
    private int start, end;
    Line(int start, int end) {
        super(start);
        this.start = start;
        this.end = end;
        print(''Drawing Line: '' + start + '', '' + end);
    }
    void dispose() {
        print(''Erasing Line: '' + start + '', '' + end);
        super.dispose();
    }
}

public class CADSystem extends Shape {
    private Circle c;
    private Triangle t;
    private Line[] lines = new Line[3];
    public CADSystem(int i) {
        super(i + 1);
        for(int j = 0; j < lines.length; j++)
            lines[j] = new Line(j, j*j);
        c = new Circle(1);
        t = new Triangle(1);
        print(''Combined constructor'');
    }
    public void dispose() {
        print(''CADSystem.dispose()'');
        // The order of cleanup is the reverse
        // of the order of initialization:
        t.dispose();
        c.dispose();
        for(int i = lines.length - 1; i >= 0; i--)
            lines[i].dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        CADSystem x = new CADSystem(47);
        try {
            // Code and exception handling...
        } finally {
            x.dispose();
        }
    }
} /* Output:
```

```
    Shape constructor
    Shape constructor
    Drawing Line: 0, 0
    Shape constructor
    Drawing Line: 1, 1
    Shape constructor
    Drawing Line: 2, 4
    Shape constructor
    Drawing Circle
    Shape constructor
    Drawing Triangle
    Combined constructor
    CADSystem.dispose()
    Erasing Triangle
    Shape dispose
    Erasing Circle
    Shape dispose
    Erasing Line: 2, 4
    Shape dispose
    Erasing Line: 1, 1
    Shape dispose
    Erasing Line: 0, 0
    Shape dispose
    Shape dispose
*///:~
```

Jede Klasse dieses CAD-Systems ist von der Klasse `Shape` abgeleitet (`Shape` ist wiederum implizit von der Wurzelklasse `Object` abgeleitet). Jede abgeleitete Klasse überschreibt die `dispose()`-Methode von `Shape` und ruft mittels `super` die Basisklassenversion von `dispose()` auf. Die Konstruktoren der drei abgeleiteten Klassen `Circle`, `Triangle` und `Line` „zeichnen“ die jeweilige Figur auf dem Bildschirm. Aber auch jede andere, während des Lebenszyklus' eines Objektes aufgerufene Methode kann eine Operation ausführen, die Aufräumarbeiten erforderlich macht. Jede Klasse hat eine individuelle `dispose()`-Methode, um speicherunabhängige Dinge in den Zustand vor der Existenz des Objektes zurücksetzen zu können.

[38] Die `main()`-Methode enthält zwei neue Schlüsselwörter, die erst in Kapitel 13 erklärt werden: `try` und `finally`. Das Schlüsselwort `try` zeigt an, daß der folgende, durch ein Paar geschweifter Klammern definierte Block ein *geschützter Bereich* ist, das heißt einer besonderen Behandlung bedarf. Ein Teil dieser Sonderbehandlung besteht darin, daß die Anweisungen in der `finally`-Klausel *immer* verarbeitet werden, gleichgültig auf welche Weise die `try`-Klausel verlassen wird. (Die Ausnahmebehandlung gestattet das Verlassen einer `try`-Klausel auf mehreren ~~non-ordinary~~ Wegen.) Die `finally`-Klausel bewirkt im obigen Beispiel, daß die `dispose()`-Methode des von `x` referenzierten Objektes stets aufgerufen wird, unabhängig davon was geschieht.

[39] Beachten Sie, daß Sie die `dispose()`-Methoden der Basisklasse und der abgeleiteten Klassen in einer bestimmten Reihenfolgen aufrufen müssen, wenn ein Unterobjekt von einem anderen abhängt. Im allgemeinen sollten Sie die Reihenfolge anwenden, die der Compiler von C++ den Destruktoren auferlegt: Zuerst führen Sie die zum Aufräumen erforderlichen Schritte in der umgekehrten Reihenfolge bezüglich der Objekterzeugung durch. (Dies setzt in der Regel voraus, daß die Komponenten des Objektes der Basisklasse noch „lebendig“ sind.) Anschließend rufen Sie die Aufräummethode der Basisklasse auf (siehe oben).

[40] Das Aufräumen ist in vielen Fällen unproblematisch und kann der automatischen Speicherbereinigung überlassen werden. Beim expliziten Aufräumen ist Gewissenhaftigkeit und Aufmerksamkeit notwendig, da Sie sich hinsichtlich der Speicherbereinigung nur auf wenige Dinge verlassen können.

Eventuell wird die Speicherbereinigung überhaupt nicht aufgerufen. Wird die Speicherbereinigung aufgerufen, so haben Sie keinen Einfluß auf die Reihenfolge, in der die Objekte aufgeräumt werden. Sie können sich bei der automatischen Speicherbereinigung nur auf die Freigabe von Arbeitsspeicher verlassen. Sind Aufräumarbeiten erforderlich, so schreiben Sie eine eigene Aufräummethode und verzichten Sie auf die Finalisierung (`finalize()`).

Übungsaufgabe 12: (3) Ergänzen Sie die Klassenhierarchie in Übungsaufgabe 9 (Seite 199) um `dispose()`-Methoden. ■

8.4.2 Überladene Methoden bleiben bei Ableitung erhalten

[41–42] Definiert eine Basisklasse in Java eine mehrfach überladene Methode, so bewirkt eine Neudefinition in einer abgeleiteten Klasse (im Gegensatz zu C++) *nicht*, daß die Versionen in der Basisklasse verborgen werden. Die Methode gilt in der abgeleiteten Klasse als überladen, unabhängig davon, ob sie auf dieser Ebene oder in einer darüber liegenden Basisklasse definiert wurde:

```
//: reusing/Hide.java
// Overloading a base-class method name in a derived
// class does not hide the base-class versions.
import static net.mindview.util.Print.*;

class Homer {
    char doh(char c) {
        print("doh(char)");
        return 'd';
    }
    float doh(float f) {
        print("doh(float)");
        return 1.0f;
    }
}

class Milhouse {}

class Bart extends Homer {
    void doh(Milhouse m) {
        print("doh(Milhouse)");
    }
}

public class Hide {
    public static void main(String[] args) {
        Bart b = new Bart();
        b.doh(1);
        b.doh('x');
        b.doh(1.0f);
        b.doh(new Milhouse());
    }
} /* Output:
      doh(float)
      doh(char)
      doh(float)
      doh(Milhouse)
  *///:~
```

Alle Versionen der überladenen Methode in der Klasse `Homer` sind in der Klasse `Bart` verfügbar, obwohl `Bart` eine weitere Version definiert (bei C++ wären die Versionen in der Basisklasse dadurch

verborgen). Wie Sie im nächsten Kapitel lernen werden, kommt das Überschreiben von Methoden in abgeleiteten Klassen mit der/dem in der Basisklasse definierten Signatur und Rückgabetyp häufig vor. Die Verteilung einer überladenen Methode auf mehr als eine Klasse kann andererseits Verwirrung auslösen (und ist bei C++ aus diesem Grund nicht erlaubt, damit Sie nichts tun, das sich wahrscheinlich als Fehler herausstellt).

[43] Version 5 der Java Standard Edition (SE 5) hat die Annotation `@Override` eingeführt, die eigentlich kein Schlüsselwort ist, aber wie eines verwendet werden kann. Die Annotation dient zur bewußten Kennzeichnung einer Methode, die Sie überschreiben wollen. Der Compiler gibt eine Fehlermeldung aus, wenn Sie die Methode unabsichtlich überladen statt überschreiben:

```
//: reusing/Lisa.java
// {CompileTimeError} (Won't compile)

class Lisa extends Homer {
    @Override
    void doh(Milhouse m) {
        System.out.println("doh(Milhouse)");
    }
} ///:~
```

Der Kommentar `{CompileTimeError}` schließt die Datei *Lisa.java* vom Ant-Skript zur Quelltextdistribution dieses Buches aus. Wenn Sie das Programm manuell übersetzen, bekommen Sie die folgende Fehlermeldung:

```
method does not override a method from its superclass
```

Die Annotation `@Override` schützt Sie also davor, versehentlich eine Methode zu überladen.

Übungsaufgabe 13: (1) Schreiben Sie eine Klasse mit einer dreifach überladenen Methode. Leiten Sie eine neue Klasse ab und überladen Sie die Methode dort ein weiteres Mal. Zeigen Sie, daß alle vier Methoden in der abgeleiten Klasse verfügbar sind. ■

8.5 Entscheidung: Komposition oder Ableitung (Teil 1 von 2)

[44] Sowohl beim Kompositions- als auch beim Ableitungsansatz platzieren Sie ein Unterobjekt im Objekt der neuen Klasse (bei Komposition explizit, bei Ableitung implizit). Wodurch unterscheiden sich die Ansätze und unter welchen Bedingungen sollten Sie den einen dem anderen Ansatz vorziehen?

[45] Der Kompositionsansatz wird gewählt, um die Eigenschaften und Fähigkeiten der existierenden Klasse, nicht aber die Schnittstelle, in der neuen Klasse zu nutzen. Sie betten ein Objekt der existierenden Klasse in die neue Klasse ein, um mit Hilfe seines Verhaltenes die Funktionalität der neuen Klasse zu implementieren. Die Clientprogrammierer sehen aber die Schnittstelle der neuen Klasse statt der Schnittstelle der Klasse des eingebetteten Objektes. Das eingebettete Objekt wird von einem privaten Feld referenziert.

[46] Es ist gelegentlich sinnvoll, den Clientprogrammierern direkten Zugriff auf die von Ihrer Klasse referenzierten Objekte zu gewähren, das heißt die entsprechenden Felder als öffentlich zu deklarieren. Da die Komponentenobjekte ihre Implementierungen selbst verbergen, ist dieser Zugriffsmodus sicher. Wenn der Clientprogrammierer weiß, daß Ihre Klasse aus mehreren Teilen zusammengesetzt ist, läßt sich die Schnittstelle leichter verstehen. Eine Klasse, die Autos repräsentiert, ist ein gutes Beispiel:

```
//: reusing/Car.java
// Composition with public objects.

class Engine {
    public void start() {}
    public void rev() {}
    public void stop() {}
}

class Wheel {
    public void inflate(int psi) {}
}

class Window {
    public void rollup() {}
    public void rolldown() {}
}

class Door {
    public Window window = new Window();
    public void open() {}
    public void close() {}
}

public class Car {
    public Engine engine = new Engine();
    public Wheel[] wheel = new Wheel[4];
    public Door
        left = new Door(),
        right = new Door(); // 2-door
    public Car() {
        for(int i = 0; i < 4; i++)
            wheel[i] = new Wheel();
    }
    public static void main(String[] args) {
        Car car = new Car();
        car.left.window.rollup();
        car.wheel[0].inflate(72);
    }
} ///:~
```

[47] Weil der Zusammenbau des Autos in diesem Fall zur Untersuchung des Problems gehört (und nicht einfach ein Teil des unterliegenden Designs ist), helfen die als öffentlich deklarierten Komponentenobjekte dem Clientprogrammierer, die Verwendung der Klasse zu verstehen und der Autor der Klasse kann auf Komplexität verzichten. Beachten Sie aber, daß dies ein Spezialfall ist und Sie Felder im allgemeinen als privat deklarieren sollten.

[48] Beim Ableiten wählen Sie eine existierende Klasse und legen eine spezialisierte Version davon an. Das bedeutet im allgemeinen, daß Sie eine universelle Klasse an einen bestimmten Bedarf anpassen. Wenn Sie etwas darüber nachdenken, werden Sie erkennen, daß es nicht sinnvoll wäre, ein Auto durch Komposition mit einem Fahrzeug zu modellieren, da ein Auto kein Fahrzeug *enthält*, sondern ein Fahrzeug *ist*. Das Ableitungskonzept drückt eine „ist ein"-Beziehung aus, das Kompositionskonzept dagegen eine „hat ein"-Beziehung.

Übungsaufgabe 14: (1) Definieren Sie in der Klasse `Engine` im Beispiel *Car.java* eine `service()`-Methode und rufen Sie sie in der `main()`-Methode auf. ■

8.6 Der protected-Modifikator

[49] Nun, nachdem Sie das Ableitungskonzept kennengelernt haben, nimmt die Funktion das Schlüsselwort `protected` Gestalt an. Unter idealen Voraussetzungen würde das Schlüsselwort `private` ausreichen. In realen Projekten kommt es aber vor, daß Sie etwas im Großen und Ganzen vor der Welt verbergen, den Komponenten abgeleiteter Klasse aber den Zugriff darauf erlauben möchten.

[50] Das Schlüsselwort `protected` ist ~~a mod to~~ Pragmatismus und deklariert die damit bezeichnete Komponente den Clientprogrammierern gegenüber als `private`, abgeleiteten Klassen gegenüber sowie innerhalb des Packages aber als erreichbar (`protected` schließt den Packagezugriff ein).

[51–52] Es ist zwar möglich, ein Feld als `protected` zu deklarieren, aber die beste Vorgehensweise besteht darin, alle Felder privat zu belassen. Sie sollten sich stets das Recht sichern, die Ihrer Klasse unterliegende Implementierung zu ändern. Anschließend können Sie durch als `protected` deklarierte Methoden abgeleiteten Klassen kontrollierten Zugriff gewähren:

```
//: reusing/Orc.java
// The protected keyword.
import static net.mindview.util.Print.*;

class Villain {
    private String name;
    protected void set(String nm) { name = nm; }
    public Villain(String name) { this.name = name; }
    public String toString() {
        return ''I'm a Villain and my name is '' + name;
    }
}

public class Orc extends Villain {
    private int orcNumber;
    public Orc(String name, int orcNumber) {
        super(name);
        this.orcNumber = orcNumber;
    }
    public void change(String name, int orcNumber) {
        set(name); // Available because it's protected
        this.orcNumber = orcNumber;
    }
    public String toString() {
        return ''Orc '' + orcNumber + '': '' + super.toString();
    }
    public static void main(String[] args) {
        Orc orc = new Orc(''Limburger'', 12);
        print(orc);
        orc.change(''Bob'', 19);
        print(orc);
    }
} /* Output:
      Orc 12: I'm a Villain and my name is Limburger
      Orc 19: I'm a Villain and my name is Bob
   *///:~
```

Die `change()`-Methode kann `set()` aufrufen, da `set()` als `protected` deklariert ist. Beachten Sie außerdem, daß die `toString()`-Methode der Klasse `Orc` die `toString()`-Methode der Basisklasse `Villain` aufruft.

Übungsaufgabe 15: (2) Schreiben Sie eine Klasse mit einer als `protected` deklarierten Methode

und ordnen Sie die Klasse einem Package zu. Rufen Sie diese Methode außerhalb des Packages auf und erklären Sie das Ergebnis. Leiten Sie nun eine neue Klasse von Ihrer ersten Klasse ab und rufen Sie die als `protected` deklarierte Methode in der abgeleiteten Klasse auf. ■

8.7 Aufwärtsgerichtete Typumwandlung (Teil 1 von 2)

[53] Der wichtigste Aspekt der Ableitung ist nicht, daß Methoden an die neue Klasse vererbt werden, sondern der Ausdruck der Beziehung zwischen Basisklasse und abgeleiteter Klasse. Diese Beziehung bedeutet zusammengefaßt, daß die abgeleitete Klasse auch dem Typ der Basisklasse entspricht.

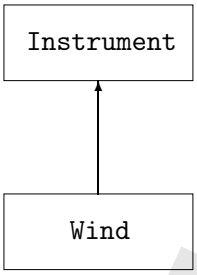
[54] Diese Beschreibung ist nicht einfach eine phantasievolle Erklärung des Ableitungsmechanismus', sondern wird von der Programmiersprache direkt unterstützt. Stellen Sie sich zum Beispiel eine Klasse `Instrument` vor, die Musikinstrumente repräsentiert und eine abgeleitete Klasse `Wind` für Blasinstrumente. Da Ableitung bedeutet, daß alle Methoden der Basisklasse auch der abgeleiteten Klasse zur Verfügung stehen, kann jede Nachricht an ein Objekt der Basisklasse auch an ein Objekt der abgeleiteten Klasse gesendet werden. Hat `Instrument` eine `play()`-Methode, so auch `Wind`. Wir können also zutreffend sagen, daß ein `Wind`-Objekt auch dem Typ `Instrument` angehört. Das folgende Beispiel zeigt, wie der Compiler dieses Konzept unterstützt:

```
//: reusing/Wind.java
// Inheritance & upcasting.

class Instrument {
    public void play() {}
    static void tune(Instrument i) {
        // ...
        i.play();
    }
}

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    public static void main(String[] args) {
        Wind flute = new Wind();
        Instrument.tune(flute); // Upcasting
    }
} ///:~
```

[55] Das interessante an diesem Beispiel ist die Methode `tune()`, die eine Referenz auf ein Objekt vom Typ `Instrument` erwartet. Allerdings ruft die `main()`-Methode der Klasse `Wind` die `tune()`-Methode mit einer Referenz auf ein `Wind`-Objekt auf. Wenn Java nun bei der Typprüfung kleinlich ist, so ist es sonderbar, daß eine Methode, die einen bestimmten Argumenttyp erwartet, bereitwillig ein Argument eines anderen Typs akzeptiert, bis Sie erkennen, daß ein `Wind`-Objekt auch ein `Instrument`-Objekt ist und die `tune()`-Methode keine Methode der Klasse `Instrument` aufrufen kann, die nicht ebenfalls in `Wind` definiert ist. Die Anweisungen in der `tune()`-Methode funktionieren für Objekte vom Typ `Instrument` und jeder von `Instrument` abgeleiteten Klasse. Die Konvertierung einer Referenz vom Typ `Wind` in eine Referenz vom Typ `Instrument` wird als *aufwärtsgerichtete Typumwandlung* bezeichnet.

8.7.1 Die Herkunft der Bezeichnung „aufwärtsgerichtet“

[56] Die Bezeichnung „aufwärtsgerichtet“ stammt von der Art und Weise, in der Ableitungsdiagramme für Klassenhierarchien traditionell gezeichnet werden. Die Wurzelklasse steht am Kopfende der Seite und die Hierarchie wächst nach unten. (Natürlich können Sie Ihre Diagramm auf jede Weise zeichnen, die Ihnen hilfreich erscheint.) Die folgende Abbildung zeigt das Diagramm für *Wind.java*:



[57] Die Umwandlung eines abgeleiteten Typs in einen Basistyp verläuft im Ableitungsdiagramm *aufwärts* und wird daher häufig als *aufwärtsgerichtete Typumwandlung* bezeichnet. Eine aufwärtsgerichtete Typumwandlung ist stets sicher, da Sie von einem spezialisierten zu einem allgemeineren Typ übergehen. Die abgeleitete Klasse kann mehr Methoden definieren als ihre Basisklasse, muß aber zumindest die in der Basisklasse definierten Methoden beinhalten. Die Schnittstelle einer Klasse kann bei einer aufwärtsgerichteten Typumwandlung nur Methoden verlieren, aber nicht hinzu gewinnen. Aus diesem Grund erlaubt der Compiler die aufwärtsgerichtete Typumwandlung ohne explizite Umwandlungssyntax oder eine andere Notation.

[58] Die Typumwandlung ist auch in die entgegengesetzte Richtung möglich und wird in diesem Fall als *abwärtsgerichtete Typumwandlung* bezeichnet, beinhaltet aber ein Dilemma, das ~~im folgenden Kapitel~~ sowie in Kapitel 15 weiter untersucht wird.

8.7.2 Entscheidung: Komposition oder Ableitung (Teil 2 von 2)

[59] Die häufigste Art und Weise, Quelltext zu schreiben und zu verwenden, besteht in der objektorientierten Programmierung darin, Felder und Methoden in einer Klasse zusammenzufassen und Objekte dieser Klasse zu erzeugen. Die Wiederverwendung existierender Klassen per Komposition kommt ebenfalls häufig vor. Der Ableitungsansatz tritt dagegen seltener auf. Auch wenn die Ableitung beim Erlernen der objektorientierten Programmierung viel Aufmerksamkeit erhält, sollen Sie sie nicht überall einsetzen, wo es möglich ist. Sie sollten die Ableitung, ganz im Gegenteil, sparsam gebrauchen und nur dann verwenden, wenn es offensichtlich sinnvoll ist. Eine der besten Hilfen bei der Entscheidung zwischen Komposition und Ableitung ist die Frage, ob Sie jemals eine Referenz auf ein Objekt Ihrer neuen Klasse in den Typ der Basisklasse umwandeln müssen. Ist diese Typumwandlung erforderlich, so ist Ableitung notwendig. Besteht dagegen kein Bedarf für die Typumwandlung, so sollten Sie sich genau überlegen, wozu Sie Ableitung brauchen. Kapitel 9 „Polymorphie: Dynamische Bindung“ liefert einen zwingenden Grund für die aufwärtsgerichtete Typumwandlung. Wenn Sie an die Frage nach der Typumwandlung erinnern, haben Sie aber ein gutes Hilfsmittel für die Entscheidung zwischen Komposition und Ableitung.

Übungsaufgabe 16: (2) Schreiben Sie eine Klasse namens `Amphibian`. Leiten Sie von `Amphibian` eine Klasse namens `Frog` ab. Definieren Sie in der Basisklasse einige passende Methoden. Erzeugen Sie in der `main()`-Methode ein Objekt der Klasse `Frog`, wandeln Sie die Referenz auf dieses Objekt in den Typ `Amphibian` um und zeigen Sie, daß nach wie vor alle Methoden (der Klasse `Amphibian`)

aufgerufen werden können. ■

Übungsaufgabe 17: (1) Ändern Sie Übungsaufgabe 16, so daß die Klasse `Frog` die in der Basisklasse definierten Methoden überschreibt (die Methoden neu definiert und dabei dieselben Signaturen verwendet). Beobachten Sie, was in der `main()`-Methode geschieht. ■

8.8 Der final-Modifikator

[60] Die Wirkung des Modifikators `final` ändert sich geringfügig je nach Kontext, aber eine mit `final` deklarierte Komponente kann im allgemeinen nicht verändert werden. Es gibt zwei Gründe, Änderungen zu vermeiden: Design und Effizienz. Da diese beiden Gründe recht unterschiedlich sind, kann das Schlüsselwort `final` mißbraucht werden.

[61] Die folgenden drei Unterabschnitte diskutieren die drei Stellen, an denen der `final`-Modifikator verwendet werden kann: Bei Feldern, Methoden und Klassen.

8.8.1 Finale Felder

[62] Viele Programmiersprachen haben die Möglichkeit, dem Compiler mitzuteilen, daß etwas „konstant“ ist. Es gibt zwei Arten von Konstanten:

- Ein Wert kann bereits zur Übersetzungszeit konstant sein.
- Eine Wert kann zur Laufzeit konstant sein.

Bei einer **Konstanten zur Übersetzungszeit** (*compile-time constant*) kann der Compiler in sämtlichen Berechnungen statt des Namens der Konstanten ihren Wert einsetzen, das heißt die Berechnung kann bereits zur Übersetzungszeit ausgeführt und ein Teil der andernfalls zur Laufzeit anfallenden Unkosten vermieden werden. Diese Art von Konstanten sind bei Java Felder oder lokale Variablen primitiven Typs und sind mit Modifikator `final` deklariert. Eine solche Konstante muß bei ihrer Deklaration initialisiert werden.

[63] Ein statisches und finales Feld ist mit nur einer Stelle im Arbeitsspeicher verknüpft, deren Inhalt nicht verändert werden kann.

[64] Die Wirkung des `final`-Modifikators auf eine Referenzvariable statt eines Feldes oder einer lokalen Variablen primitiven Typs kann auf den ersten Blick verwirren. Bei einem Feld oder einer lokalen Variablen primitiven Typs macht `final` den *Inhalt* (Wert) konstant, bei einer Referenzvariablen dagegen die *Referenz*. Ist eine finale Referenzvariable einmal mit einer Objektreferenz initialisiert, so kann sie nicht mehr geändert werden, um auf ein anderes Objekt zu verweisen. Das referenzierte Objekt selbst, kann dennoch modifiziert werden. Java hat keine Möglichkeit, ein beliebiges Objekt als konstant zu deklarieren. (Sie können allerdings eine Klasse so schreiben, daß ihre Objekte sich effektiv wie Konstanten verhalten.) Diese Einschränkung gilt auch für Arrays, die selbst Objekte sind.

[65] Das folgende Beispiel demonstriert finale Felder. Beachten Sie, daß die Namen statischer finaler Felder (das heißt Konstanten bezüglich der Übersetzungszeit) durchgängig in Großbuchstaben geschrieben werden und mehrere Worte durch Unterstriche voneinander getrennt werden:

```
//: reusing/FinalData.java
// The effect of final on fields.
import java.util.*;
import static net.mindview.util.Print.*;
```

```
class Value {
    int i; // Package access
    public Value(int i) { this.i = i; }
}

public class FinalData {
    private static Random rand = new Random(47);
    private String id;
    public FinalData(String id) { this.id = id; }
    // Can be compile-time constants:
    private final int valueOne = 9;
    private static final int VALUE_TWO = 99;
    // Typical public constant:
    public static final int VALUE_THREE = 39;
    // Cannot be compile-time constants:
    private final int i4 = rand.nextInt(20);
    static final int INT_5 = rand.nextInt(20);
    private Value v1 = new Value(11);
    private final Value v2 = new Value(22);
    private static final Value VAL_3 = new Value(33);
    // Arrays:
    private final int[] a = { 1, 2, 3, 4, 5, 6 };
    public String toString() {
        return id + ": " + "i4 = " + i4 + ", INT_5 = " + INT_5;
    }
    public static void main(String[] args) {
        FinalData fd1 = new FinalData("fd1");
        //! fd1.valueOne++; // Error: can't change value
        fd1.v2.i++; // Object isn't constant!
        fd1.v1 = new Value(9); // OK - not final
        for(int i = 0; i < fd1.a.length; i++)
            fd1.a[i]++; // Object isn't constant!
        //! fd1.v2 = new Value(0); // Error: Can't
        //! fd1.VAL_3 = new Value(1); // change reference
        //! fd1.a = new int[3];
        print(fd1);
        print("Creating new FinalData");
        FinalData fd2 = new FinalData("fd2");
        print(fd1);
        print(fd2);
    }
} /* Output:
      fd1: i4 = 15, INT_5 = 18
      Creating new FinalData
      fd1: i4 = 15, INT_5 = 18
      fd2: i4 = 13, INT_5 = 18
   *///:~
```

[66] Da `valueOne` und `VALUE_TWO` finale Felder primitiven Typs mit zur Übersetzungszeit definierten Werten sind, kommen beide als Konstanten bezüglich der Übersetzungszeit in Frage und unterscheiden sich nicht wesentlich voneinander. Das Feld `VALUE_THREE` führt die typische Art und Weise vor, in der solche Konstanten deklariert werden: öffentlich, so daß die Konstante auch außerhalb des Packages verwendet werden kann, statisch, so daß nur ein Exemplar dieser Konstanten existiert und final, um das Feld als Konstante zu deklarieren. Konventionsgemäß werden die Namen statischer finaler Felder primitiven Typs mit konstantem Initialwert (also Konstanten bezüglich der Übersetzungszeit) durchgängig mit Großbuchstaben benannt, wobei mehrere Worte durch Unterstriche getrennt werden. (Wie bei Konstanten in C. Die Konvention stammt von dort.)

[67] Die Deklaration eines Feldes mit dem `final`-Modifikator bedeutet keineswegs, daß der Wert der Konstanten bereits zur Übersetzungszeit feststeht, wie die zur Laufzeit mit Zufallswerten initialisierten Felder `i4` und `INT_5` zeigen. Dieser Abschnitt des Beispiels dokumentiert außerdem den Unterschied zwischen einem statischen und einem nicht statischen finalen Feld. Dieser Unterschied ist nur bei Konstanten sichtbar, die erst zur Laufzeit initialisiert werden, da der Compiler die zur Übersetzungszeit initialisierten Felder are treated the same (und voraussichtlich „wegoptimiert"). Der Unterschied zeigt sich beim Aufrufen des Programms: Beachten Sie, daß das `i4`-Feld bei den `FinalData`-Objekten `fd1` und `fd2` unterschiedliche Werte enthält, während der Wert von `INT_5` durch das Erzeugen eines neuen `FinalData`-Objektes nicht verändert wird, weil das statische `INT_5`-Feld nur einmal beim Laden der Klasse `FinalData` initialisiert wird und nicht einmal pro neuem Objekt.

[68] Die Felder `v1`, `v2` und `VAL_3` demonstrieren die Wirkung finaler Referenzvariablen. Die Deklaration von `v2` als finales Feld bedeutet keinesfalls, daß Sie die Eigenschaft `i` des von `v2` referenzierten `Value`-Objektes nicht ändern können (siehe `main()`-Methode). Die `final`-Deklaration bezieht sich auf den Feldinhalt, also die Referenz auf das `Value`-Objekt: Sie können `v2` keine Referenz auf ein anderes `Value`-Objekt zuweisen. Der `final`-Modifikator hat bei Referenzvariablen für Arrayobjekte dieselbe Wirkung (siehe Feld `a`). (Mir ist keine Möglichkeit bekannt, das referenzierte Arrayobjekt selbst als `final` zu deklarieren.) Die Deklaration finaler Referenzvariablen scheint weniger nützlich zu sein, als die Deklaration finaler Felder oder lokaler Variablen.

Übungsaufgabe 18: (2) Schreiben Sie eine Klasse mit einem statischen finalen Feld und einem (nicht statischen) finalen Feld. Führen Sie den Unterschied zwischen beiden vor. ■

8.8.1.1 Finale Felder ohne unmittelbare Initialisierung

[69–70] Java gestattet die Deklaration finaler Felder ohne Initialwert. Der Compiler garantiert die Initialisierung ein solches Feldes vor seiner ersten Verwendung. Die Deklaration eines nicht initialisierten finalen Feldes liefert mehr Flexibilität hinsichtlich der Verwendung des `final`-Modifikators, da Sie beispielsweise ein finales Feld bei jedem Objekt individuell bewerten können, ohne auf die Unveränderlichkeit verzichten zu müssen. Ein Beispiel:

```
//: reusing/BlankFinal.java
// ''Blank'' final fields.

class Poppet {
    private int i;
    Poppet(int ii) { i = ii; }
}

public class BlankFinal {
    private final int i = 0; // Initialized final
    private final int j; // Blank final
    private final Poppet p; // Blank final reference
    // Blank finals MUST be initialized in the constructor:
    public BlankFinal() {
        j = 1; // Initialize blank final
        p = new Poppet(1); // Initialize blank final reference
    }
    public BlankFinal(int x) {
        j = x; // Initialize blank final
        p = new Poppet(x); // Initialize blank final reference
    }
    public static void main(String[] args) {
        new BlankFinal();
        new BlankFinal(47);
```

```
    }
} ///:~
```

Sie sind gezwungen, einem finalen Feld entweder bei seiner Deklaration oder in jedem Konstruktor der Klasse einen Wert zuzuweisen. Auf diese Weise garantiert der Compiler, daß das finale Feld vor seiner Verwendung initialisiert ist.

Übungsaufgabe 19: (2) Schreiben Sie eine Klasse mit einem nicht initialisierten Feld zur Aufnahme einer Objektreferenz. Initialisieren Sie dieses Feld in den Konstruktoren Ihrer Klasse. Zeigen Sie, daß ein finales Feld vor Gebrauch garantiert initialisiert ist sowie daß ein einmal initialisiertes finales Feld nicht mehr geändert werden kann. ■

8.8.1.2 Finale Argumente bei Methoden

[71–72] Java gestattet die Deklaration finaler Argumente in der Argumentliste einer Methode. Das Verweisziel eines solchen Argumentes kann innerhalb der Methode nicht geändert werden:

```
//: reusing/FinalArguments.java
// Using ''final'' with method arguments.

class Gizmo {
    public void spin() {}
}

public class FinalArguments {
    void with(final Gizmo g) {
        //! g = new Gizmo(); // Illegal - g is final
    }
    void without(Gizmo g) {
        g = new Gizmo(); // OK - g not final
        g.spin();
    }
    // void f(final int i) { i++; } // Can't change
    // You can only read from a final primitive:
    int g(final int i) { return i + 1; }
    public static void main(String[] args) {
        FinalArguments bf = new FinalArguments();
        bf.without(null);
        bf.with(null);
    }
} ///:~
```

Die Methoden `f()` und `g()` zeigen die Möglichkeiten beim Umgang mit finalen Argumenten primitiven Typs: Sie können das Argument abfragen aber nicht ändern. Diese Eigenschaft ist wichtig für die Übergabe von Daten an anonyme innere Klassen (siehe Abschnitt 11.6).

8.8.2 Finale Methoden

[73] Es gibt zwei Motivationen, um eine Methode als final zu deklarieren. Erstens, um die Methode „zu sperren“, so daß ihre Funktionalität in abgeleiteten Klassen nicht geändert werden kann. Eine solche Sperre wird aus designbezogenen Gründen installiert, um sicherzustellen, daß das Verhalten der Methoden bei Ableitung erhalten bleibt und nicht überschrieben werden kann.

[74] Zweitens, aus Effizienzgründen: Bei früheren Java-Versionen konnten Sie dem Compiler durch die Deklaration einer Methode als final ermöglichen, sämtliche Aufrufe dieser Methode in „Inline-

Aufrufe“ umzuwandeln. Erkannte der Compiler eine finale Methode, so konnte er, nach eigenem Ermessen, anstelle der normalen Prozedur des Methodenaufrufs (deponieren der Argumente auf dem Aufrufstapel, Sprung zum Methodenkörper und anschließende Verarbeitung, Rücksprung, Aufräumen des Aufrufstapels und Behandeln des Rückgabewertes) eine Kopie der Anweisungen im Methodenkörper einsetzen. Dadurch konnte der Aufwand des Methodenaufrufs vermieden werden. Bei einer umfangreichen Methode kann das Volumen des ausführbaren Codes durch solche „Inline-Aufrufe“ natürlich stark anwachsen, so daß Sie unter Umständen keinen Performanzgewinn bemerken, da die durch `final` bewirkte Optimierung relativ zur Ausführungszeit der Methode nicht mehr ins Gewicht fällt.

[75] Bei moderneren Java-Versionen ist die Laufzeitumgebung (vor allem die HotSpot-Technologie) in der Lage, solche Situationen zu erkennen und die zusätzliche Indirektion „wegzuoptimieren“. Die Verwendung des `final`-Modifikators zur Unterstützung der Optimierung ist daher eigentlich nicht mehr notwendig und im allgemeinen wird davon abgeraten. Ab der SE 5/6 sollten Sie Effizienzangelegenheiten dem Compiler und der Laufzeitumgebung überlassen und den `final` Modifikator nur bei Methoden verwenden, deren Überschreibung Sie explizit verhindern wollen.[1]

8.8.2.1 Private Methoden sind implizit final und können nicht überschrieben werden

[76] Jede private Methode in einer Klasse ist implizit final. Da Sie in einer abgeleiteten Klasse keinen Zugriff auf eine private Methode haben, können Sie sie auch nicht überschreiben. Sie können eine private Methode explizit als final deklarieren, der Modifikator hat hier aber keine zusätzliche Bedeutung.

[77] Dieser Eigenschaft kann Verwirrung auslösen, da das Überschreiben einer privaten (implizit finalen) Methode scheinbar möglich ist und der Compiler keine Fehlermeldung ausgibt:

```
//: reusing/FinalOverridingIllusion.java
// It only looks like you can override
// a private or private final method.
import static net.mindview.util.Print.*;

class WithFinals {
    // Identical to ''private'' alone:
    private final void f() { print(''WithFinals.f()''); }
    // Also automatically ''final'':
    private void g() { print(''WithFinals.g()''); }
}

class OverridingPrivate extends WithFinals {
    private final void f() {
        print(''OverridingPrivate.f()'');
    }
    private void g() {
        print(''OverridingPrivate.g()'');
    }
}

class OverridingPrivate2 extends OverridingPrivate {
    public final void f() {
        print(''OverridingPrivate2.f()'');
    }
```

[1] Fallen Sie nicht dem Drang zur verfrühten Optimierung zum Opfer. Wenn Sie ein Programm zur Funktionstüchtigkeit entwickelt haben und feststellen, daß es zu langsam läuft, ist zweifelhaft, daß Ihnen eine Lösung mit `final`-Modifikator gelingt. Unter der Webadresse *http://www.mindview.net/Books/BetterJava* finden Sie Informationen über Profiling. Dieses Verfahren *kann* Ihnen helfen, Ihr Programm zu beschleunigen.

```
    public void g() {
        print("OverridingPrivate2.g()");
    }
}

public class FinalOverridingIllusion {
    public static void main(String[] args) {
        OverridingPrivate2 op2 = new OverridingPrivate2();
        op2.f();
        op2.g();
        // You can upcast:
        OverridingPrivate op = op2;
        // But you can't call the methods:
        //! op.f();
        //! op.g();
        // Same here:
        WithFinals wf = op2;
        //! wf.f();
        //! wf.g();
    }
} /* Output:
      OverridingPrivate2.f()
      OverridingPrivate2.g()
   *///:~
```

[78] Das Überschreiben einer Methode ist nur möglich, wenn die Methode zur Schnittstelle der Basisklasse gehört, das heißt Sie müssen in der Lage sein, eine Referenz auf ein Objekt der abgeleiten Klasse aufwärts in den Typ der Basisklasse umzuwandeln und dieselbe Methode aufzurufen (die Hintergründe klären sich im nächsten Kapitel). Eine private Methode gehört allerdings nicht zur Schnittstelle der Basisklasse, sondern stellt nur einen in der Klasse verborgenen Teil der Implementierung dar, dessen Name zufällig mit dem Namen einer Methode in einer abgeleiteten Klasse übereinstimmt. Es besteht keinerlei Verbindung zwischen einer privaten Methode der Basisklasse und einer gleichnamigen, als `public` oder `protected` deklarierten beziehungsweise unter Packagezugriff stehenden Methode in einer abgeleiteten Klasse. Sie haben die Methode in der abgeleiteten Klasse nicht überschrieben, sondern lediglich eine neue Methode definiert. Da eine private Methode nicht erreichbar und effektiv unsichtbar ist, wird sie nirgends einbezogen, außer in der Organisation des Quelltextes der Klasse, in der sie definiert ist.

Übungsaufgabe 20: (1) Zeigen Sie, daß die Annotation `@Override` das in diesem Unterunterabschnitt beschriebene Problem löst. ■

Übungsaufgabe 21: (1) Schreiben Sie eine Klasse mit einer finalen Methode. Leiten Sie eine neue Klasse von der ersten Klasse ab und versuchen Sie diese Methode zu überschreiben. ■

8.8.3 Finale Klassen

[79–80] Wenn Sie eine ganze Klasse als final deklarieren, indem Sie der Definition den Modifikator `final` voranstellen, legen Sie fest, daß Sie selbst keine weitere Klasse von dieser Klasse ableiten werden und dies auch Clientprogrammierern nicht erlauben möchten. Sie wollen, mit anderen Worten, Unterklassen aus Sicherheitsgründen vermeiden oder weil Sie Ihre Klasse hinsichtlich ihres Designs für derart beschaffen halten, daß keine Änderungen erforderlich werden können.

```
//: reusing/Jurassic.java
// Making an entire class final.

class SmallBrain {}
```

```
final class Dinosaur {
    int i = 7;
    int j = 1;
    SmallBrain x = new SmallBrain();
    void f() {}
}

//! class Further extends Dinosaur {}
// error: Cannot extend final class 'Dinosaur'

public class Jurassic {
    public static void main(String[] args) {
        Dinosaur n = new Dinosaur();
        n.f();
        n.i = 40;
        n.j++;
    }
} ///:~
```

Beachten Sie, daß die Felder einer finalen Klasse final sein können oder nicht, je nach Ihrer Wahl. Dasselbe gilt für finale Felder hinsichtlich der Frage, ob die Klasse selbst final ist oder nicht. Nachdem eine finale Klasse nicht abgeleitet werden kann, sind allerdings alle Methoden implizit final, da es keine Möglichkeit gibt, sie zu überschreiben. Die können eine Methode einer finalen Klasse explizit als final deklarieren, bekommen dadurch aber keine zusätzliche Wirkung.

Übungsaufgabe 22: (1) Schreiben Sie eine finale Klasse und versuchen Sie, eine weitere Klasse abzuleiten. ■

8.8.4 Vorsicht beim Gebrauch des final-Modifikators

[81] Während des Designs einer Klasse kann es sinnvoll erscheinen, eine Methode als final zu deklarieren. Sie haben vielleicht den Eindruck, daß niemand eine Ihrer Methoden überschreiben wollen könnte. Dies ist (nur) manchmal zutreffend.

[82] Seien Sie mit Ihren Annahmen aber vorsichtig. Im allgemeinen ist es schwierig, die Wiederverwendungsmöglichkeiten einer Klasse vorauszusehen, insbesondere bei einer universell anwendbaren Klasse. Deklarieren Sie eine Methode als final, so verhindern Sie unter Umständen die Wiederverwendung Ihrer Klasse per Ableitung im Projekt eines Clientprogrammierers nur dadurch, daß Sie sich diese Anwendungssituation nicht vorstellen konnten.

[83] Die Standardbibliothek von Java liefert ein gutes Beispiel: Die seit Java 1.0/1.0 vorhandene Klasse `java.util.Vector` wurde häufig verwendet und hätte sich vielleicht als noch nützlicher erwiesen, wären nicht, um der Effizienz willen, sämtliche Methoden als final deklariert worden. Sie können sich ohne weiteres vorstellen, daß es attraktiv ist, eine so fundamentale und nützliche Klasse abzuleiten und die Funktionalität durch Überschreiben von Methoden anzupassen. Die Designer der Standardbibliothek sind allerdings irgendwie zu der Entscheidung kommen, daß sich die Klasse `Vector` hierfür nicht eignet. Das ist aus zwei Gründen ironisch. Einerseits ist die Klasse `java.util.Stack` von `Vector` abgeleitet, das heißt zwischen beiden Klassen besteht eine „ist ein"-Beziehung, die vom logischen Standpunkt aus betrachtet nicht gegeben ist. Insbesondere aber, ist `Stack` ein Beispiel für eine Klasse, die die Java-Designer selbst von `Vector` abgeleitet haben. An dem Punkt, an dem die Designer die Klasse `Stack` in dieser Weise definiert haben, hätten sie erkennen sollen, daß finale Methoden zu restriktiv sind.

[84] Andererseits sind die meisten wichtigen Methoden der Klasse `Vector` synchronisiert, zum Beispiel `addElement()` und `elementAt()`. Wie Sie in Kapitel 22 lernen werden, bewirkt diese Syn-

chronisierung einen deutlich Performanzaufwand, der wahrscheinlich den Vorteil durch die finalen Methoden zunichte macht. Diese Beobachtung unterstützt die Theorie, der zufolge Programmierer in der Regel keine Ahnung haben, an welchen Stellen Optimierungen notwendig sind. Es ist bedauerlich, daß ein so ungeschicktes Design den Weg in die Standardbibliothek gefunden hat, wo jedermann damit zurecht kommen muß. (Glücklicherweise ersetzt die moderne Containerbibliothek die Klasse `Vector` durch die Klasse `java.util.ArrayList`, die sich sinnvoller verhält. Leider wird nach wie vor neuer Quelltext geschrieben, der Klassen aus der alten Containerbibliothek benutzt.)

[85] Interessanterweise hat `java.util.Hashtable`, eine weitere Klasse aus der Standardbibliothek unter Java 1.0/1.0, keine finalen Methoden. Wie auch andernorts in diesem Buch dokumentiert, wurden offensichtlich einige Klassen von völlig anderen Designern erarbeitet, als andere Klassen. (Beispielsweise sind die Methodenamen in `Hashtable` viel kürzer als diejenigen in `Vector`.) Genau solche Dinge sollten sich dem Konsumenten einer Klassenbibliothek nicht offensichtlich darstellen. Inkonsistente Strukturen bedeuten nur mehr Arbeit für die Clientprogrammierer. Wieder ein Lobgesang auf den Wert von Design- und Quelltextbesprechungen. (Beachten Sie, daß die moderne Containerbibliothek die Klasse `Hashtable` durch die Klasse `java.util.HashMap` ersetzt.)

8.9 Initialisierung und Klassenladen

[86] Bei den traditionellen Programmiersprachen wird ein Programm während des Startvorgangs auf einmal in den Arbeitsspeicher geladen. Danach folgt die Initialisierung und anschließend der Programmstart. Der Initialisierungsvorgang muß bei diesen Sprachen sorgfältig kontrolliert werden, damit die Initialisierungsreihenfolge der statischen Felder keine Schwierigkeiten bereitet. C++ hat beispielsweise Probleme, wenn ein statisches Feld die Gültigkeit eines anderen statischen Feldes erwartet, bevor das letztere Feld initialisiert ist.

[87] Java hat dieses Problem nicht, da beim Laden ein anderer Ansatz verwendet wird. Dies ist ein Beispiel für die Dinge, die dadurch vereinfacht werden, daß bei Java alles ein Objekt ist. Rufen Sie sich in Erinnerung, daß jede Klasse beim Übersetzen eines Quelltextes in einer eigenen Datei ablegt wird. Diese Datei wird nicht geladen, bevor die Klasse benötigt wird. Im allgemeinen können Sie davon ausgehen, daß eine Klasse an der Stelle ihrer ersten Verwendung geladen wird. Dies ist in der Regel die Erzeugung des ersten Objektes der Klasse, kann aber auch beim ersten Zugriff auf ein statisches Feld oder eine statische Methode geschehen.[2]

[88] Die Stelle der ersten Verwendung bezeichnet auch den Zeitpunkt der Initialisierung der statischen Felder. Alle statischen Felder und statischen Initialisierungsblöcke werden im Rahmen des Ladevorgangs in der Reihenfolge ihres Auftretens im Quelltext der Klasse initialisiert beziehungsweise verarbeitet. Die statischen Felder und Initialisierungsblöcke werden natürlich nur ein einziges Mal initialisiert beziehungsweise verarbeitet.

8.9.1 Initialisierungsreihenfolge bei Ableitung

[89] Um eine vollständige Vorstellung der Abläufe zu bekommen, ist es nützlich, den gesamten Initialisierungsvorgang unter Berücksichtigung der Ableitung zu sehen. Betrachten Sie das folgende Beispiel:

```
//: reusing/Beetle.java
// The full process of initialization.
import static net.mindview.util.Print.*;
```

[2] Auch der Konstruktor ist eine statische Methode, obwohl das Schlüsselwort nicht explizit angegeben wird. Um genau zu sein, wird eine Klasse also beim ersten Zugriff auf eine ihrer statischen Komponenten geladen.

```
class Insect {
    private int i = 9;
    protected int j;
    Insect() {
        print("i = " + i + ", j = " + j);
        j = 39;
    }
    private static int x1 = printInit("static Insect.x1 initialized");
    static int printInit(String s) {
        print(s);
        return 47;
    }
}

public class Beetle extends Insect {
    private int k = printInit("Beetle.k initialized");
    public Beetle() {
        print("k = " + k);
        print("j = " + j);
    }
    private static int x2 = printInit("static Beetle.x2 initialized");
    public static void main(String[] args) {
        print("Beetle constructor");
        Beetle b = new Beetle();
    }
} /* Output:
      static Insect.x1 initialized
      static Beetle.x2 initialized
      Beetle constructor
      i = 9, j = 0
      Beetle.k initialized
      k = 47
      j = 39
   *///:~
```

[90] Der erste Schritt nach dem Aufrufen des Programms über `java Beetle` ist der Zugriff auf die `main()`-Methode (eine statische Methode) der Klasse `Beetle`, so daß sich der Klassenlader auf die Suche nach der übersetzten Version der Klasse `Beetle` macht (eine Klassendatei namens *Beetle.class*). Während des Ladevorgangs erkennt der Klassenlader anhand des Schlüsselwortes `extends`, daß `Beetle` eine Basisklasse hat und lädt auch diese. Die Basisklasse wird unabhängig davon geladen, ob Sie ein Objekt dieser Klasse erzeugen. (Vergewissern Sie sich, indem Sie die Objekterzeugung auskommentieren.)

[91] Hätte die Basisklasse wiederum eine Basisklasse, so würde auch diese geladen werden und so weiter. Dann werden die statischen Felder der obersten Basisklasse (hier `Insect`) initialisiert, anschließend die der nächsten unmittelbar abgeleiteten Klasse und so weiter. Dies ist wichtig, weil die Initialisierung der statischen Felder der abgeleiteten Klasse von der korrekten Initialisierung statischer Felder in der Basisklasse abhängen kann.

[92] Nun sind die benötigten Klassen geladen, so daß ein Objekt erzeugt werden kann. Zuerst werden alle Felder primitiven Typs des Objektes mit ihren typspezifischen Initialwerten belegt, Felder nicht primitiven Typs mit `null`. Diese Initialisierung geschieht „auf einen Schlag“, indem der für das Objekt reservierte Arbeitsspeicher auf binär Null gesetzt wird. Danach wird der Konstruktor der Basisklasse aufgerufen. Im obigen Beispiel erfolgt dieser Aufruf automatisch, aber Sie können den Basisklassenkonstruktor auch per `super` als erste Anweisung im Konstruktor der Klasse `Beetle` explizit aufrufen. Nach der Verarbeitung des Basisklassenkonstruktors werden die nicht statischen

Felder in der Reihenfolge ihrer Deklaration initialisiert. Schließlich werden die restlichen Anweisungen im Körper des Konstruktors aufgerufen.

Übungsaufgabe 23: (2) Zeigen Sie, daß das Laden einer Klasse nur einmal stattfindet. Zeigen Sie, daß das Laden entweder durch Erzeugen des ersten Objektes oder den ersten Zugriff auf eine statische Komponente ausgelöst wird. ■

Übungsaufgabe 24: (2) Leiten Sie im Beispiel *Beetle.java* eine Klasse für eine Käfer-Unterart von `Beetle` ab und folgen Sie dabei dem Format der beiden existierenden Klassen. Verfolgen und erläutern Sie die Ausgabe. ■

8.10 Zusammenfassung

[93] Sowohl die Ableitung als auch de Komposition gestatten die Definition einer neuen Klasse unter Bezugnahme auf eine existierende Klasse. Die Komposition wiederverwendet existierende Klassen als Teil der einer neuen Klasse unterliegenden Implementierung. Die Ableitung wiederverwendet die Schnittstelle einer existierenden Klasse.

[94] Die abgeleitete Klasse hat die Schnittstelle ihrer Basisklasse, so daß eine Referenz auf ein Objekt der abgeleiteten Klasse aufwärts in den Typ der Basisklasse umgewandelt werden kann. Dieser Aspekt ist wesentlich für die Polymorphie, wie Sie im nächsten Kapitel lernen werden.

[95] Trotz der allgegenwärtigen Betonung des Ableitungsmechanismus im Kontext der objektorientierten Programmierung, sollten Sie, am Anfang eines Designs im Regelfall die Komposition (gegebenenfalls auch die Delegation) bevorzugen und nur dann auf die Ableitung zurückgreifen, wenn ihre Notwendigkeit offensichtlich ist. Die Komposition ist tendentiell flexibler. Durch Kombination von Komposition und Ableitung bei den Komponententypen, können Sie den exakten Typ und somit das Verhalten der Komponentenobjekte, also auch das Verhalten des komponierten Objektes, zur Laufzeit verändern.

[96] Beim Design eines Systems besteht Ihr Ziel darin, einen Satz von Klassen zu finden beziehungsweise zu entwickeln, so daß jede Klasse eine spezifische Aufgabe erfüllt und weder zu umfangreich noch zu klein ist, das heißt weder zu viel Funktionalität beinhaltet, so daß sie zu unhandlich ist, um wiederverwendet zu werden, noch zu wenig Funktionalität, so daß sie ohne Ergänzungen nicht zu gebrauchen ist. Wird Ihr Design zu komplex, so hilft es oft, die Funktionalität auf mehrere Objekte zu verteilen, indem Sie die vorhandenen Objekte in kleinere Teile zerlegen.

[97] Wenn Sie mit dem Design eines Systems beginnen, kommt es darauf an, zu erkennen, daß die Softwareentwicklung, wie das menschliche Lernen, ein inkrementeller Prozeß ist. Der Fortschritt beruht auf Experimenten. Sie können so viel Untersuchungsaufwand investieren, wie Sie wollen, können aber am Anfang eines Projektes nicht alle Antworten parat haben. Sie werden mehr Erfolg haben und unmittelbare Rückmeldungen bekommen, wenn Sie Ihr Projekt wie einen Organismus oder ein evolutionäres Wesen wachsen lassen, statt es zu konstruieren wie ein komplexes Bauwerk. Ableitung und Komposition sind zwei der fundamentalsten Werkzeuge der objektorientierten Programmierung, die Ihnen solche Experimente ermöglichen.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 9

Polymorphie: Dynamische Bindung

Inhaltsübersicht

„I have been asked, 'Pray, Mr. Babbage, if you put into the machine wrong figures, will the right answers come out?' I am not able to rightly apprehend the kind of confusion of ideas that could provoke such a question.“ Charles Babbage (1791-1871)

Übersetzt etwa: „Ich wurde einmal gefragt, "Bitte, Mr. Babbage, werden die richtigen Antworten herauskommen, wenn Sie der Maschine falsche Zahlen geben?" Ich bin nicht im Stande, die gedankliche Verwirrung recht zu begreifen, die Anlaß zu einer solchen Frage gibt.“

[0] Die Polymorphie ist, nach der Datenabstraktion und der Ableitung, die dritte wesentliche Eigenschaft einer objektorientierten Programmiersprache.

[1] Die Polymorphie öffnet eine weitere Dimension hinsichtlich der Trennung von Schnittstelle und Implementierung, nämlich die Entkopplung des *was* vom *wie*. Sie gestattet verbesserte Organisation und Lesbarkeit des Quelltextes sowie das *Erweitern* von Programmen, nicht nur während

der ursprünglichen Entwicklungsphase des Projektes, sondern auch wenn neue Eigenschaften oder Fähigkeiten gewünscht werden.

[2] Die Kapselung schafft neue Datentypen durch Kombination von charakteristischen Eigenschaften und Verhalten. Das Verbergen der Implementierung trennt die Schnittstelle einer Klasse von deren Implementierung, indem Einzelheiten als privat deklariert werden. Diese Art „mechanischer Organisation“ des Quelltextes leuchtet Programmierern mit prozeduralem Hintergrund ein. Die Polymorphie betrifft dagegen die Entkopplung von Typen. Im vorigen Kapitel haben Sie gesehen, daß Ihnen die Ableitung gestattet, die Referenz auf ein Objekt als dem eigentlichen Objekttyp *oder* dem Typ seiner Basisklasse zugehörig zu behandeln. Diese Fähigkeit ist wesentlich, da sie gestattet, einerseits viele, vom gleichen Basistyp abgeleitete Typen wie einen einzigen Typ zu behandeln und andererseits ein einzelnes Stück Quelltext in gleicher Weise auf jeden dieser Typen anzuwenden. Der polymorphe Methodenaufruf gestattet einem Typ, den Unterschied gegenüber einem anderen ähnlichen Typ auszudrücken, wenn beide von ein und demselben Basistyp abgeleitet sind. Die Verschiedenheit drückt sich durch unterschiedliches Verhalten der Methoden aus, die Sie über eine Referenz vom Typ der Basisklasse aufrufen.

[3] In diesem Kapitel lernen Sie den Effekt der Polymorphie (auch als *dynamische Bindung*, *späte Bindung* oder *Bindung zur Laufzeit* bezeichnet) kennen. Das Kapitel beginnt anhand einfacher Beispiele mit den Grundlagen, wobei alles außer dem polymorphen Verhalten des Programms fortgelassen wird.

9.1 Aufwärtsgerichtete Typumwandlung (Teil 2 von 2)

[4] Sie haben im vorigen Kapitel gesehen, daß Sie ein Objekt, genauer eine Referenz auf ein Objekt, als dem eigentlichen Typ, aber auch als dessen Basistyp zugehörig behandeln können. Die Behandlung einer Objektreferenz als dem Basistyp zugehörig, wird als *aufwärtsgerichtete Typumwandlung* bezeichnet, da Ableitungshierarchien mit der Basisklasse am oberen Ende dargestellt werden.

[5] Sie haben dabei auch ein Problem kommen sehen, welches dem folgenden Beispiele mit Musikinstrumenten ebenfalls innewohnt.

[6] Da mehrere Beispiele in diesem Kapitel Noten „spielen“, lohnt es sich, einen separaten Aufzählungstyp `Note` im Package `polymorphism.music` zu definieren:

```
//: polymorphism/music/Note.java
// Notes to play on musical instruments.
package polymorphism.music;

public enum Note {
    MIDDLE_C, C_SHARP, B_FLAT; // Etc.
} ///:~
```

Aufzählungstypen wurden in Abschnitt 6.9 vorgestellt und in Kapitel 20 ausführlich behandelt.

[7] Blasinstrumente sind Instrumente, so daß die Klasse `Wind` von `Instrument` abgeleitet wird:

```
//: polymorphism/music/Instrument.java
package polymorphism.music;
import static net.mindview.util.Print.*;

class Instrument {
    public void play(Note n) {
        print("Instrument.play()");
    }
}
```

```
///:~

//: polymorphism/music/Wind.java
package polymorphism.music;

// Wind objects are instruments
// because they have the same interface:
public class Wind extends Instrument {
    // Redefine interface method:
    public void play(Note n) {
        System.out.println("Wind.play() " + n);
    }
} ///:~

//: polymorphism/music/Music.java
// Inheritance & upcasting.
package polymorphism.music;

public class Music {
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        tune(flute); // Upcasting
    }
} /* Output:
      Wind.play() MIDDLE_C
   *////:~
```

[8] Die statische `Music`-Methode `tune()` akzeptiert eine Referenz vom Typ `Instrument`, aber auch jedes von `Instrument` abgeleiteten Typs. Die `main()`-Methode gebraucht diese Variabilität, indem Sie der `tune()`-Methode eine Referenz vom Typ `Wind` übergibt. Dabei ist keine Typumwandlung erforderlich. Das ist in Ordnung, weil die Schnittstelle der Klasse `Instrument` aufgrund der Ableitungsbeziehung auch in `Wind` existiert. Die Schnittstelle der Klasse `Wind` kann zwar durch die aufwärtsgerichtete Typumwandlung reduziert werden, aber nicht unter die vollständige Schnittstelle der Basisklasse `Instrument`.

9.1.1 Forgetting The Object Type

[9] Eventuell kommt Ihnen das Beispiel *Music.java* sonderbar vor. Was könnte einen Programmierer dazu bewegen, den Typ einer Referenz absichtlich nicht beachten? Aber genau dies geschieht bei einer aufwärtsgerichteten Typumwandlung. Wäre eine `tune()`-Methode, die eine Referenz vom Typ `Wind` erwartet, nicht eine viel einfachere Lösung? Diese Frage führt zu einem entscheidenden Punkt: In diesem Fall müßten Sie nämlich zu jedem neuen, von `Instrument` abgeleiteten Typ, eine neue Version der `tune()`-Methode anlegen. Angenommen wir folgen diesem Ansatz und definieren zwei neue Klassen `Stringed` und `Brass`:

```
//: polymorphism/music/Music2.java
// Overloading instead of upcasting.
package polymorphism.music;
import static net.mindview.util.Print.*;

class Stringed extends Instrument {
    public void play(Note n) {
```

```
        print("Stringed.play() " + n);
    }
}

class Brass extends Instrument {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
}

public class Music2 {
    public static void tune(Wind i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Stringed i) {
        i.play(Note.MIDDLE_C);
    }
    public static void tune(Brass i) {
        i.play(Note.MIDDLE_C);
    }
    public static void main(String[] args) {
        Wind flute = new Wind();
        Stringed violin = new Stringed();
        Brass frenchHorn = new Brass();
        tune(flute); // No upcasting
        tune(violin);
        tune(frenchHorn);
    }
} /* Output:
      Wind.play() MIDDLE_C
      Stringed.play() MIDDLE_C
      Brass.play() MIDDLE_C
   *///:~
```

[10] Der Ansatz funktioniert zwar, hat aber einen großen Nachteil: Sie müssen zu jeder neuen, von `Instrument` abgeleiteten Klasse eine typspezifische `tune()`-Methode definieren. Das bedeutet mehr Programmierarbeit am Anfang, aber auch, daß Sie beim späteren Hinzufügen einer neuen Methode wie `tune()` oder eines neuen Instrumententyps eine Menge Arbeit vor sich haben. Nehmen Sie noch hinzu, daß der Compiler keine Fehlermeldung ausgibt, wenn Sie eine Version einer Ihrer überladenen Methoden vergessen und die Arbeit mit den abgeleiteten Typen ist nicht mehr zu beherrschen.

[11] Wäre es nicht schöner, wenn Sie nur ein einzige Methode definieren müßten, die ein Argument vom Typ der Basisklasse und keinen der abgeleiteten Typen erwartet? Sprich, wäre es nicht schön, wenn Sie vergessen könnten, daß es abgeleitete Klassen gibt und Ihren Quelltext so schreiben könnten, daß er sich nur auf die Basisklasse konzentriert?

[12] Die Polymorphie gestattet Ihnen genau das zu tun. Die meisten Programmierer mit prozeduralem Hintergrund haben allerdings ein wenig Schwierigkeiten mit der Funktionsweise der Polymorphie.

Übungsaufgabe 1: (2) Schreiben Sie eine Klasse namens `Cycle` und leiten Sie drei Klassen namens `UniCycle`, `BiCycle` und `TriCycle` von `Cycle` ab. Zeigen Sie anhand einer Methode `ride()`, daß eine Referenz auf ein Objekt dieser drei abgeleiteten Klasse aufwärts in den Typ `Cycle` umgewandelt werden kann. ■

9.2 Die überraschende Wendung

[13–14] Das Problem am Beispiel *Music.java* zeigt sich, wenn Sie das Programm aufrufen. Die Ausgabe lautet „`Wind.play()`“. Dies ist zweifellos die gewünschte Ausgabe, aber es leuchtet auf den ersten Blick nicht ein, warum sich der Methodenaufruf `i.play(Note.MIDDLE_C)` im Körper der Methode `tune()` so verhält:

```
public static void tune(Instrument i) {
    // ...
    i.play(Note.MIDDLE_C);
}
```

Die `tune()`-Methode erwartet eine Referenz vom Typ `Instrument`. Wie kann der Compiler „wissen“, daß diese `Instrument`-Referenz in diesem Fall auf ein Objekt vom Typ `Wind` verweist und nicht vom Typ `Brass` oder `Stringed`? Der Compiler kann es nicht „wissen“. Zum tieferen Verständnis dieses Punktes ist es hilfreich, das Thema *dynamische Bindung* zu untersuchen.

9.2.1 Bindung: Verknüpfung von Methodenaufruf und -körper

[15] Der Begriff *Bindung* beschreibt die Verknüpfung eines Methodenaufrufs mit einem Methodenkörper. Findet die Verknüpfung vor dem Programmaufruf statt (per Compiler und Linker, sofern vorhanden), so spricht man von *statischer Bindung* oder *früher Bindung*. Eventuell haben Sie diesen Begriff noch nie zuvor gesehen, da prozedurale Sprachen keine Alternative zur statischen Bindung haben. C-Compiler zum Beispiel, kennen nur die statische Bindung.

[16] Das Geheimnis des Verhaltens des Beispiels *Music.java* betrifft statische Bindung in der Hinsicht, daß der Compiler nicht „wissen“ kann, welche Methode die richtig ist, wenn er nur eine Referenz vom Typ `Instrument` zur Verfügung hat.

[17] Das Geheimnis von *Music.java* ist die *dynamische Bindung*, das heißt die auf dem eigentlichen Typ des referenzierten Objektes basierende *Bindung zur Laufzeit*. Die *dynamische Bindung* wird auch als *späte Bindung* bezeichnet. Eine Sprache mit dynamischer Bindung braucht einen Mechanismus, um den Typ des referenzierten Objektes zur Laufzeit zu bestimmen und die passende Methode aufzurufen. Der Compiler kennt den Objekttyp nach wie vor nicht, aber der Aufrufmechanismus ermittelt zur Laufzeit den richtigen Methodenkörper und veranlaßt den Methodenaufruf. Die Implementierung des dynamischen Bindungsmechanismus' variiert von Sprache zu Sprache, Sie können sich aber vorstellen, daß die Objekte eine Art Typinformationen beinhalten müssen.

[18] Mit Ausnahme statischer oder finaler Methoden basieren alle Methodenaufrufe bei Java auf dynamischer Bindung (beachten Sie, daß private Methoden implizit final sind, siehe Unterunterabschnitt 8.8.2.1). Sie brauchen also in der Regel nicht zu entscheiden, ob dynamische oder statische Bindung eintritt, sondern es geschieht automatisch.

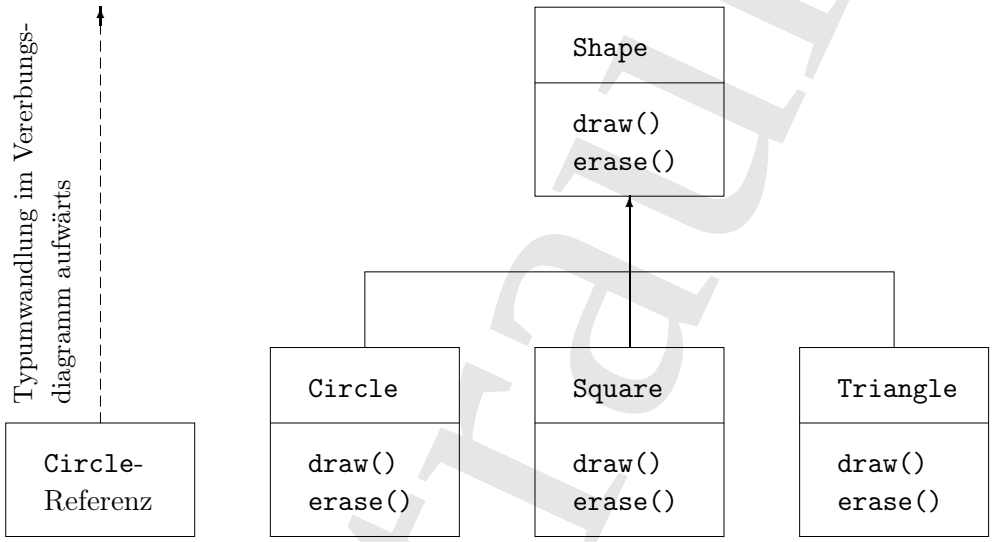
[19] Aus welchem Grund könnten Sie eine Methode aus final deklarieren? Wie Sie in Unterabschnitt 8.8.2 gelernt haben, verhindert die `final`-Deklaration einer Methode, daß diese Methode in abgeleiteten Klassen überschrieben werden kann. Vor dem Hintergrund dieses Kapitels fällt vielleicht noch mehr ins Gewicht: Der `final`-Modifikator „schaltet“ die dynamische Bindung effektiv „ab“ oder genauer, teilt dem Compiler mit, daß keine dynamische Bindung benötigt wird. Der Compiler kann bei finalen Methoden etwas effizienteren Code generieren. In der Mehrzahl der Fälle wirkt sich diese Verbesserung aber nicht auf die insgesamte Performanz Ihres Programms aus, so daß Sie den `final`-Modifikator nur im Rahmen des Designs verwenden sollten, nicht aber, um die Performanz zu steigern.

9.2.2 Producing The Right Behavior

[20] Nachdem Sie nun wissen, daß die Verknüpfung von Methodenaufrufen und -körpern bei Java polymorph, das heißt durch dynamische Bindung vor sich geht, können Sie Ihren Quelltext auf die Basisklasse beziehen und wissen, daß alle Objekte abgeleiteter Klassen korrekt von denselben Anweisungen verarbeitet werden. Sie „senden eine Nachricht an ein Objekt und überlassen es dem Objekt, herauszufinden, welches Verhalten richtig ist.“

[21] Geometrische Figuren sind ein klassisches Beispiel in der objektorientierten Programmierung. Dieses Beispiel wird aufgrund seiner Anschaulichkeit häufig gewählt, kann aber Programmieranfänger zu der Fehlannahme verleiten, die objektorientierte Programmierung betreffe lediglich den Bereich der Graphikprogrammierung, was natürlich nicht wahr ist.

[22] Das Figurenbeispiel besteht aus einer Basisklasse namens `Shape` und verschiedenen hiervon abgeleiteten Klassen: `Circle`, `Square`, `Triangle` und so weiter. Dieses Beispiel funktioniert so gut, weil Sie einfach sagen können „ein Kreis ist eine Figur“ und verstanden werden. Das folgende Ableitungsdiagramm zeigt die Beziehungen:



[23] Die aufwärtsgerichtete Typumwandlung kann mit einer so einfachen Anweisung wie dieser vorgenommen werden:

```
Shape s = new Circle();
```

Die Zeile erzeugt ein `Circle`-Objekt und weist eine Referenz darauf unmittelbar einer Referenzvariablen vom Typ `Shape` zu. Die Zuweisung sieht zunächst wie ein Fehler aus (Typunstimmigkeit), ist aber korrekt, da ein `Circle`-Objekt aufgrund der Ableitungsbeziehung auch ein `Shape`-Objekt ist. Der Compiler ist daher mit der Zuweisung einverstanden und gibt keine Fehlermeldung aus.

[24–25] Angenommen, Sie rufen eine der Basisklassenmethoden auf, die in den abgeleiteten Klassen überschrieben sind:

```
s.draw();
```

Sie erwarten eventuell wiederum, daß die `draw()`-Methode der Klasse `Shape` aufgerufen wird, da `s` schließlich eine Referenzvariable vom Typ `Shape` ist. Woher kann der Compiler „wissen“, daß er etwas anderes tun soll. Dennoch wird, aufgrund der dynamischen Bindung (Polymorphie), die richtige Methode `Circle.draw()` aufrufen.

[26–27] Das folgende Beispiel puts it a slightly different way. Wir legen zuerst eine wiederverwendbare Bibliothek von Shape-Unterklassen an:

```
//: polymorphism/shape/Shape.java
package polymorphism.shape;

public class Shape {
    public void draw() {}
    public void erase() {}
} ///:~

//: polymorphism/shape/Circle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Circle extends Shape {
    public void draw() { print("Circle.draw()"); }
    public void erase() { print("Circle.erase()"); }
} ///:~

//: polymorphism/shape/Square.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Square extends Shape {
    public void draw() { print("Square.draw()"); }
    public void erase() { print("Square.erase()"); }
} ///:~

//: polymorphism/shape/Triangle.java
package polymorphism.shape;
import static net.mindview.util.Print.*;

public class Triangle extends Shape {
    public void draw() { print("Triangle.draw()"); }
    public void erase() { print("Triangle.erase()"); }
} ///:~

//: polymorphism/shape/RandomShapeGenerator.java
// A "factory" that randomly creates shapes.
package polymorphism.shape;
import java.util.*;

public class RandomShapeGenerator {
    private Random rand = new Random(47);
    public Shape next() {
        switch(rand.nextInt(3)) {
        default:
        case 0: return new Circle();
        case 1: return new Square();
        case 2: return new Triangle();
        }
    }
} ///:~

//: polymorphism/Shapes.java
// Polymorphism in Java.
import polymorphism.shape.*;

public class Shapes {
    private static RandomShapeGenerator gen = new RandomShapeGenerator();
    public static void main(String[] args) {
        Shape[] s = new Shape[9];
```

```
        // Fill up the array with shapes:
        for(int i = 0; i < s.length; i++)
            s[i] = gen.next();
        // Make polymorphic method calls:
        for(Shape shp : s)
            shp.draw();
    }
} /* Output:
      Triangle.draw()
      Triangle.draw()
      Square.draw()
      Triangle.draw()
      Square.draw()
      Triangle.draw()
      Square.draw()
      Triangle.draw()
      Circle.draw()
   *///:~
```

Die Basisklasse `Shape` definiert die gemeinsame Schnittstelle aller von `Shape` abgeleiteten Klassen, jede Figur kann also gezeichnet und gelöscht werden. Jede abgeleitete Klasse überschreibt die Methoden `draw()` und `erase()`, um für jede Art von Figur ein eindeutiges Verhalten zu definieren.

[28] Die Klasse `RandomShapeGenerator` ist eine „Fabrikklasse“, die bei jedem Aufruf der `next()`-Methode eine Referenz auf ein Objekt einer zufällig gewählten, von `Shape` abgeleiteten Klasse zurückgibt. Beachten Sie, daß die aufwärtsgerichtete Typumwandlung in den `return`-Anweisungen stattfindet, die jeweils eine Referenz auf ein Objekt vom Typ `Circle`, `Square` beziehungsweise `Triangle` entgegennehmen und als Referenz vom Rückgabetyp `Shape` aus der `next()`-Methode zurückgeben. Sie können einer von `next()` gelieferten Objektreferenz nicht ansehen, welchen Typ das referenzierte Objekt tatsächlich hat, da Sie stets eine schlichte `Shape`-Referenz bekommen.

[29] Die `main()`-Methode beinhaltet ein Array von `Shape`-Referenzen, welches durch Aufrufe der `RandomShapeGenerator`-Methode bewertet wird. Sie wissen an dieser Stelle, daß Sie `Shape`-Referenzen zur Verfügung haben, aber nichts genaueres (und auch der Compiler „weiß“ nicht mehr). Nehmen Sie aber ein Element nach dem anderen aus dem Array und rufen seine `draw()`-Methode auf, so zeigt sich auf magische Art und Weise das korrekte typspezifische Verhalten, wie Sie beim Aufrufen des Programms an der Ausgabe sehen können.

[30] Die Typen der Objekte der von `Shape` abgeleiteten Klassen werden zufällig ausgewählt, um zu verdeutlichen, daß der Compiler über keine zusätzlichen Informationen verfügen kann, welche die statische Bindung zur Übersetzungszeit gestatten. Sämtliche Aufrufe der `draw()`-Methode müssen folglich durch dynamische Bindung geschehen.

Übungsaufgabe 2: (1) Ergänzen Sie das Figurenbeispiel um die Annotation `@Override`. ■

Übungsaufgabe 3: (1) Definieren Sie in der Basisklasse `Shape` eine neue Methode, die eine Meldung ausgibt, aber in der abgeleiteten Klassen nicht überschrieben wird. Erläutern Sie was geschieht. Überschreiben Sie die Methode nun in einer abgeleiteten Klasse, nicht aber in den anderen und beobachten Sie, was geschieht. Überschreiben Sie die Methode nun auch in den übrigen abgeleiteten Klassen. ■

Übungsaufgabe 4: (2) Definieren Sie im Figurenbeispiel *Shapes.java* eine weitere geometrische Figur und verifizieren Sie in der `main()`-Methode, daß die dynamische Bindung auch bei Ihrer neuen Klasse so funktioniert, wie bei den bereits vorhandenen Klassen ■

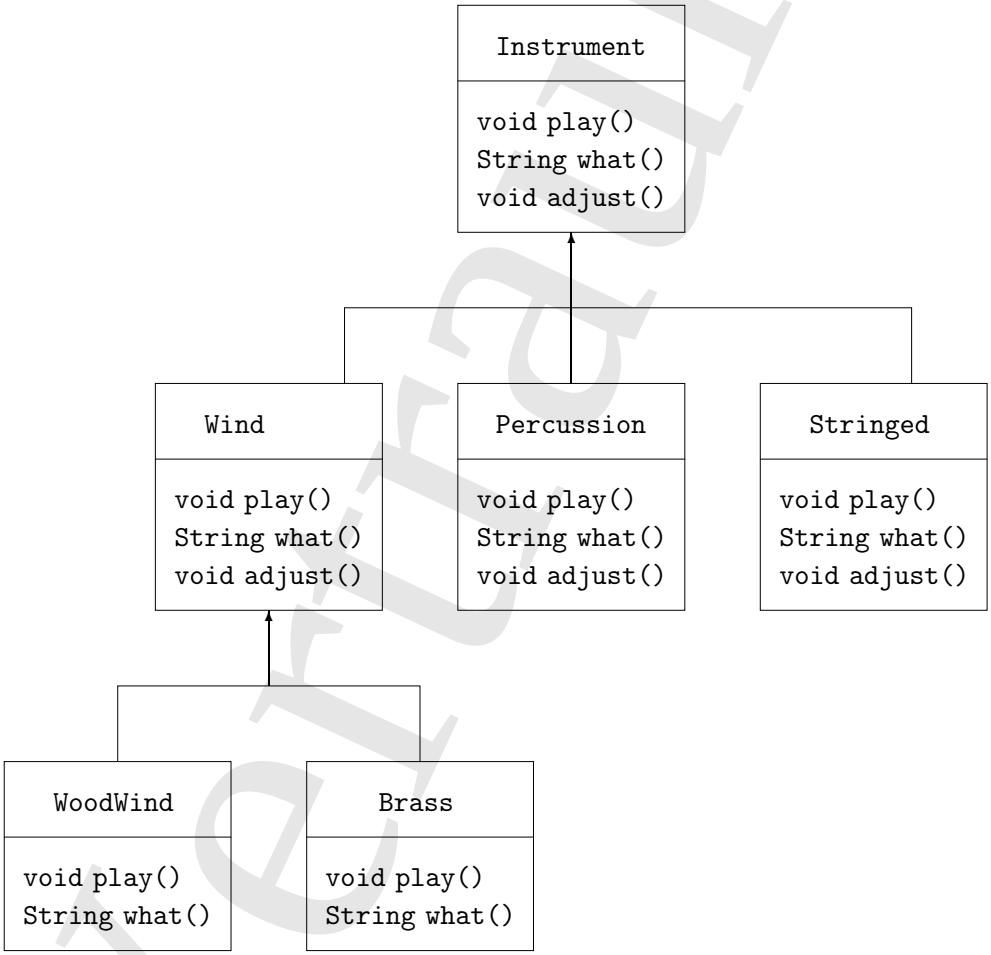
Übungsaufgabe 5: (1) Definieren Sie in der Klasse `Cycle` von Übungsaufgabe 1 (Seite 224) eine

Methode `wheels()`, die die Anzahl der Räder zurückgibt. Ändern Sie die `ride()`-Methode, so daß Sie `wheels()` aufruft und verifizieren Sie, daß die dynamische Bindung funktioniert. ■

9.2.3 Erweiterbarkeit

[31] Kehren wir noch einmal zum Instrumentenbeispiel zurück. Aufgrund der Polymorphie können Sie beliebig viele neue Klassen von `Instrument` ableiten, ohne die `tune()`-Methode ändern zu müssen. Bei einem objektorientierten Programm mit gutem Design folgen die meisten, wenn nicht sogar alle Methoden dem Modell von `tune()` und kommunizieren ausschließlich über die Schnittstelle der Basisklasse mit ihren Argumenten. Ein solches Programm ist in dem Sinne *erweiterbar*, daß Sie neue Klassen von der allgemeinen Basisklasse ableiten können. Die Methoden, welche die durch die Basisklasse gegebene Schnittstelle bedienen, müssen nicht angepaßt werden, um die neuen Klassen ins System aufzunehmen.

[32] Das folgende Diagramm zeigt das Instrumentenbeispiel mit zwei neuen Methoden `what()` und `adjust()` in der Basisklasse und einigen neuen, von `Instrument` abgeleiteten Klassen:



[33–34] Die alte unveränderte `tune()`-Methode harmoniert mit allen Klassen in diesem Ableitungsdiagramm. Selbst wenn die `tune()`-Methode in einer separaten Datei definiert ist und neue Klassen von `Instrument` abgeleitet werden, funktioniert `tune()` nach wie vor, auch ohne erneutes Übersetzen ihrer Datei. Das folgende Beispiel zeigt die Implementierung des obigen Ableitungsdiagramms:

```
//: polymorphism/music3/Music3.java
// An extensible program.
```

```
package polymorphism.music3;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

class Instrument {
    void play(Note n) { print("Instrument.play() " + n); }
    String what() { return "Instrument"; }
    void adjust() { print("Adjusting Instrument"); }
}

class Wind extends Instrument {
    void play(Note n) { print("Wind.play() " + n); }
    String what() { return "Wind"; }
    void adjust() { print("Adjusting Wind"); }
}

class Percussion extends Instrument {
    void play(Note n) { print("Percussion.play() " + n); }
    String what() { return "Percussion"; }
    void adjust() { print("Adjusting Percussion"); }
}

class Stringed extends Instrument {
    void play(Note n) { print("Stringed.play() " + n); }
    String what() { return "Stringed"; }
    void adjust() { print("Adjusting Stringed"); }
}

class Brass extends Wind {
    void play(Note n) { print("Brass.play() " + n); }
    void adjust() { print("Adjusting Brass"); }
}

class Woodwind extends Wind {
    void play(Note n) { print("Woodwind.play() " + n); }
    String what() { return "Woodwind"; }
}

public class Music3 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    public static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    public static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
```

```
        Wind.play() MIDDLE_C
        Percussion.play() MIDDLE_C
        Stringed.play() MIDDLE_C
        Brass.play() MIDDLE_C
        Woodwind.play() MIDDLE_C
    *///:~
```

Die neue `what()`-Methode gibt eine Referenz auf ein `String`-Objekt zurück, das einen Beschreibungstext der jeweiligen Klasse enthält. Die neue `adjust()`-Methode dient dazu, das Instrument zu stimmen.

[35] Jede in das von `orchestra` referenzierte Array in der `main()`-Methode eingesetzte Referenz wird automatisch aufwärts in den Typ `Instrument` umgewandelt.

[36] Sie sehen an der unveränderten Definition der `tune()`-Methode, daß sie hinsichtlich der Änderungen in ihrer Umgebung völlig ahnungslos ist, aber dennoch korrekt funktioniert. Genau diesen Effekt soll die Polymorphie bewirken. Eine Änderung an Ihrem Quelltext zieht keine Programmteile in Mitleidenschaft, die von dieser Änderung nicht betroffen sein sollten. Mit anderen Worten: Die Polymorphie ist ein wichtiges Werkzeug für den Programmierer, um „die veränderlichen Dinge von den unveränderlichen Dingen zu trennen“.

Übungsaufgabe 6: (1) Ändern Sie das Beispiel *Music3.java*, indem Sie die Methode `what()` durch die `Object`-Methode `toString()` ersetzen. Versuchen Sie, die Objekte der von `Instrument` abgeleiteten Klasse per `System.out.println()` (ohne Typumwandlung) auszugeben. ■

Übungsaufgabe 7: (2) Definieren Sie ein neues Instrument im Beispiel *Music3.java* und verifizieren Sie das Funktionieren der dynamischen Bindung für Ihre neue Klasse. ■

Übungsaufgabe 8: (2) Ändern Sie das Beispiel *Music3.java*, so daß es Objekte zufällig gewählten Typs erzeugt. Orientieren Sie sich dabei am Beispiel *Shapes.java* (Seite 227). ■

Übungsaufgabe 9: (3) Legen Sie eine Ableitungshierarchie für Nagetiere unter einer Basisklasse namens `Rodent` an. Leiten Sie Klassen wie `Mouse`, `Gerbil`, `Hamster` und so weiter von `Rodent` ab. Definieren Sie in der Basisklasse einige für Nagetiere typische Methoden und überschreiben Sie sie in den abgeleiteten Klassen, um jeder Unterart ein spezifisches Verhalten zu geben. Legen Sie ein Array für Referenzen vom Typ `Rodent` an und bewerten Sie es mit Objekten der verschiedenen, von `Rodent` abgeleiteten Klassen. Rufen Sie die in der Basisklasse definierten Methoden auf und beobachten Sie was geschieht. ■

Übungsaufgabe 10: (3) Schreiben Sie eine Basiklasse mit zwei Methoden. Rufen Sie in der ersten Methode die zweite Methode auf. Leiten Sie eine Klasse von der Basisklasse ab und überschreiben Sie die zweite Methode. Erzeugen Sie ein Objekt der abgeleiteten Klasse und wandeln Sie die Referenz auf dieses Objekt aufwärts in den Typ der Basisklasse um. Rufen Sie die erste Methode auf und erläutern Sie was geschieht. ■

9.2.4 Vorsicht Falle: Scheinbares Überschreiben privater Methoden

[37–38] Das folgende Beispiel zeigt einen Fehler, der Ihnen aus Arglosigkeit unterlaufen kann:

```
//: polymorphism/PrivateOverride.java
// Trying to override a private method.
package polymorphism;
import static net.mindview.util.Print.*;

public class PrivateOverride {
    private void f() { print("private f()"); }
```

```
    public static void main(String[] args) {
        PrivateOverride po = new Derived();
        po.f();
    }
}

class Derived extends PrivateOverride {
    public void f() { print("public f()"); }
} /* Output:
      private f()
   *///:~
```

Es wäre nicht unvernünftig, die Ausgabe „`public f()`“ zu erwarten, aber eine private Methode ist implizit final und somit vor der abgeleiteten Klasse verborgen. Die Methode `f()` der Klasse `Derived` überschreibt die gleichnamige Methode der Basisklasse somit nicht und definiert auch keine Überladung, da die Methode in der Basisklasse für die abgeleitete Klasse unsichtbar ist.

[39] Die Kernaussage diese Unterabschnitts lautet: Nur nicht private Methoden können überschrieben werden. Achten Sie auf Vorkommen von Methoden, die private Methode zu überschreiben *scheinen*, da der Compiler keine Fehlermeldung ausgibt und eine solche Methode nicht tut, was Sie möglicherweise erwarten. Wählen Sie zum Vermeiden von Mißverständnissen in abgeleiteten Klassen keine Namen privater Basisklassenmethoden.

9.2.5 Vorsicht Falle: Keine Polymorphie bei Feldern und statischen Methoden

[40–41] Wenn Sie beginnen, sich mit Polymorphie zu beschäftigen, ist der Gedanke naheliegend, daß überall Polymorphie im Spiel ist. Allerdings wirkt die dynamische Bindung nur bei gewöhnlichen Methodenaufrufen. Beispielsweise wird ein direkter Zugriff auf ein Feld bereits zur Übersetzungszeit aufgelöst, wie das folgende Beispiel zeigt:[1]

```
//: polymorphism/FieldAccess.java
// Direct field access is determined at compile time.

class Super {
    public int field = 0;
    public int getField() { return field; }
}

class Sub extends Super {
    public int field = 1;
    public int getField() { return field; }
    public int getSuperField() { return super.field; }
}

public class FieldAccess {
    public static void main(String[] args) {
        Super sup = new Sub(); // Upcast
        System.out.println("sup.field = " + sup.field +
                           ", sup.getField() = " + sup.getField());
        Sub sub = new Sub();
        System.out.println("sub.field = " + sub.field
                           + ", sub.getField() = " + sub.getField()
                           + ", sub.getSuperField() = " + sub.getSuperField());
    }
} /* Output:
      sup.field = 0, sup.getField() = 1
```

[1] Danke an Randy Nichols, der diese Frage gestellt hat.

```
        sub.field = 1, sub.getField() = 1, sub.getSuperField() = 0
    *///:~
```

Trotz der aufwärtsgerichteten Typumwandlung der Referenz auf das `Sub`-Objekt in den Basistyp `Super`, wird der Feldzugriff bereits vom Compiler aufgelöst, hat also nichts mit Polymorphie zu tun. In diesem Beispiel werden für die Felder `Super.field` und `Sub.field` verschiedene Bereiche im Arbeitsspeicher allokiert. Ein `Sub`-Objekt hat also eigentlich zwei `field`-Felder, nämlich eines von seiner eigenen Klasse und eines von `Super`. Ein Bezug auf `field` in einem `Sub`-Objekt liefert allerdings *nicht* die `Super`-Version als Standardverhalten. Sie müssen vielmehr explizit `super.field` referenzieren, um das Feld in der Basisklasse zu erreichen.

[42] Diese Situation wirkt zwar unübersichtlich, kommt aber in Praxis so gut wie nicht vor. Einerseits deklarieren Sie in der Regel alle Felder als privat, so daß kein direkter Zugriff möglich ist, sondern nur als Seiteneffekt eines Methodenaufrufs. Andererseits sollten Sie Felder in der Basisklasse und abgeleiteten Klassen nicht mit dem gleichen Namen bezeichnen, da offensichtlich Verwirrung entsteht.

[43] Eine statische Methode verhält sich nicht polymorph:

```
//: polymorphism/StaticPolymorphism.java
// Static methods are not polymorphic.

class StaticSuper {
    public static String staticGet() {
        return ''Base staticGet()'';
    }
    public String dynamicGet() {
        return ''Base dynamicGet()'';
    }
}

class StaticSub extends StaticSuper {
    public static String staticGet() {
        return ''Derived staticGet()'';
    }
    public String dynamicGet() {
        return ''Derived dynamicGet()'';
    }
}

public class StaticPolymorphism {
    public static void main(String[] args) {
        StaticSuper sup = new StaticSub(); // Upcast
        System.out.println(sup.staticGet());
        System.out.println(sup.dynamicGet());
    }
} /* Output:
      Base staticGet()
      Derived dynamicGet()
   *///:~
```

Statische Methoden gehören zur Klasse, nicht aber zu den individuellen Objekten.

9.3 Konstruktoren und Polymorphie

[44] Konstruktoren unterscheiden sich wie gewöhnlich von den übrigen Methoden. Dies gilt auch im Hinblick auf die Polymorphie. Auch wenn Konstruktoren keine polymorphen, das heißt dynamisch

gebundenen Methoden sind (es sind eigentlich implizit statische Methoden), ist es wichtig, ihre Funktionsweise in komplexen Hierarchien mit Vererbung zu verstehen. Dieses Verständnis wird Ihnen dabei helfen, leidige Unordnung zu vermeiden.

9.3.1 Reihenfolge der Konstruktoraufrufe

[45] Die Reihenfolge der Konstruktoraufrufe wurde in den Unterabschnitten 6.7.1 und 8.9.1 kurz diskutiert. Zu diesem Zeitpunkt war aber die Polymorphie noch nicht eingeführt.

[46] Beim Erzeugen eines Objektes einer abgeleiteten Klasse wird stets ein Konstruktor der Basisklasse aufgerufen, wobei sich die Verkettung der Konstruktoraufrufe bis zur Wurzelklasse `Object` hin fortsetzt. Dies ist im Hinblick auf die besondere Aufgabe des Konstruktors sinnvoll, nämlich für die korrekte Erzeugung eines Objektes zu sorgen. Eine abgeleitete Klasse hat in der Regel nur Zugriff auf ihre eigenen Felder, nicht aber auf die Felder der Basisklasse, die typischerweise als privat deklariert sind. Alleine der Basisklassenkonstruktor verfügt über die notwendigen Informationen und Berechtigungen, um die Felder eines Objektes seiner Klasse zu initialisieren. Es ist daher wichtig, daß alle Konstruktoren aufgerufen werden, da das Objekt andernfalls nicht vollständig erzeugt werden würde. Aus diesem Grund erzwingt der Compiler beim Erzeugen eines Objektes einer abgeleiteten Klasse einen Konstruktoraufruf für das Unterobjekt der Basisklasse. Falls Sie im Konstruktor einer abgeleiteten Klasse nicht explizit einen Konstruktor der Basisklasse aufrufen, wählt der Compiler stillschweigend den Standardkonstruktor der Basisklasse. Findet der Compiler keinen Standardkonstruktor, so gibt er eine Fehlermeldung aus. Definiert eine Klasse keinen Konstruktor, so erzeugt der Compiler automatisch einen Standardkonstruktor.

[47] Das folgende Beispiel zeigt die Auswirkungen von Komposition, Ableitung und Polymorphie auf den Ablauf der Objekterzeugung:

```
//: polymorphism/Sandwich.java
// Order of constructor calls.
package polymorphism;
import static net.mindview.util.Print.*;

class Meal {
    Meal() { print("Meal()"); }
}

class Bread {
    Bread() { print("Bread()"); }
}

class Cheese {
    Cheese() { print("Cheese()"); }
}

class Lettuce {
    Lettuce() { print("Lettuce()"); }
}

class Lunch extends Meal {
    Lunch() { print("Lunch()"); }
}

class PortableLunch extends Lunch {
    PortableLunch() { print("PortableLunch()");}
}

public class Sandwich extends PortableLunch {
    private Bread b = new Bread();
```

```
    private Cheese c = new Cheese();
    private Lettuce l = new Lettuce();
    public Sandwich() { print("Sandwich()"); }
    public static void main(String[] args) {
        new Sandwich();
    }
} /* Output:
      Meal()
      Lunch()
      PortableLunch()
      Bread()
      Cheese()
      Lettuce()
      Sandwich()
   *///:~
```

[48] Dieses Beispiel definiert eine komplexe Klasse aus anderen Klassen, wobei jede Klasse einen Konstruktor hat, der bei Aufruf eine Textmeldung ausgibt. Die Hauptklasse `Sandwich` steht in der dritten Ebene einer Ableitungshierarchie (sogar vier Ebenen, wenn Sie die implizite Ableitung von der Wurzelklasse `Object` einbeziehen) und referenziert drei Komponentenobjekte. Die Ausgabe beim Erzeugen eines `Sandwich`-Objektes in der `main()`-Methode dokumentiert die folgende Reihenfolge von Konstruktoraufrufen:

- Der Konstruktor der Basisklasse `PortableLunch` wird aufgerufen. Dieser Schritt wird rekursiv bis zur Wurzelklasse der Ableitungshierarchie (`Object`) wiederholt. Dadurch wird das `Object`-Unterobjekt zuerst erzeugt, danach das Unterobjekt der nächsten direkt abgeleiteten Klasse (`Meal`) und so weiter, bis zum *Erreichen des Konstruktors* der letzten abgeleiteten Klasse der Hierarchie (`Sandwich`).

- Die nicht statischen Felder des `Sandwich`-Objektes werden in der Reihenfolge ihrer Deklaration initialisiert.

- Die Anweisungen im Körper des Konstruktors der Klasse `Sandwich` werden verarbeitet.

[49] Die Reihenfolge der Konstruktoraufrufe ist wesentlich: Beim Ableiten einer Klasse wissen Sie „alles“ über die Basisklasse und haben Zugriff auf deren als `public` oder `protected` deklarierte Komponenten. Sie müssen sich also in einer abgeleiteten Klasse darauf verlassen können, daß alle Felder der Basisklasse initialisiert sind. Beim Aufrufen einer gewöhnlichen Methode hat die Objekterzeugung bereits stattgefunden, so daß alle Felder in allen Teilen des Objektes bewertet sind. Im Konstruktor müssen Sie dagegen voraussetzen können, daß alle referenzierten Felder initialisiert sind. Die einzige Möglichkeit, dies zu garantieren, besteht darin, den Konstruktor der Basisklasse zuerst aufzurufen. Anschließend sind alle Komponenten der Basisklasse initialisiert, die Sie vom Konstruktor der Basisklasse aus erreichen können. Die Gewißheit, daß alle Felder eines Objektes beim Eintritt in den Körper des Konstruktors initialisiert sind, ist auch der Grund dafür, daß Sie alle Felder für Komponentenobjekte (das heißt per Komposition in Klasse eingesetzte Objekte) an der Stelle ihrer Deklaration initialisieren sollten (beispielsweise die Felder `b`, `c` und `l` im obigen Beispiel). Wenn Sie diese Regeln befolgen, unterstützen Sie die Garantie, daß alle Felder im Unterobjekt der Basisklasse und alle Komponentenobjekte des erzeugten Objektes initialisiert werden. Leider deckt diese Vorgehensweise aber nicht jeden Fall ab, wie Sie im nächsten Abschnitt oder Unterabschnitt? sehen werden.

Übungsaufgabe 11: (1) Ergänzen Sie das Beispiel *Sandwich.java* um eine Klasse `Pickle`. ■

9.3.2 Ableitung und Aufräumen

[50] Wenn Sie per Komposition und Ableitung neue Klassen definieren, brauchen Sie sich in der Regel keine Sorgen über das Aufräumen zu machen, da die Unterobjekte in der Regel der automatischen Speicherbereinigung überlassen werden können. Sind Aufräumarbeiten erforderlich, so müssen Sie gewissenhaft sein und für Ihre neue Klasse eine `dispose()`-Methode definieren. (Sie können den Methodennamen frei wählen. Ich habe mich beim folgenden Beispiel für `dispose()` entschieden.) Wenn Sie die `dispose()`-Methode in einer abgeleiteten Klasse überschreiben, ist es wichtig, daran zu denken, die Basisklassenversion aufzurufen, weil die dortigen Anweisungen sonst nicht verarbeitet werden. Das folgende Beispiel zeigt ~~this/was?~~:

```
//: polymorphism/Frog.java
// Cleanup and inheritance.
package polymorphism;
import static net.mindview.util.Print.*;

class Characteristic {
    private String s;
    Characteristic(String s) {
        this.s = s;
        print("Creating Characteristic " + s);
    }
    protected void dispose() {
        print("disposing Characteristic " + s);
    }
}

class Description {
    private String s;
    Description(String s) {
        this.s = s;
        print("Creating Description " + s);
    }
    protected void dispose() {
        print("disposing Description " + s);
    }
}

class LivingCreature {
    private Characteristic p = new Characteristic("is alive");
    private Description t = new Description("Basic Living Creature");
    LivingCreature() {
        print("LivingCreature()");
    }
    protected void dispose() {
        print("LivingCreature dispose");
        t.dispose();
        p.dispose();
    }
}

class Animal extends LivingCreature {
    private Characteristic p = new Characteristic("has heart");
    private Description t = new Description("Animal not Vegetable");
    Animal() { print("Animal()"); }
    protected void dispose() {
        print("Animal dispose");
        t.dispose();
```

```
        p.dispose();
        super.dispose();
    }
}

class Amphibian extends Animal {
    private Characteristic p = new Characteristic(''can live in water'');
    private Description t = new Description(''Both water and land'');
    Amphibian() {
        print(''Amphibian()'');
    }
    protected void dispose() {
        print(''Amphibian dispose'');
        t.dispose();
        p.dispose();
        super.dispose();
    }
}

public class Frog extends Amphibian {
    private Characteristic p = new Characteristic(''Croaks'');
    private Description t = new Description(''Eats Bugs'');
    public Frog() { print(''Frog()''); }
    protected void dispose() {
        print(''Frog dispose'');
        t.dispose();
        p.dispose();
        super.dispose();
    }
    public static void main(String[] args) {
        Frog frog = new Frog();
        print(''Bye!'');
        frog.dispose();
    }
} /* Output:
      Creating Characteristic is alive
      Creating Description Basic Living Creature
      LivingCreature()
      Creating Characteristic has heart
      Creating Description Animal not Vegetable
      Animal()
      Creating Characteristic can live in water
      Creating Description Both water and land
      Amphibian()
      Creating Characteristic Croaks
      Creating Description Eats Bugs
      Frog()
      Bye!
      Frog dispose
      disposing Description Eats Bugs
      disposing Characteristic Croaks
      Amphibian dispose
      disposing Description Both water and land
      disposing Characteristic can live in water
      Animal dispose
      disposing Description Animal not Vegetable
      disposing Characteristic has heart
      LivingCreature dispose
```

```
        disposing Description Basic Living Creature
        disposing Characteristic is alive
    *///:~
```

[51–52] Jede Klasse der Hierarchie beinhaltet je ein Komponentenobjekt der beiden Klassen `Characteristic` und `Description`, die ebenfalls aufgeräumt werden müssen. Für den Fall, daß die Unterobjekte voneinander abhängig sind, sollte beim Aufräumen die bezüglich der Objekterzeugung umgekehrte Reihenfolge gewählt werden. Bei Feldern ist dies die der Deklaration entgegengesetzte Reihenfolge (da Felder in der Anordnung ihrer Deklaration initialisiert werden). Bei Klassen in einer Ableitungshierarchie sollte, der Vorgehensweise bei Destruktoren in C++ entsprechend, das Objekt der abgeleiteten Klasse vor dem Unterobjekt der Basisklasse aufgeräumt werden, weil das Aufräumen des Objektes der abgeleiteten Klasse das Aufrufen von Methoden aus der Schnittstelle der Basisklasse erfordern kann, so daß das Unterobjekt noch „lebendig“ sein muß und nicht zu früh zerstört werden darf. Die Ausgabe zeigt, daß alle Komponenten des `Frog`-Objektes in zur Objekterzeugung entgegengesetzter Reihenfolge aufgeräumt werden. Dieses Beispiel zeigt, daß das Aufräumen Sorgfalt und Bewußtsein erfordert.

Übungsaufgabe 12: (3) Ändern Sie Übungsaufgabe 9 (Seite 231), so daß das Programm die Initialisierungsreihenfolge anzeigt. Legen Sie in der Basisklasse und in den abgeleiteten Klassen Komponentenobjekte an und zeigen Sie in welcher Reihenfolge sie bei der Objekterzeugung initialisiert werden. ■

[53] Beachten Sie im obigen Beispiel, daß ein `Frog`-Objekt der alleinige Besitzer seiner beiden Komponentenobjekte ist. Das `Frog`-Objekt erzeugt seine Komponentenobjekte selbst und „weiß“ wie lange sie benötigt werden (solange das `Frog`-Objekt existiert), also auch wann die Komponentenobjekte aufgeräumt werden müssen. Die Situation wird komplizierter, wenn ein Objekt eines seiner Komponentenobjekte mit anderen Objekten teilt, Sie also nicht einfach die Aufräummethode aufrufen können. Unter diesen Umständen kann *Referenzzählung* notwendig sein, um die Anzahl der Objekte zu überwachen, die das gemeinsame Objekt noch referenzieren:

```
//: polymorphism/ReferenceCounting.java
// Cleaning up shared member objects.
import static net.mindview.util.Print.*;

class Shared {
    private int refcount = 0;
    private static long counter = 0;
    private final long id = counter++;
    public Shared() {
        print("Creating " + this);
    }
    public void addRef() { refcount++; }
    protected void dispose() {
        if(--refcount == 0)
            print("Disposing " + this);
    }
    public String toString() { return "Shared " + id; }
}

class Composing {
    private Shared shared;
    private static long counter = 0;
    private final long id = counter++;
    public Composing(Shared shared) {
        print("Creating " + this);
        this.shared = shared;
        this.shared.addRef();
```

```
    }
    protected void dispose() {
        print("disposing " + this);
        shared.dispose();
    }
    public String toString() { return "Composing " + id; }
}

public class ReferenceCounting {
    public static void main(String[] args) {
        Shared shared = new Shared();
        Composing[] composing = { new Composing(shared),
                                  new Composing(shared),
                                  new Composing(shared),
                                  new Composing(shared),
                                  new Composing(shared) };
        for(Composing c : composing)
            c.dispose();
    }
} /* Output:
      Creating Shared 0
      Creating Composing 0
      Creating Composing 1
      Creating Composing 2
      Creating Composing 3
      Creating Composing 4
      disposing Composing 0
      disposing Composing 1
      disposing Composing 2
      disposing Composing 3
      disposing Composing 4
      Disposing Shared 0
   *///:~
```

[54] Das statische `long`-Feld `counter` der Klasse `Shared` zählt die erzeugten `Shared`-Objekte und liefert auch den Wert des `id`-Feldes. Das `counter`-Feld hat den Typ `long` statt `int`, um einen Überlauf zu verhindern. (Dies ist nur eine gute Angewohnheit. Es ist unwahrscheinlich, daß bei den Beispielen in diesem Buch ein solcher Zähler überläuft.) Das `id`-Feld ist final, da während des Lebenszyklus' des Objektes keine Änderung zu erwarten ist.

[55] Sie müssen daran denken, die `addRef()`-Methode aufzurufen, wenn Sie ein `Shared`-Objekt mit einem Objekt der Klasse `Composing` verknüpfen, während die `dispose()`-Methode den Referenzzähler selbst überwacht und entscheidet, wann die Aufräumarbeiten ausgeführt werden. Die Anwendung dieses Ansatz erfordert zusätzliche Gewissenhaftigkeit, aber wenn mehrere Objekte ein gemeinsames Objekt verwenden, bei dem Aufräumarbeiten erforderlich sind, gibt es nur wenige Alternativen.

Übungsaufgabe 13: (3) Definieren Sie im Beispiel *ReferenceCounting.java* eine `finalize()`-Methode, um die Terminierungsvoraussetzung zu verifizieren (siehe Unterabschnitt 6.5.3). ■

Übungsaufgabe 14: (4) Ändern Sie Übungsaufgabe 12 (Seite 238), so daß eines der Komponentenobjekte ein mit anderen Objekten gemeinsam verwendetes Objekt mit Referenzzählung ist. Zeigen Sie, daß das Aufräumen funktioniert. ■

9.3.3 Vorsicht Falle: Dynamisch gebundene Methoden im Konstruktor einer Basisklasse

[56] Die Hierarchie der Konstruktoraufrufe führt zu einem interessanten Problem: Was geschieht nämlich, wenn Sie im Körper eines Konstruktors eine dynamische gebundene Methode des gegenwärtig erzeugten Objektes aufrufen?

[57] Im Körper einer gewöhnlichen Methode wird die dynamische Bindung zur Laufzeit vorgenommen, da der Compiler nicht „wissen“ kann, ob die Referenz auf ein Objekt der Klasse verweist, in der die Methode erstmals definiert ist, oder eine davon abgeleiteten Klasse.

[58] Wenn Sie eine dynamisch gebundene Methode in einem Konstruktor aufrufen, wird die überschriebene Version dieser Methode gewählt. Ein solcher Methodenaufruf kann allerdings unerwartete Auswirkungen haben, da die überschriebene Methode aufgerufen wird, bevor das zugehörige Objekt vollständig erzeugt wurde.

[59] Der Konstruktor hat vom Konzept her die Aufgabe, das Objekt „ins Leben zu rufen“ (keine gewöhnliche Leistung). Im Konstruktor ist das Objekt eventuell erst zum Teil „fertig“. Sie wissen lediglich, daß das Unterobjekt der Basisklasse bereits initialisiert ist. Ist der Konstruktor nur ein Schritt beim Erzeugen eines Objektes einer Klasse, die von der Klasse abgeleitet ist, zu welcher der Konstruktor gehört, so ist der Anteil der abgeleiteten Klasse zum Zeitpunkt des Konstruktoraufrufs noch uninitialisiert. Ein dynamisch gebundener Methodenaufruf reicht über den Einflußbereich einer Klasse hinaus und die Ableitungshierarchie hinunter, das heißt es wird eine Methode in einer abgeleiteten Klasse aufgerufen. Enthält ein Konstruktir eine solche Anweisung, so rufen Sie eine Methode auf, die unter Umständen den Inhalt eines noch nicht initialisierten Feldes abfragt oder ändert: Ein sicherer Weg ins Unheil.

[60–61] Das folgende Beispiel zeigt das Problem:

```
//: polymorphism/PolyConstructors.java
// Constructors and polymorphism
// don't produce what you might expect.
import static net.mindview.util.Print.*;

class Glyph {
    void draw() { print("Glyph.draw()"); }
    Glyph() {
        print("Glyph() before draw()");
        draw();
        print("Glyph() after draw()");
    }
}

class RoundGlyph extends Glyph {
    private int radius = 1;
    RoundGlyph(int r) {
        radius = r;
        print("RoundGlyph.RoundGlyph(), radius = " + radius);
    }
    void draw() {
        print("RoundGlyph.draw(), radius = " + radius);
    }
}

public class PolyConstructors {
    public static void main(String[] args) {
        new RoundGlyph(5);
    }
```

```
} /* Output:
      Glyph() before draw()
      RoundGlyph.draw(), radius = 0
      Glyph() after draw()
      RoundGlyph.RoundGlyph(), radius = 5
   *///:~
```

Die `draw()`-Methode der Basisklasse `Glyph` ist zum Überschreiben vorgesehen und wird in der abgeleiteten Klasse `RoundGlyph` überschrieben. Allerdings ruft der Konstruktor von `Glyph` die `draw()`-Methode auf und dieser Aufruf führt, scheinbar beabsichtigt, schließlich zu der Version in `RoundGlyph`. Die Ausgabe zeigt, daß das `radius`-Feld beim Aufruf der `draw()`-Methode im Konstruktor von `Glyph` noch nicht einmal mit dem programmatisch definierten Standardwert 1 initialisiert ist, sondern mit dem typspezifischen Initialwert 0. Auf dem Bildschirm würde voraussichtlich entweder ein Punkt oder überhaupt nicht gezeichnet werden und Sie würden sich wundern und versuchen, herauszufinden, warum das Programm nicht funktioniert.

[62] Die in Abschnitt 9.3.1 beschriebene Initialisierungsreihenfolge ist noch unvollständig und hier liegt auch die Lösung des Rätsels. Die Initialisierungsreihenfolge lautet in Wirklichkeit:

- Der für das Objekt allokierte Arbeitsspeicher wird auf binär Null gesetzt, bevor etwas anderes geschieht.
- Die Basisklassenkonstruktoren werden aufgerufen, wie zuvor beschrieben. An diesem Punkt wird die überschriebene `draw()`-Methode aufgerufen (in der Tat: vor dem Aufruf des Konstruktors der Klasse `RoundGlyph`), wodurch sich der Initialwert 0 des `radius`-Feldes zeigt, entsprechend dem vorigen Schritt.
- Die nicht statischen Felder werden in der Reihenfolge ihrer Deklaration initialisiert.
- Die Anweisungen im Konstruktor der abgeleiteten Klasse werden verarbeitet.

[63] Es gibt allerdings eine Obergrenze in der Hinsicht, daß jedes Feld wenigstens mit seinem typspezifischen Initialwert versehen wird und nicht einfach uninitialisiert bleibt. Das schließt die Felder zur Aufnahme von Referenzen auf Komponentenobjekte ein, die mit `null` bewertet werden. Übersehen Sie die Initialisierung eines solchen Feldes, so wird zur Laufzeit eine Ausnahme ausgeworfen. Alle übrigen Felder bekommen den Anfangswert Null (ein beim Durchsehen der Ausgabe verräterischer Wert).

[64] Andererseits sollten Sie von der Ausgabe dieses Programms aber auch entsetzt sein. Sie haben eine logische Sache getan und dennoch verhält sich das Programm auf rätselhafte Weise falsch, ohne daß der Compiler eine Fehlermeldung ausgibt. (C++ verhält sich in dieser Situation vernünftiger.) Derartige Fehler können leicht tief im Quelltext verborgen sein und sie zu entdecken kostet viel Zeit.

[65] Daher lautet eine gute Richtlinie für Konstruktoren: „Tun Sie so wenig wie möglich, um das Objekt in einen vernünftigen Zustand zu versetzen und rufen Sie, wenn möglich, keine anderen Methoden der Klasse auf, zu der der Konstruktor gehört.“ Die einzigen im Konstruktor sicher aufrufbaren Methoden sind die finalen Methoden der Basisklasse. (Gilt auch für private Methoden, die implizit final sind.) Finale Methoden können nicht überschrieben werden, so daß Überraschungen wie im Beispiel *PolyConstructors.java* ausgeschlossen sind. Diese Richtlinie läßt sich nicht immer einhalten, ist aber erstrebenswert.

Übungsaufgabe 15: (2) Erweitern Sie das Beispiel *PolyConstructors.java* um eine Klasse `RectangularGlyph` und führen Sie das in diesem Unterabschnitt beschriebene Problem vor. ■

9.4 Kovariante Rückgabetypen

[66–67] Seit Version 5 der Java Standard Edition (SE 5) gibt es *kovariante Rückgabetypen*, das heißt, daß eine überschreibende Methode in einer abgeleiteten Klasse einen Typ zurückgeben darf, der vom Rückgabetyp der überschriebenen Methode in der Basisklasse abgeleitet ist:

```
//: polymorphism/CovariantReturn.java
class Grain {
    public String toString() { return ''Grain''; }
}

class Wheat extends Grain {
    public String toString() { return ''Wheat''; }
}

class Mill {
    Grain process() { return new Grain(); }
}

class WheatMill extends Mill {
    Wheat process() { return new Wheat(); }
}

public class CovariantReturn {
    public static void main(String[] args) {
        Mill m = new Mill();
        Grain g = m.process();
        System.out.println(g);
        m = new WheatMill();
        g = m.process();
        System.out.println(g);
    }
} /* Output:
      Grain
      Wheat
   *///:~
```

Der Unterschied zwischen der SE 5 und den früheren Java-Versionen besteht darin, daß die früheren Java-Versionen die überschriebene `process()`-Methode zwangen, den Typ `Grain` statt `Wheat` zurückzugeben, obwohl `Wheat` von `Grain` abgeleitet und somit eigentlich ein zulässiger Rückgabetyp ist. Durch kovariante Rückgabetypen die Wahl des spezifischeren Typs `Wheat` möglich.

9.5 Design mit Ableitung

[68] Wenn Sie beginnen, sich mit Polymorphie auseinanderzusetzen, scheint es, als ob die Ableitung so oft wie möglich verwendet werden sollte, da die dynamische Bindung ein so raffinierter Mechanismus ist. Zuviel Ableitung kann Ihr Design aber belasten. Wenn Sie sich beim Anlegen einer neuen Klasse stets zuerst für die Ableitung einer existierenden Klasse entscheiden, können die Dinge unnötig kompliziert werden.

[69] Der Kompositionsansatz ist für den Anfang besser geeignet, vor allem, wenn nicht offensichtlich ist, welchen Ansatz Sie wählen sollten. Die Komposition führt nicht zwangsläufig zu einem Design mit Ableitungshierarchie. Der Kompositionsansatz ist auch flexibler, da Sie den Typ eines Objektes (also Verhalten) dynamisch wählen können, während bei Ableitung ein exakter Typ zur Übersetzungszeit bekannt sein muß:

```
//: polymorphism/Transmogrify.java
// Dynamically changing the behavior of an object
// via composition (the ''State'' design pattern).
import static net.mindview.util.Print.*;

class Actor {
    public void act() {}
}

class HappyActor extends Actor {
    public void act() { print(''HappyActor''); }
}

class SadActor extends Actor {
    public void act() { print(''SadActor''); }
}

class Stage {
    private Actor actor = new HappyActor();
    public void change() { actor = new SadActor(); }
    public void performPlay() { actor.act(); }
}

public class Transmogrify {
    public static void main(String[] args) {
        Stage stage = new Stage();
        stage.performPlay();
        stage.change();
        stage.performPlay();
    }
} /* Output:
      HappyActor
      SadActor
   *///:~
```

[70] Ein `Stage`-Objekt enthält ein Feld vom Typ `Actor`, das mit einer Referenz auf ein `HappyActor`-Objekt initialisiert ist. Die Methode `performPlay()` liefert somit ein bestimmtes Verhalten. Da eine Referenzvariable aber zur Laufzeit auf ein anderes Objekt gerichtet werden kann, können Sie dem `actor`-Feld eine Referenz auf ein `SadActor`-Objekt zuweisen, wodurch sich das Verhalten der `performPlay()`-Methode ändert. Sie gewinnen also Flexibilität zur Laufzeit. Für diesen Ansatz ist auch die Bezeichnung „*State*-Entwurfsmuster“ gebräuchlich. Siehe hierzu *Thinking in Patterns, Problem-Solving Techniques using Java*. Im Gegensatz dazu können Sie eine Ableitungsbeziehung zur Laufzeit nicht auf eine andere Basisklasse ausrichten, da alle Bezüge zur Übersetzungszeit bestimmt sein müssen.

[71] Eine allgemeine Richtlinie lautet: „Nutzen Sie Ableitung, um Verhaltensunterschiede und Felder, um Zustandunterschiede auszudrücken.“ Im obigen Beispiel treten beide Fälle auf: Zwei abgeleitete Klassen beschreiben in ihren `act()`-Methoden verschiedenes Verhalten und die Klasse `Stage` verwendet Komposition, um Zustandsänderungen ihrer Objekte zu erlauben. In diesem Fall bewirkt eine Zustandänderung eine Verhaltensänderung.

Übungsaufgabe 16: (3) Legen Sie eine Klasse namens `Starship` an, die ein Feld vom Typ `AlertStatus` enthält, das drei verschiedene Zustände anzeigen kann. Legen Sie Methoden an, um die Zustände zu ändern. Orientieren Sie sich am Beispiel *Transmogrify.java*. ■

9.5.1 Vergleich zwischen Substitution und Erweiterung

[72] Der „reine“ Ansatz scheint die beste Vorgehensweise zum Anlegen einer Ableitungshierarchie zu sein. Sprich, die abgeleiteten Klassen definieren und überschreiben nur die in der Basisklasse definierten Methoden. Abbildung 9.1 könnte als eine reine „ist ein“-Beziehung bezeichnet werden, da die Schnittstelle einer Klasse establishes what is is. Die Ableitung garantiert, daß jede abgeleitete Klasse die Schnittstelle der Basisklasse hat und nichts weniger. Die abgeleiteten Klassen in diesem Diagramm haben allerdings auch nicht mehr, als die Schnittstelle der Basisklasse.

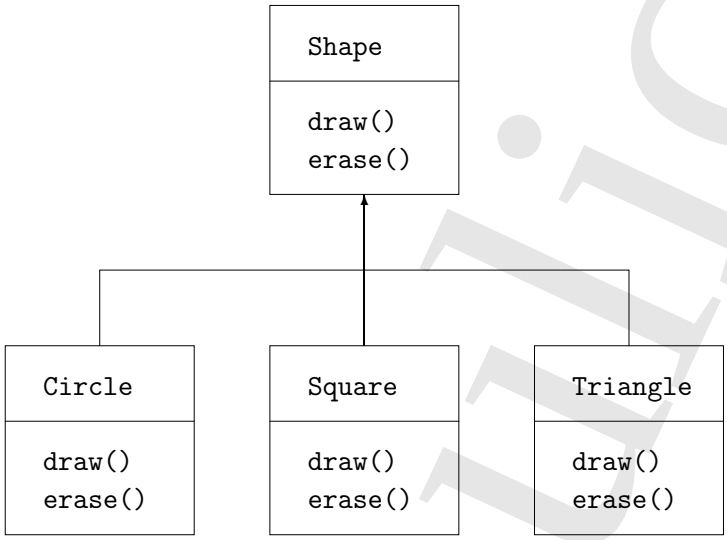


Abbildung 9.1: Reine „ist ein“-Beziehung. Die abgeleiteten Klassen überschreiben nur die in der Basisklasse definierten Methoden.

[74] Diese Konstellation kann als *reine Substitution* betrachtet werden, da die Objekte der abgeleiteten Klassen perfekt anstelle der Basisklasse eingesetzt werden können und Sie niemals etwas über die abgeleiteten Klassen wissen müssen, um sie anwenden zu können (siehe Abbildung 9.2).

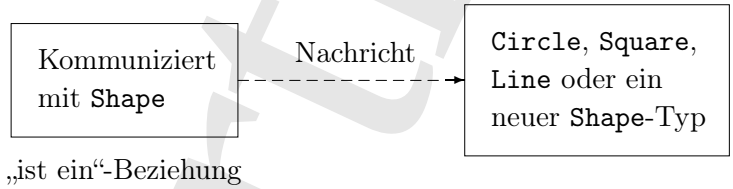


Abbildung 9.2: Reine Substitution: Die Objekte der abgeleiteten Klassen können perfekt anstelle der Basisklasse eingesetzt werden.

[75] Die Basisklasse kann also dieselben Nachrichten empfangen, als die abgeleiteten Klassen, da stets dieselbe Schnittstelle vorliegt. Sie müssen nicht mehr tun, als eine Referenz auf ein Objekt einer abgeleiteten Klasse aufwärts in den Typ der Basisklasse umzuwandeln und sich mit mehr um den eigentlichen Typ des Objektes zu sorgen, mit dem Sie arbeiten. Die dynamische Bindung kümmert sich um alles.

[76] So gesehen, scheint eine reine „ist ein“-Beziehung der einzige vernünftige Weg zu sein, während jedes andere Design auf verhaspelte Überlegungen hindeutet und automatisch fehlerhaft ist. Aber auch dies ist eine Falle. Sobald Sie sich an diese Denkweise gewöhnen, „schalten Sie um“ und entdecken, daß das Erweitern der Schnittstelle (wozu das Schlüsselwort `extends` unglücklicherweise zu ermutigen scheint) die perfekte Lösung für Ihr aktuelles Problem ist. Sie könnten von einer „ähnelt

einem“-Beziehung sprechen, da die abgeleiteten Klassen der Basisklasse durch dieselbe fundamentale Schnittstelle ähnlich sind, aber weitere Eigenschaften und Fähigkeiten implementieren, durch die zusätzliche Methoden erforderlich sind (siehe Abbildung 9.3).

[77] Obwohl auch dies ein nützlicher und sinnvoller Ansatz ist (je nach Situation), hat er einen Nachteil. Der erweiterte Teil der Schnittstelle in der abgeleiteten Klasse ist über eine Referenz vom Typ der Basisklasse nicht erreichbar, können also nach einer aufwärtsgerichteten Typumwandlung nicht mehr aufgerufen werden (siehe Abbildung 9.4).

[78] Wenn Sie die Objektreferenz nicht aufwärts umwandeln, brauchen Sie sich nicht um diese Einschränkung zu kümmern. Es gibt aber nicht selten Situationen, in denen Sie den eigentlichen Typ eines Objektes ermitteln müssen, um die zusätzlichen Methoden dieses Typs aufrufen zu können. Der folgende und letzte Unterabschnitt in diesem Kapitel zeigt wie eine solche *abwärts*gerichtete Typumwandlung ausgeführt wird.

9.5.2 Abwärtsgerichtete Typumwandlung und Typidentifikation zur Laufzeit

[79] Da die Information über den eigentlichen Objekttyp während einer *aufwärtsgerichteten Typumwandlung* („Bewegung“ nach oben in der Ableitungshierarchie, hin zur Basisklasse) verloren geht, liegt nahe, daß Sie die Typinformation durch eine *abwärtsgerichtete Typumwandlung* („Bewegung“ nach unten in der Ableitungshierarchie) wieder zurückbekommen können. Eine Typumwandlung in Richtung der Basisklasse ist stets sicher, da die Basisklasse keine größere Schnittstelle haben kann, als eine ihrer abgeleiteten Klassen. Aus diesem Grund wird jede, durch die Schnittstelle der Basisklasse gesendete Nachricht garantiert akzeptiert. Bei einer abwärtsgerichteten Typumwandlung können Sie dagegen nicht wissen, ob eine Referenz vom Typ `Shape` tatsächlich auf ein `Circle`-Objekt verweist. Der eigentliche Objekttyp kann ebensogut `Triangle`, `Square` oder noch ein anderer Typ sein.

[80] Die Lösung dieses Problems setzt eine Möglichkeit voraus, um zu garantieren, daß eine abwärtsgerichtete Typumwandlung zulässig ist, damit Sie nicht versehentlich einen falschen Zieltyp

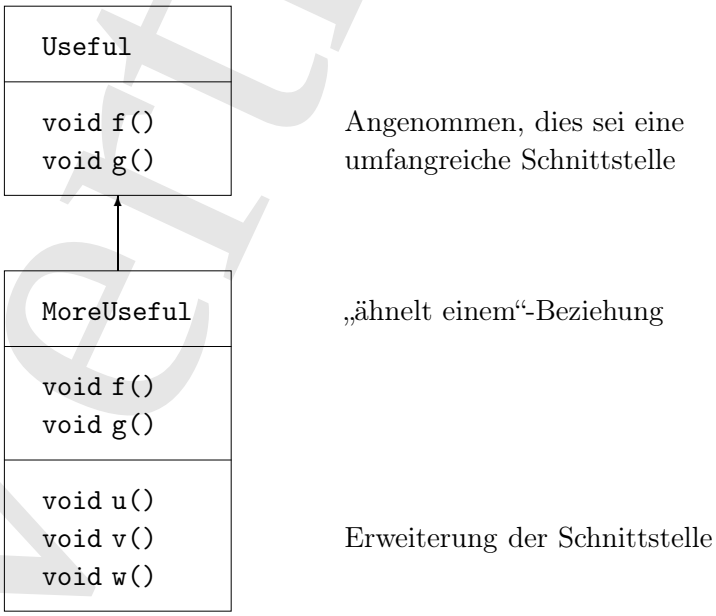


Abbildung 9.3: „ähnelt einem“-Beziehung: Die abgeleiteten Klasse ähneln der Basisklasse hinsichtlich ihrer Schnittstellen, verfügen aber über zusätzliche Funktionalität, implementieren also zusätzliche Methoden.

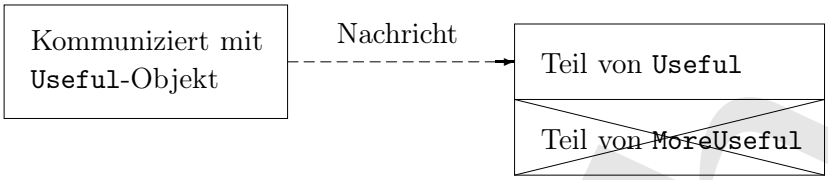


Abbildung 9.4: Der erweiterte Teil der Schnittstelle in der abgeleiteten Klasse ist über eine Referenz vom Typ der Basisklasse nicht erreichbar, können also nach einer aufwärtsgerichteten Typumwandlung nicht mehr aufgerufen werden.

wählen und dem Objekt eine Nachricht senden, die es nicht akzeptieren kann. Andernfalls wäre die Typumwandlung ziemlich unsicher.

[81] Bei manchen Sprachen, darunter C++, ist eine spezielle Operation erforderlich, um eine typsichere abwärtsgerichtete Umwandlung zu vollziehen, während bei Java *jede* Typumwandlung geprüft wird! Auch wenn die explizite Notation nur wie die gewöhnliche eingeklammerte Typumwandlungssyntax aussieht, wird diese Umwandlung zur Laufzeit geprüft, um sicherzustellen, daß der angegebene Zieltyp dem eigentlichen Objekttyp auch wirklich entspricht. Andernfalls wird eine Ausnahme vom Typ `java.lang.ClassCastException` ausgeworfen. Dieser Typprüfungsvorgang zur Laufzeit heißt *Runtime Type Identification (RTTI)*, übersetzt etwa „Typidentifikation zur Laufzeit“. Das folgende Beispiel demonstriert die Funktionsweise der RTTI:

```
//: polymorphism/RTTI.java
// Downcasting & Runtime type information (RTTI).
// {ThrowsException}

class Useful {
    public void f() {}
    public void g() {}
}

class MoreUseful extends Useful {
    public void f() {}
    public void g() {}
    public void u() {}
    public void v() {}
    public void w() {}
}

public class RTTI {
    public static void main(String[] args) {
        Useful[] x = {
            new Useful(),
            new MoreUseful()
        };
        x[0].f();
        x[1].g();
        // Compile time: method not found in Useful:
        //! x[1].u();
        ((MoreUseful)x[1]).u(); // Downcast/RTTI
        ((MoreUseful)x[0]).u(); // Exception thrown
    }
} ///:~
```

[82] Die Klasse `MoreUseful` erweitert die Schnittstelle der Klasse `Useful`, wie in Abbildung 9.3. Nachdem `MoreUseful` von `Useful` abgeleitet ist, kann eine Referenz auf ein `MoreUseful`-Objekt aufwärts in den Typ `Useful` umgewandelt werden. Da beide Referenzen im Array vom Typ `Useful`

sind, können Sie bei beiden Objekten die Methoden `f()` und `g()` aufrufen, während der Versuch, die Methode `u()` aufzurufen (die nur in der Klasse `MoreUseful` definiert ist), eine Fehlermeldung zur Übersetzungszeit hervorruft.

[83] Wenn Sie auf die erweiterte Schnittstelle eines `MoreUseful`-Objektes zugreifen wollen, können Sie eine abwärtsgerichtete Typumwandlung versuchen. Stimmen Umwandlungszieltyp und Objekttyp überein, so wird die Typumwandlung vollzogen. Andernfalls wird eine Ausnahme vom Typ `ClassCastException` ausgeworfen. Die Ausnahme muß nicht behandelt werden, da sie einen Programmierfehler anzeigt, der überall in einem Programm auftreten kann und umgehend korrigiert werden muß. Der Kommentar `{ThrowsException}` teilt dem Ant-Skript zur Quelltextdistribution dieses Buches mit, daß beim Ausführen dieses Programms eine Ausnahme zu erwarten ist.

[84] Der RTTI-Mechanismus kann mehr, als nur einfache Typumwandlungen ausführen. Es gibt beispielsweise eine Möglichkeit, den Zieltyp *vor* der Typumwandlung zu testen. Kapitel 15 widmet sich auschließlich dem Studium verschiedener Aspekte der Typidentifikation zur Laufzeit bei Java.

Übungsaufgabe 17: (2) Legen Sie in den abgeleiteten Klasse `UniCycle` und `BiCycle` der `Cycle`-Hierarchie von Übungsaufgabe 1 (Seite 224) eine `balance()`-Methode an, nicht aber in `TriCycle`. Erzeugen Sie von jeder abgeleiteten Klasse ein Objekt und speichern Sie die Referenzen in einem Array vom Typ `Cycle`. Versuchen Sie, auf jedem Element des Arrays die `balance()`-Methode aufzurufen und beobachten Sie das Ergebnis. Führen Sie abwärtsgerichtete Typumwandlungen ein, rufen Sie die `balance()`-Methode nochmals auf und beobachten Sie das geschieht. ■

9.6 Zusammenfassung

[85] Polymorphie bedeutet „Vielgestaltigkeit“. In der objektorientierten Programmierung definiert eine Basisklasse eine Schnittstelle aus Methoden und es gibt verschiedene abgeleitete Klassen („Gestalten“), die diese Schnittstelle mittels unterschiedlicher Versionen der dynamischen gebundener Methoden.

[86] In diesem Kapitel haben Sie gesehen, daß Sie kein Beispiel für Polymorphie ohne Datenabstraktion und Ableitung verstehen, geschweige denn schreiben können. Die Polymorphie ist eine Eigenschaft, die nicht isoliert betrachtet werden kann (wie beispielsweise eine `switch`-Anweisung), sondern nur als Teil eines größeren Kontextes aus Beziehungen zwischen Klassen funktioniert.

[87] Die Anwendung der Polymorphie und damit auch anderer objektorientierter Ansätze in Ihren Programmen setzt voraus, daß Sie Ihren Programmierhorizont erweitern und nicht nur Felder und Nachrichtigen an Objekte einer individuellen Klasse betrachten, sondern auch die Gemeinsamkeiten zwischen Klassen sowie ihren gegenseitigen Beziehungen. Dies erfordert zwar viel Mühe, ist aber eine ehrbare Anstrengung. Die Ergebnisse sind schnellere Entwicklung von Programmen, bessere Organisation des Quelltextes, erweiterbare Programme und leichtere Pflege des Quelltextes.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 10

Interfaces und abstrakte Klassen

Inhaltsübersicht

[0] Interfaces und abstrakte Klassen gestatten die strukturelle Trennung von Schnittstelle und Implementierung.

[1] Derartige Mechanismen sind in den Programmiersprachen nicht verbreitet. C++ unterstützt Interfaces und abstrakte Klasse beispielsweise nur indirekt. Die Tatsache, daß Java über eigene Schlüsselwörter verfügt, zeigt, daß diese Konzepte als wichtig genug betrachtet wurden, um direkte Unterstützung zu bieten.

[2] Wir besprechen zuerst das Konzept der abstrakten Klasse, einer Art Zwischenschritt zwischen gewöhnlicher Klasse und Interface. Auch wenn Sie zuerst den Drang verspüren, ein Interface zu definieren, ist die abstrakte Klasse ein wichtiges und notwendiges Hilfsmittel, um Klassen zu definieren, die nicht implementierte Methoden enthalten. Sie können nicht immer ein reines Interface wählen.

10.1 Abstrakte Klassen und Methoden

[3] Die Methoden in der Basisklasse `Instrument` der Instrumentenbeispiele im vorigen Kapitel waren stets nur Platzhalter. Sollte jemals eine dieser Methoden aufgerufen werden, dann haben Sie etwas falsch gemacht. Die Klasse `Instrument` wurde mit der Absicht angelegt, eine gemeinsame Schnittstelle für die von ihr abgeleiteten Klassen zu definieren.

[4] Diese gemeinsame Schnittstelle wurde bei diesen Beispielen aus nur einem Grund eingeführt, nämlich um in jeder abgeleiteten Klasse ein anderes Verhalten ausdrücken zu können. Die Schnittstelle definiert eine grundlegende Erscheinungsform, die allen abgeleiteten Klassen gemeinsam ist. Sie könnten `Instrument` also auch als *abstrakte Basisklasse* oder einfach als *abstrakte Klasse* bezeichnen.

[5] Die Objekte einer „abstrakten Klasse“[1] wie `Instrument` haben in der Regel keine eigene Funktionalität. Sie legen eine abstrakte Klasse an, um abgeleitete Klassen über die gemeinsame Schnittstelle zu bedienen. `Instrument` ist also nur dazu gedacht, um die Schnittstelle auszudrücken, nicht aber als eigene Implementierung. Es ist daher unsinnig, ein `Instrument`-Objekt zu erzeugen und Sie möchten den Clientprogrammierer eventuell daran hindern. Sie könnten beispielsweise alle Methoden in der Klasse `Instrument` veranlassen, eine Fehlermeldung auszugeben, wodurch die Information aber erst zur Laufzeit verfügbar wird. Außerdem müßten Sie sich darauf verlassen, daß der Clientprogrammierer sein Programm erschöpfend testet. Es ist in der Regel besser, Probleme bereits zur Übersetzungszeit abzufangen.

[6] Java verfügt zu diesem Zweck über *abstrakte Methoden*.[2] Eine abstrakte Methode ist unvollständig, das heißt sie besteht nur aus einer Deklaration und hat keinen Methodenkörper. Die Syntax zur Deklaration einer abstrakten Methode lautet:

```
abstract void f();
```

Eine Klasse wird als *abstrakte Klasse* bezeichnet, wenn sie wenigstens eine abstrakte Methode beinhaltet und muß in diesem Fall selbst als abstrakt deklariert werden. (Der Compiler gibt andernfalls eine Fehlermeldung aus.)

[7] Wie soll sich der Compiler verhalten, wenn jemand versucht, ein Objekt einer abstrakten (also unvollständigen) Klasse zu erzeugen? Der Compiler reagiert mit einer Fehlermeldung, da er nicht in der Lage ist, ein Objekt einer abstrakten Klasse zu erzeugen und garantiert auf diese Weise die Reinheit der abstrakten Klasse, so daß Sie sich keine Gedanken über Mißbrauch machen müssen.

[8] Wenn Sie eine Klasse von einer abstrakten Klasse ableiten und Objekte des neuen Typs erzeugen wollen, müssen Sie zu jeder abstrakten Methode in der Basisklasse eine Methodendefinition angeben. Andernfalls ist die abgeleitete Klasse wiederum abstrakt und der Compiler erzwingt die Kennzeichnung dieser Klasse mit dem Schlüsselwort `abstract`.

[9] Es ist zulässig, eine Klasse als abstrakt zu deklarieren, obwohl sie keine abstrakten Methoden enthält. Diese Option ist nützlich, wenn Sie in einer Klasse keine abstrakten Methoden anlegen, zugleich aber das Erzeugen von Objekten dieser Klasse unterbinden wollen.

[10] Die Klasse `Instrument` aus dem vorigen Kapitel läßt sich leicht in eine abstrakte Klasse ändern. Nur ein Teil der Methoden ist abstrakt, da Sie beim Anlegen einer abstrakten Klasse nicht gezwungen sind, alle Methoden als abstrakt zu deklarieren. Abbildung 10.1 zeigt die Klassenhierarchie.

[11] Hier ist das geänderte Instrumentenbeispiel mit abstrakten Klassen und Methoden:

```
//: interfaces/music4/Music4.java
// Abstract classes and methods.
package interfaces.music4;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;
```

[1] Anmerkung des Übersetzers: Die Klasse `Instrument` aus Kapitel 9 ist keine abstrakte Klasse im eigentlichen Sinn. Von einer abstrakten Klasse können keine Objekte erzeugt werden. Zur Verdeutlichung wurde der Begriff „abstrakten Klasse“ daher in Anführungszeichen gesetzt.

[2] Für C++-Programmierer: Die abstrakten Methoden von Java sind das Äquivalent rein virtueller Funktionen.

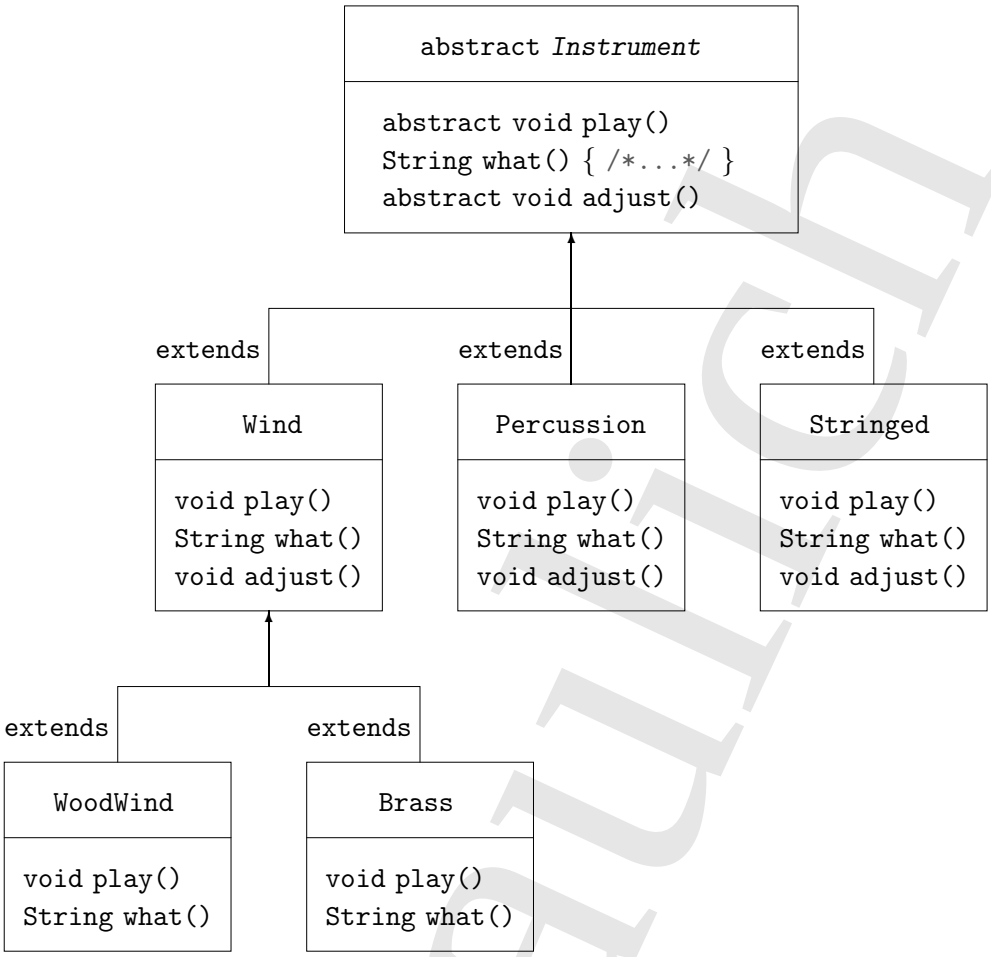


Abbildung 10.1: Änderung der Klasse `Instrument` aus Kapitel 9 in eine abstrakte Klasse mit Hierarchie aus einigen abgeleiten Klassen (siehe auch Beispiel *Music4.java*, Seite 250f).

```
abstract class Instrument {
    private int i; // Storage allocated for each
    public abstract void play(Note n);
    public String what() { return ''Instrument''; }
    public abstract void adjust();
}

class Wind extends Instrument {
    public void play(Note n) {
        print(''Wind.play() '' + n);
    }
    public String what() { return ''Wind''; }
    public void adjust() {}
}

class Percussion extends Instrument {
    public void play(Note n) {
        print(''Percussion.play() '' + n);
    }
    public String what() { return ''Percussion''; }
    public void adjust() {}
}

class Stringed extends Instrument {
```

```
    public void play(Note n) {
        print("Stringed.play() " + n);
    }
    public String what() { return "Stringed"; }
    public void adjust() {}
}

class Brass extends Wind {
    public void play(Note n) {
        print("Brass.play() " + n);
    }
    public void adjust() { print("Brass.adjust()"); }
}

class Woodwind extends Wind {
    public void play(Note n) {
        print("Woodwind.play() " + n);
    }
    public String what() { return "Woodwind"; }
}

public class Music4 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
      Wind.play() MIDDLE_C
      Percussion.play() MIDDLE_C
      Stringed.play() MIDDLE_C
      Brass.play() MIDDLE_C
      Woodwind.play() MIDDLE_C
   *///:~
```

Abgesehen von der Basisklasse ist der Quelltext unverändert.

[12] Das Anlegen abstrakter Klassen und Methoden ist sinnvoll, weil dadurch die Abstraktheit explizit erkennbar wird und den Clientprogrammierern Ankunft darüber gibt, für welchen Verwendungszweck sie vorgesehen sind. Abstrakte Klassen sind außerdem ein nützliches Hilfsmittel zur Refaktorierung, da Sie gemeinsame Methoden einfach in der Ableitungshierarchie aufwärts verschieben können.

Übungsaufgabe 1: (1) Ändern Sie Übungsaufgabe 9 aus Unterabschnitt 9.2.3 (Seite 231), so daß `Rodent` eine abstrakte Klasse wird. Deklarieren Sie soviele Methoden der Klasse `Rodent` als abstrakt, wie möglich. ■

Übungsaufgabe 2: (1) Schreiben Sie eine abstrakte Klasse ohne abstrakte Methoden und verifizieren Sie, daß Sie kein Objekt dieser Klasse erzeugen können. ■

Übungsaufgabe 3: (2) Schreiben Sie eine Basisklasse mit einer abstrakten `print()`-Methode, die in einer abgeleiteten Klasse überschrieben wird. Die überschriebene Version der Methode gibt den Standardwert eines in der abgeleiteten Klasse definierten `int`-Feldes aus. Initialisieren Sie das Feld bei seiner Definition mit einem von Null verschiedenen Wert. Rufen Sie die `print()`-Methode im Konstruktor der Basisklasse auf. Erzeugen Sie in der `main()`-Methode ein Objekt der abgeleiteten Klasse und rufen Sie seine `print()`-Methode auf. Erklären Sie die Ergebnisse. ■

Übungsaufgabe 4: (3) Schreiben Sie eine abstrakte Klasse ohne Methoden. Leiten Sie eine Klasse ab und definieren Sie dort eine Methode `f()`. Legen Sie eine statische Methode an, welche eine Objektreferenz vom Typ der Basisklasse erwartet, abwärts in den Typ der abgeleiteten Klasse umwandelt und die Methode `f()` aufruft. Zeigen Sie in der `main()`-Methode, daß der Aufruf möglich ist. Deklarieren Sie nun `f()` in der Basisklasse als statische Methode, so daß die Typumwandlung nicht mehr benötigt wird. ■

10.2 Interfaces

[13] Das Schlüsselwort `abstract` ermöglicht eine Klasse mit einer oder mehreren undefinierten Methoden, Sie definieren also einen Teil der Schnittstelle ohne Implementierung. Die Implementierung wird in den abgeleiteten Klassen definiert. Das Schlüsselwort `interface` geht einen Schritt weiter in Richtung Abstraktheit und gestattet die Definition einer „vollständig abstrakten Klasse“, die überhaupt keine Implementierung enthält. Ein Interface definiert Methodennamen, Argumentlisten und Rückgabetypen, aber keine Methodenkörper. Anders ausgedrückt: Ein Interface liefert eine Form, aber keine Implementierung.

[14] Ein Interface sagt aus, daß jede Klasse, die dieses Interface implementiert, die angegebene Schnittstelle besitzt. Jede Anweisung, die sich auf ein bestimmtes Interface bezieht, „weiß“, welche Methoden aufgerufen werden können. Das ist alles. Ein Interface legt gewissermaßen ein „Protokoll“ unter den Klassen fest. (Einige objektorientierte Programmiersprachen verwenden dafür das Schlüsselwort `protocol`.)

[15] Interfaces sind allerdings mehr als nur extrem abstrakte Klassen. Da eine Klasse mehr als ein Interface implementieren kann, gestatten Interfaces eine Art „Mehrfachvererbung“.

[16] Die Definition eines Interface wird mit dem Schlüsselwort `interface` anstelle von `class` eingeleitet. Wie bei Klassendefinitionen, kann auch dem Schlüsselwort `interface` der Modifikator `public` vorangestellt werden (allerdings nur, wenn das Interface in einer Datei mit demselben Namen definiert ist). Ohne `public` steht das Interface unter Packagezugriff, kann also nur innerhalb desselben Packages verwendet werden. Ein Interface kann auch Felder beinhalten, die dann implizit als statisch und final deklariert sind.

[17] Das Schlüsselwort `implements` bezieht eine Klassendefinition auf eines oder mehrere Interfaces. Während das Interface die Erscheinungsform beschreibt, definiert die Klasse, wie das Verhalten tatsächlich implementiert ist. Davon abgesehen, fügt sich ein Interface unauffällig in die Ableitungshierarchie ein. Abbildung 10.2 zeigt das geänderte Instrumentenbeispiel, nun mit dem Interface *`Instrument`*. `WoodWind` und `Brass` zeigen, daß die Implementierung eines Interfaces eine gewöhnli-

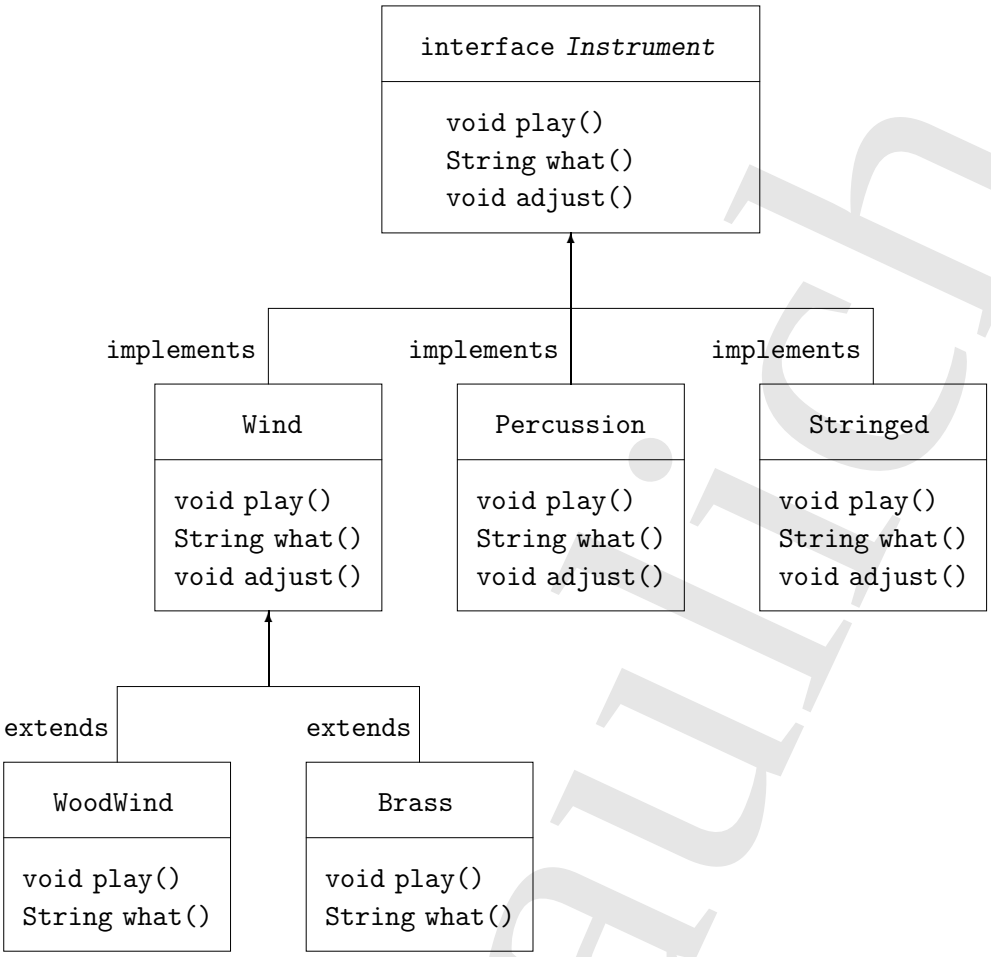


Abbildung 10.2: Änderung der Klasse `Instrument` aus Kapitel 9 in ein Interface mit Hierarchie aus einigen implementieren beziehungsweise abgeleiten Klassen (siehe auch Beispiel *Music5.java*, Seite 254f).

che Klasse ist, von der wie gewohnt weitere Klassen abgeleitet werden können.

[18] Sie können die in einem Interface deklarierten Methoden explizit als öffentlich deklarieren, obwohl sie diese Eigenschaft automatisch erhalten. Beim Implementieren eines Interfaces *müssen* die ausprogrammierten Methoden daher als öffentliche Methoden deklariert werden. Andernfalls stünden die implementierten Methoden unter Packagezugriff, das heißt der Zugriff auf diese Methoden würde unter der Ableitung eingeschränkt. Das ist nicht erlaubt und bewirkt eine Fehlermeldung des Compilers.

[19–20] Sie können dies am geänderten Instrumentenbeispiel nachvollziehen. Beachten Sie, daß jede Methode im Interface nur eine Deklaration ist, der einzige vom Compiler gestattet Modus. Außerdem ist keine der Methoden in *`Instrument`* als öffentlich deklariert, die Deklaration erfolgt aber automatisch:

```
//: interfaces/music5/Music5.java
// Interfaces.
package interfaces.music5;
import polymorphism.music.Note;
import static net.mindview.util.Print.*;

interface Instrument {
    // Compile-time constant:
    int VALUE = 5; // static & final
```

```
    // Cannot have method definitions:
    void play(Note n); // Automatically public
    void adjust();
}

class Wind implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Wind"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Percussion implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Percussion"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Stringed implements Instrument {
    public void play(Note n) {
        print(this + ".play() " + n);
    }
    public String toString() { return "Stringed"; }
    public void adjust() { print(this + ".adjust()"); }
}

class Brass extends Wind {
    public String toString() { return "Brass"; }
}

class Woodwind extends Wind {
    public String toString() { return "Woodwind"; }
}

public class Music5 {
    // Doesn't care about type, so new types
    // added to the system still work right:
    static void tune(Instrument i) {
        // ...
        i.play(Note.MIDDLE_C);
    }
    static void tuneAll(Instrument[] e) {
        for(Instrument i : e)
            tune(i);
    }
    public static void main(String[] args) {
        // Upcasting during addition to the array:
        Instrument[] orchestra = {
            new Wind(),
            new Percussion(),
            new Stringed(),
            new Brass(),
            new Woodwind()
        };
        tuneAll(orchestra);
    }
} /* Output:
```

```
    Wind.play() MIDDLE_C
    Percussion.play() MIDDLE_C
    Stringed.play() MIDDLE_C
    Brass.play() MIDDLE_C
    Woodwind.play() MIDDLE_C
  *///:~
```

Diese Version des Instrumentenbeispiels enthält noch eine weitere Änderung: Die `what()`-Methode wurde durch `toString()` ersetzt, da die `what()`-Methode zu diesem Zweck angelegt wurde. Da die `toString()`-Methode in der Wurzelklasse `Object` definiert ist, braucht sie nicht im Interface *`Instrument`* deklariert zu werden.

[21] Das restliche Programm funktioniert wie zuvor. Beachten Sie, daß die aufwärts gerichtete Typumwandlung stets dasselbe Verhalten liefert, unabhängig davon, ob sich die Typumwandlung auf eine „reguläre“ Klasse, eine abstrakte Klasse oder ein Interface bezieht. An der `tune()`-Methode können Sie sehen, daß es tatsächlich keinerlei Hinweis dafür gibt, ob der Typ `Instrument` eine „reguläre“ Klasse, eine abstrakte Klasse oder ein Interface ist.

Übungsaufgabe 5: (2) Definieren Sie ein Interface mit drei Methoden in einem eigenen Package. Implementieren Sie das Interface in einem *anderen* Package. ■

Übungsaufgabe 6: (2) Zeigen Sie, daß alle in einem Interface deklarierten Methoden automatisch öffentliche Methoden sind. ■

Übungsaufgabe 7: (1) Ändern Sie Übungsaufgabe 9 aus Unterabschnitt 9.2.3 (Seite 231), so daß `Rodent` ein Interface wird. ■

Übungsaufgabe 8: (2) Definieren Sie im Beispiel *polymorphism.Sandwich.java* aus Unterabschnitt 9.3.1 (Seite 234) ein Interface *`FastFood`* mit passenden Methoden und ändern Sie die Klasse `Sandwich`, so daß sie *`FastFood`* implementiert. ■

Übungsaufgabe 9: (3) Refaktorieren Sie das Beispiel *Music5.java*, indem Sie alle gemeinsamen Methoden der Klassen `Wind`, `Percussion` und `Stringed` in eine abstrakte Klasse verschieben. ■

Übungsaufgabe 10: (3) Ändern Sie das Beispiel *Music5.java*, indem Sie ein Interface *`Playable`* definieren. Verschieben Sie die Deklaration der `play()`-Methode nach *`Playable`*. Ergänzen Sie *`Playable`* in den `implements`-Listen der abgeleiteten Klassen. Ändern Sie die `tune()`-Methode, so daß sie ein Argument vom Typ *`Playable`* anstelle von *`Instrument`* erwartet. ■

10.3 Vollständige Entkopplung

[22] Bezieht sich ein Parameter einer Methode auf eine Klasse anstelle eines Interfaces, so sind Sie an diese Klasse und ihre Unterklassen gebunden. Wenn Sie die Methode auf eine Klasse außerhalb dieser Hierarchie anwenden wollen, haben Sie Pech gehabt. Ein Interface lockert diese Einschränkung beträchtlich und gestattet Ihnen, die Wiederverwendbarkeit Ihrer Methoden zu verbessern.

[23–24] Stellen Sie sich zum Beispiel eine Klasse `Processor` mit einer `name()`- und einer `process()`-Methode vor, die eine Eingabe erwartet, verarbeitet und eine Ausgabe liefert. Die Basisklasse wird abgeleitet, um verschiedene Arten von Prozessoren zu definieren. Das folgende Beispiel zeigt einen Prozessor, der `String`-Objekte verarbeitet (beachten Sie, daß die Rückgabetypen kovariant sein dürfen, nicht aber die Parametertypen):

```
//: interfaces/classprocessor/Apply.java
package interfaces.classprocessor;
```

```
import java.util.*;
import static net.mindview.util.Print.*;

class Processor {
    public String name() {
        return getClass().getSimpleName();
    }
    Object process(Object input) { return input; }
}

class Upcase extends Processor {
    String process(Object input) { // Covariant return
        return ((String)input).toUpperCase();
    }
}

class Downcase extends Processor {
    String process(Object input) {
        return ((String)input).toLowerCase();
    }
}

class Splitter extends Processor {
    String process(Object input) {
        // The split() argument divides a String into pieces:
        return Arrays.toString(((String)input).split(" "));
    }
}

public class Apply {
    public static void process(Processor p, Object s) {
        print("Using Processor " + p.name());
        print(p.process(s));
    }
    public static String s =
        "Disagreement with beliefs is by definition incorrect";
    public static void main(String[] args) {
        process(new Upcase(), s);
        process(new Downcase(), s);
        process(new Splitter(), s);
    }
} /* Output:
      Using Processor Upcase
      DISAGREEMENT WITH BELIEFS IS BY DEFINITION INCORRECT
      Using Processor Downcase
      disagreement with beliefs is by definition incorrect
      Using Processor Splitter
      [Disagreement, with, beliefs, is, by, definition, incorrect]
   *///:~
```

Die statische `Apply`-Methode `process()` akzeptiert jeden Prozessor vom Typ `Processor`, wendet ihn auf ein Objekt vom Typ `Object` an und gibt das Ergebnis aus. Die Vorgehensweise, eine Methode so zu definieren, daß ihr Verhalten vom Typ des als Argument übergebenen Objektes abhängt, heißt *Strategy*-Entwurfsmuster. Die Methode repräsentiert den unveränderlichen Anteil des Algorithmus, die Strategie dagegen den veränderlichen. Die Strategie ist das übergebene Objekt und beinhaltet die auszuführenden Anweisungen. Die Klasse `Processor` repräsentiert in diesem Beispiel die Strategie. Sie sehen in der `main()`-Methode drei verschiedene Strategien, angewendet auf das von `s` referenzierte `String`-Objekt.

[25] Die `split()`-Methode gehört zur Klasse `String`, trennt den Inhalt ihres `String`-Objektes gemäß dem als Argument übergebenen Trennzeichen und gibt ein `String`-Array zurück. Die `split()`-Methode tritt hier nur auf, um ohne Mühe ein `String`-Array zu erzeugen.

[26–27] Das nächste Beispiel definiert eine Reihe elektronischer Filter, die sich der `process()`-Methode übergeben lassen:

```
//: interfaces/filters/Waveform.java
package interfaces.filters;

public class Waveform {
    private static long counter;
    private final long id = counter++;
    public String toString() { return ”Waveform ” + id; }
} ///:~

//: interfaces/filters/Filter.java
package interfaces.filters;

public class Filter {
    public String name() {
        return getClass().getSimpleName();
    }
    public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/LowPass.java
package interfaces.filters;

public class LowPass extends Filter {
    double cutoff;
    public LowPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) {
        return input; // Dummy processing
    }
} ///:~

//: interfaces/filters/HighPass.java
package interfaces.filters;

public class HighPass extends Filter {
    double cutoff;
    public HighPass(double cutoff) { this.cutoff = cutoff; }
    public Waveform process(Waveform input) { return input; }
} ///:~

//: interfaces/filters/BandPass.java
package interfaces.filters;

public class BandPass extends Filter {
    double lowCutoff, highCutoff;
    public BandPass(double lowCut, double highCut) {
        lowCutoff = lowCut;
        highCutoff = highCut;
    }
    public Waveform process(Waveform input) { return input; }
} ///:~
```

Die Klassen `Filter` und `Processor` haben identische Schnittstellen. `Filter` ist allerdings nicht von `Processor` abgeleitet, da der Programmierer schließlich nicht vorhersehen konnte, daß Sie sei-

ne Klasse als Prozessor verwenden würden. Sie können der `process()`-Methode der Klasse `Apply` daher kein `Filter`-Objekt übergeben, obwohl es doch eigentlich funktionieren sollte. Im Grunde genommen, ist die Kopplung zwischen der `process()`-Methode und der Klasse `Processor` stärker ausgeprägt als notwendig und diese Eigenschaft verhindert die Wiederverwendbarkeit der Methode. Beachten Sie außerdem, daß Ein- und Ausgabe vom Typ `Waveform` sind.

[28a–28b] Ist `Processor` aber ein Interface, so sind die Einschränkungen genügend gelockert, um die Wiederverwendbarkeit der `process()`-Methode durch einen Parameter des Interfacetyps *`Processor`* zu verbessern. Die überarbeiteten Versionen von *`Processor`* und `Apply` sind:

```
//: interfaces/interfaceprocessor/Processor.java
package interfaces.interfaceprocessor;

public interface Processor {
    String name();
    Object process(Object input);
} ///:~

//: interfaces/interfaceprocessor/Apply.java
package interfaces.interfaceprocessor;
import static net.mindview.util.Print.*;

public class Apply {
    public static void process(Processor p, Object s) {
        print("Using Processor " + p.name());
        print(p.process(s));
    }
} ///:~
```

Eine Möglichkeit zur Wiederwendung implementierter Funktionalität besteht darin, daß die Clientprogrammierer Ihre Klassen konform bezüglich eines Interfaces schreiben, zum Beispiel:

```
//: interfaces/interfaceprocessor/StringProcessor.java
package interfaces.interfaceprocessor;
import java.util.*;

public abstract class StringProcessor implements Processor {
    public String name() {
        return getClass().getSimpleName();
    }
    public abstract String process(Object input);
    public static String s =
        "If she weighs the same as a duck, she's made of wood";
    public static void main(String[] args) {
        Apply.process(new Upcase(), s);
        Apply.process(new Downcase(), s);
        Apply.process(new Splitter(), s);
    }
}

class Upcase extends StringProcessor {
    public String process(Object input) { // Covariant return
        return ((String)input).toUpperCase();
    }
}

class Downcase extends StringProcessor {
    public String process(Object input) {
        return ((String)input).toLowerCase();
    }
```

```
}

class Splitter extends StringProcessor {
    public String process(Object input) {
        return Arrays.toString(((String)input).split(" "));
    }
} /* Output:
      Using Processor Upcase
      IF SHE WEIGHS THE SAME AS A DUCK, SHE'S MADE OF WOOD
      Using Processor Downcase
      if she weighs the same as a duck, she's made of wood
      Using Processor Splitter
      [If, she, weighs, the, same, as, a, duck,, she's, made, of, wood]
   *///:~
```

[29–30] Sie sind allerdings oft in der Situation, die Klassen, die Sie verwenden wollen, nicht ändern zu können. Stellen Sie sich beispielsweise vor, die Bibliothek mit den elektronischen Filtern sei nicht selbst entwickelt worden, sondern bereits vorhanden gewesen. In einem solchen Fall bietet sich das Entwurfsmuster *Adapter* an. Bei einer Implementierung dieses Entwurfsmusters verknüpfen Sie die vorhandene mit der benötigten Schnittstelle, zum Beispiel:

```
//: interfaces/interfaceprocessor/FilterProcessor.java
package interfaces.interfaceprocessor;
import interfaces.filters.*;

class FilterAdapter implements Processor {
    Filter filter;
    public FilterAdapter(Filter filter) {
        this.filter = filter;
    }
    public String name() { return filter.name(); }
    public Waveform process(Object input) {
        return filter.process((Waveform) input);
    }
}

public class FilterProcessor {
    public static void main(String[] args) {
        Waveform w = new Waveform();
        Apply.process(new FilterAdapter(new LowPass(1.0)), w);
        Apply.process(new FilterAdapter(new HighPass(2.0)), w);
        Apply.process(new FilterAdapter(new BandPass(3.0, 4.0)), w);
    }
} /* Output:
      Using Processor LowPass
      Waveform 0
      Using Processor HighPass
      Waveform 0
      Using Processor BandPass
      Waveform 0
   *///:~
```

In dieser Implementierung des *Adapter*-Entwurfsmusters erwartet der Konstruktor der Klasse `FilterAdapter` ein Objekt der vorhandenen Schnittstelle (`Filter`) und liefert ein Objekt mit der benötigen Schnittstelle vom Typ `Processor`. Die Klasse `FilterAdapter` zeigt auch Eigenschaften des Entwurfsmuster *Delegation*.

[31] Die Entkopplung von Schnittstelle und Implementierung gestattet die Anwendung des Interfaces auf viele verschiedene Implementierungen (Ableitungslinien), erhöht also die Wiederverwendbarkeit

Ihrer Klassen und Methoden.

Übungsaufgabe 11: (4) Schreiben Sie eine Klasse mit einer Methode, die ein Argument vom Typ `String` erwartet und die Referenz auf ein `String`-Objekt zurückgibt, in dem die Buchstaben der Eingabe paarweise vertauscht sind. Adaptieren Sie die Klasse, so daß sie der Methode `interfaces-.classprocessor.Apply.process()` übergeben werden kann. ■

10.4 „Mehrfachvererbung“ in Java

[32] Da ein Interface keine Implementierung beinhaltet, sprich mit einem Interface ist kein Platz im Arbeitsspeicher verknüpft, besteht kein Anlaß, eine Kombination aus mehreren Interfaces zu verhindern. Eine solche Kombinationsmöglichkeit ist nützlich, wenn Sie ausdrücken möchten, daß ein `x` sowohl ein `a`, ein `b`, als auch ein `c` ist. In C++ wird eine derartige Kombination aus den Schnittstellen mehrerer Klassen als *Mehrfachvererbung* (*multiple inheritance*) bezeichnet und bringt ziemlich klebriges Gespäck mit sich, da jede Klasse eine eine Implementierung haben kann. Auch Java gestattet die Zuordnung einer Klasse zu mehr als einem Typ, wobei aber nur eine Klasse implementiert sein darf. Die Probleme von C++ treten bei der Kombination mehrerer Interfaces in Java nicht auf (siehe Abbildung 10.3).

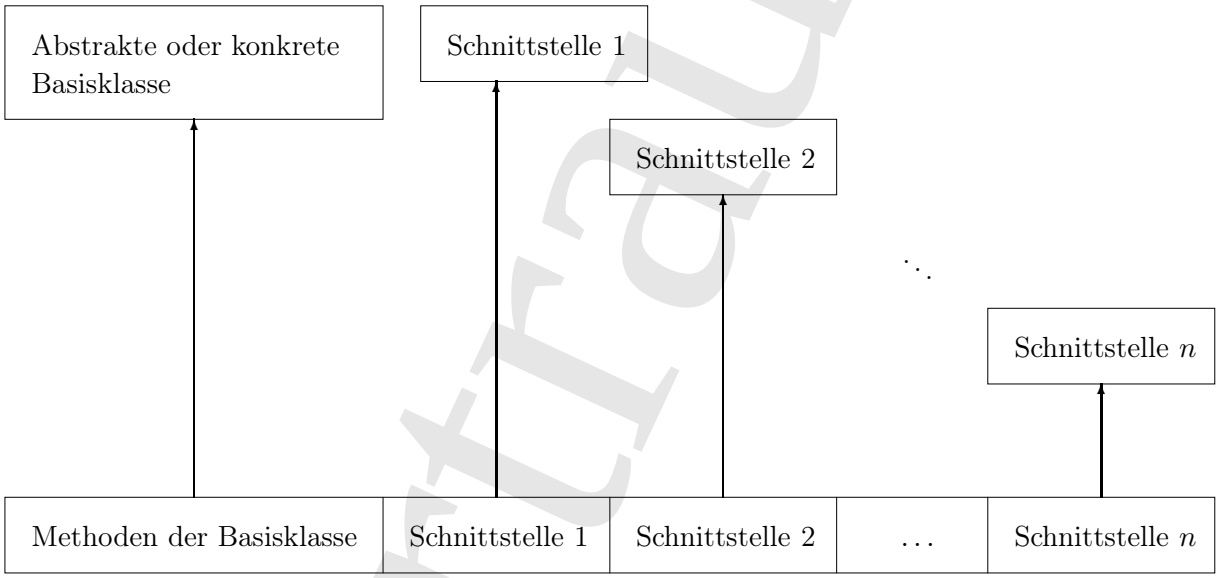


Abbildung 10.3: Beschriftung fehlt noch.

[33] Eine abgeleitete Klasse muß sich nicht zwingend auf entweder eine abstrakte oder eine „konkrete“ Basisklasse beziehen (eine konkrete Klasse enthält keine abstrakten Methoden). Die Ableitung von einem Typ der kein Interface ist, ist nur von einem solchen Typ erlaubt. Alle übrigen Basistypen müssen Interfaces sein. Die Namen der implementierten Interfaces werden als kommaseparierte Liste nach dem Schlüsselwort `implements` angegeben. Die Anzahl der Elemente in dieser Liste ist unbeschränkt. Die Referenz auf ein Objekt einer solchen Klasse kann in jeden Interfacetyp umgewandelt werden, da jedes Interface einen unabhängigen Typ darstellt. Das folgende Beispiel zeigt die Kombination einer konkreten Klasse mit mehreren Interfaces bei der Definition einer neuen Klasse:

```
//: interfaces/Adventure.java
// Multiple interfaces.
```

```
interface CanFight {
    void fight();
}

interface CanSwim {
    void swim();
}

interface CanFly {
    void fly();
}

class ActionCharacter {
    public void fight() {}
}

class Hero extends ActionCharacter implements CanFight, CanSwim, CanFly {
    public void swim() {}
    public void fly() {}
}

public class Adventure {
    public static void t(CanFight x) { x.fight(); }
    public static void u(CanSwim x) { x.swim(); }
    public static void v(CanFly x) { x.fly(); }
    public static void w(ActionCharacter x) { x.fight(); }
    public static void main(String[] args) {
        Hero h = new Hero();
        t(h); // Treat it as a CanFight
        u(h); // Treat it as a CanSwim
        v(h); // Treat it as a CanFly
        w(h); // Treat it as an ActionCharacter
    }
} ///:~
```

[34] Die Klasse `Hero` kombiniert die konkrete Klasse `ActionCharacter` mit den Interfaces *`CanFight`*, *`CanSwim`* und *`CanFly`*. Bei der Kombination einer konkreten Klasse mit Interfaces, muß die Klasse vor den Interfaces deklariert werden (der Compiler gibt andernfalls eine Fehlermeldung aus).

[35] Die Signaturen der `fight()`-Methoden im Interface *`CanFight`* und in der Klasse `ActionCharacter` stimmen überein. Beachten Sie, daß die Klasse `Hero` keine eigene Definition der `fight()`-Methode liefert. Sie können die Schnittstelle einer Klasse zwar erweitern, erhalten dadurch aber eine andere Schnittstelle. Ein Objekt einer Klasse kann aber erst dann erzeugt werden, wenn alle benötigten Definition vorhanden sind. Die Klasse `Hero` definiert die `fight()`-Methode zwar nicht selbst, erbt aber eine Definition von ihrer Basisklasse `ActionCharacter`. Somit ist die Schnittstelle der Klasse `Hero` vollständig und es können `Hero`-Objekte erzeugt werden.

[36] Die Klasse `Adventure` definiert vier Methoden, die jeweils ein Argument vom Typ eines der drei Interfaces beziehungsweise vom Typ der konkreten Klasse `ActionCharacter` erwarten. Das von `h` referenzierte `Hero`-Objekt kann jeder dieser vier Methoden übergeben werden, das heißt die Referenz wird nacheinander in jeden der vier Basistypen umgewandelt. Durch die Funktionsweise der Interfaces bei Java, ist hier keine besondere Anstrengung des Clientprogrammierers erforderlich.

[37] Das obige Beispiel demonstriert eines der Kernargumente für Interfaces: Die aufwärts gerichtete Umwandlung einer Objektreferenz in mehr als einen Basistyp (zusammen mit der dadurch gegebenen Flexibilität). Ein weiteres Argument ist identisch mit der Motivation zur Verwendung abstrakter Basisklassen: Den Clientprogrammierer daran zu hindern, ein Objekt dieses Typs zu erzeugen und festzulegen, daß der Typ lediglich eine Schnittstelle darstellt.

[38] Damit erhebt sich die Frage, ob Sie eher ein Interface oder eine abstrakte Klasse wählen sollten? Wenn Sie eine Basisklasse ohne Methoden- und Felddefinitionen anlegen können, sollten Sie ein Interface vorziehen. Eigentlich können Sie stets ein Interface in Betracht ziehen, wenn Sie die Absicht haben, eine Basisklasse anzulegen. (Dieses Thema wird in der Zusammenfassung dieses Kapitels noch einmal aufgegriffen, siehe Abschnitt 10.10.)

Übungsaufgabe 12: (2) Definieren Sie im Beispiel *Adventure.java* ein Interface `CanClimb`. Orientieren Sie sich dabei an den drei anderen Interfaces. ■

Übungsaufgabe 13: (2) Definieren Sie ein Interface und leiten Sie zwei neue Interfaces davon ab. Leiten Sie ein viertes Interfaces von den beiden vorigen ab („Mehrfachvererbung“).[3] ■

10.5 Erweiterung von Interfaces durch „Ableitung“

[39] Sie können per Ableitung ein Interface mühelos um neue Methoden ergänzen und mehrere Interfaces zu einem neuen Interface kombinieren. Das folgende Beispiel zeigt, daß Sie in beiden Fällen ein neues Interfaces bekommen:

```
//: interfaces/HorrorShow.java
// Extending an interface with inheritance.

interface Monster {
    void menace();
}

interface DangerousMonster extends Monster {
    void destroy();
}

interface Lethal {
    void kill();
}

class DragonZilla implements DangerousMonster {
    public void menace() {}
    public void destroy() {}
}

interface Vampire extends DangerousMonster, Lethal {
    void drinkBlood();
}

class VeryBadVampire implements Vampire {
    public void menace() {}
    public void destroy() {}
    public void kill() {}
    public void drinkBlood() {}
}

public class HorrorShow {
    static void u(Monster b) { b.menace(); }
    static void v(DangerousMonster d) {
        d.menace();
        d.destroy();
    }
    static void w(Lethal l) { l.kill(); }
    public static void main(String[] args) {
```

[3] Dies zeigt, wie Interfaces das Diamond-Problem der Mehrfachvererbung von C++ lösen.

```
        DangerousMonster barney = new DragonZilla();
        u(barney);
        v(barney);
        Vampire vlad = new VeryBadVampire();
        u(vlad);
        v(vlad);
        w(vlad);
    }
} ///:~
```

DangerousMonster ist eine Erweiterung von *Monster* und wiederum ein Interface. Die Klasse `DragonZilla` implementiert dieses Interface.

[40] Die beim Interface *Vampire* verwendete `extends`-Syntax funktioniert *ausschließlich* bei Interfaces. Bei einer Klassendefinition ist nach `extends` nur eine einzige Klasse erlaubt. Die Definition eines Interfaces kann sich dagegen nach `extends` auf mehrere Interfaces beziehen, deren Namen als kommaseparierte Liste angegeben werden.

Übungsaufgabe 14: (2) Definieren Sie drei Interfaces mit je zwei Methoden. Leiten Sie ein neues Interface ab, das die zuerst definierten drei Interfaces kombiniert und eine weitere Methode deklariert. Schreiben Sie eine Klasse, die das vierte Interface implementiert und außerdem von einer konkreten Klasse abgeleitet wird. Schreiben Sie vier Methoden, die jeweils eines der vier Interfaces als Argumenttyp erwarten. Erzeugen Sie in der `main()`-Methode ein Objekt Ihrer Klasse und übergeben Sie es jeder der vier Methoden. ■

Übungsaufgabe 15: (2) Ändern Sie Übungsaufgabe 14, indem Sie die Klasse von einer selbst definierten abstrakten Klasse ableiten. ■

10.5.1 Nameskollisionen beim Kombinieren von Interfaces

[41] Es gibt beim Implementieren mehrerer Interfaces eine kleine Falle. Im Beispiel *Adventure.java* auf Seite 261f kommt sowohl im Interface *CanFight* als auch in der Klasse `ActionCharacter` die Methode `fight()` mit Rückgabetyp `void` vor. Identische Methodennamen sind unproblematisch, aber was geschieht, wenn sich die Signaturen oder Rückgabetypen unterscheiden? Ein Beispiel:

```
//: interfaces/InterfaceCollision.java
package interfaces;

interface I1 { void f(); }
interface I2 { int f(int i); }
interface I3 { int f(); }
class C { public int f() { return 1; } }

class C2 implements I1, I2 {
    public void f() {}
    public int f(int i) { return 1; } // overloaded
}

class C3 extends C implements I2 {
    public int f(int i) { return 1; } // overloaded
}

class C4 extends C implements I3 {
    // Identical, no problem:
    public int f() { return 1; }
}

// Methods differ only by return type:
```

```
//! class C5 extends C implements I1 {}
//! interface I4 extends I1, I3 {} ///:~
```

[42] Das Problem besteht in der ungünstigen Kombination von Überschreibung, Implementierung und Überladung. Überladene Methoden dürfen sich nicht nur durch ihren Rückgabetyp unterscheiden. Werden die Kommentarzeichen vor den beiden letzten Zeilen entfernt, so lautet die Fehlermeldung des Compilers:

```
InterfaceCollision.java:24: interfaces.C5 is not abstract and does not \
    override abstract method f() in interfaces.I1
class C5 extends C implements I1 {}
^
InterfaceCollision.java:25: types interfaces.I3 and interfaces.I1 are \
    incompatible; both define f(), but with unrelated return types
interface I4 extends I1, I3 {} ///:~
^
2 errors
```

Identische Methodennamen in verschiedenen, zur Kombination vorgesehenen Interfaces stiften außerdem beim Lesen des Quelltextes im allgemeinen Verwirrung. Versuchen Sie, diese Situation zu vermeiden.

10.6 Adaptieren einer Klasse an ein Interface

[43] Einer der wichtigsten Gründe für die Aufnahme von Interfaces in den Sprachumfang besteht darin, daß sie mehrere Implementierungen ein und derselben Schnittstelle ermöglichen. Ein einfaches Beispiel ist eine Methode, die ein Argument vom Typ eines Interfaces erwartet. Es bleibt Ihnen überlassen, das Interface zu implementieren und der Methode ein entsprechendes Objekt zu übergeben.

[44] Interfaces treten häufig im Rahmen des bereits erwähnten Entwurfsmusters *Strategy* auf. Sie definieren eine Methoden, die bestimmte Operationen ausführt und ein Argument vom Typ eines Interfaces erwartet, das Sie ebenfalls vorgeben. Im Grunde genommen kann Ihrer Methode jedes Objekt übergeben werden, solange es konform zu Ihrem Interface ist. Ihre Methode wird dadurch flexibler, allgemeiner und läßt sich besser wiederverwenden.

[45–46] Die seit Version 5 der Java Standard Edition (SE 5) vorhandene Klasse `java.util.Scanner` (die Sie in Abschnitt 14.7 kennenlernen werden) hat einen Konstruktor, der ein Argument des Interfacetyps *`java.lang.Readable`* erwartet. Das Interface *`Readable`* tritt bei keiner anderen Methode in der Standardbibliothek von Java als Argumenttyp auf, sondern wurde eigens für `Scanner` definiert, damit die Konstruktoren nicht auf eine bestimmte Klasse beschränkt werden müssen. Die Klasse `Scanner` kann auf diese Weise mit mehr Typen zusammenarbeiten. Wenn Sie eine neue Klasse schreiben und mit `Scanner` kombinieren wollen, so implementieren Sie das Interface *`Readable`*, zum Beispiel:

```
//: interfaces/RandomWords.java
// Implementing an interface to conform to a method.
import java.nio.*;
import java.util.*;

public class RandomWords implements Readable {
    private static Random rand = new Random(47);
    private static final char[] capitals =
        ''ABCDEFGHIJKLMNOPQRSTUVWXYZ''.toCharArray();
    private static final char[] lowers =
```

```
        ''abcdefghijklmnopqrstuvwxyz''.toCharArray();
    private static final char[] vowels =
        ''aeiou''.toCharArray();
    private int count;
    public RandomWords(int count) { this.count = count; }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1; // Indicates end of input
        cb.append(capitals[rand.nextInt(capitals.length)]);
        for(int i = 0; i < 4; i++) {
            cb.append(vowels[rand.nextInt(vowels.length)]);
            cb.append(lowers[rand.nextInt(lowers.length)]);
        }
        cb.append('' '');
        return 10; // Number of characters appended
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new RandomWords(10));
        while(s.hasNext())
            System.out.println(s.next());
    }
} /* Output:
      Yazeruyac
      Fowenucor
      Goeazimom
      Raeuuacio
      Nuoadesiw
      Hageaikux
      Ruqicibui
      Numasetih
      Kuuuuozog
      Waqizeyoy
   *///:~
```

Das Interface *Readable* verlangt lediglich die Implementierung einer einzigen Methode: `read()`. Die `read()`-Methode erwartet ein Argument vom Typ `CharBuffer`, an dessen Inhalt Sie per `append()` Zeichen anhängen (es gibt mehrere Möglichkeiten, einem `CharBuffer`-Objekt Werte zu übergeben, siehe API-Dokumentation der Klasse `CharBuffer`), oder -1 zurückgeben, wenn die Eingabe beendet ist.

[47] Angenommen, eine Klasse implementiert *Readable* noch nicht. Wie können Sie vorgehen, um diese Klasse mit `Scanner` kombinierbar zu machen? Die folgende Klasse liefert zufällig bestimmte Fließkommawerte:

```
//: interfaces/RandomDoubles.java
import java.util.*;

public class RandomDoubles {
    private static Random rand = new Random(47);
    public double next() { return rand.nextDouble(); }
    public static void main(String[] args) {
        RandomDoubles rd = new RandomDoubles();
        for(int i = 0; i < 7; i ++)
            System.out.print(rd.next() + '' '');
    }
} /* Output:
      0.7271157860730044 0.5309454508634242 0.16020656493302599 \
      0.18847866977771732 0.5166020801268457 0.2678662084200585 \
```

```
      0.2613610344283964
   *///:~
```

[48] Wir implementieren nochmals das *Adapter*-Entwurfsmuster. Die Adapterklasse wird diesesmal durch Ableiten von der Klasse `RandomDoubles` und Implementieren des Interfaces ***`Readable`*** definiert. Die Pseudo-Mehrfachvererbung mit Hilfe von `implements` liefert eine neue Klasse, die sowohl dem Typ `RandomDoubles` als auch ***`Readable`*** angehört:

```
//: interfaces/AdaptedRandomDoubles.java
// Creating an adapter with inheritance.
import java.nio.*;
import java.util.*;

public class AdaptedRandomDoubles extends RandomDoubles
    implements Readable {
    private int count;
    public AdaptedRandomDoubles(int count) {
        this.count = count;
    }
    public int read(CharBuffer cb) {
        if(count-- == 0)
            return -1;
        String result = Double.toString(next()) + " ";
        cb.append(result);
        return result.length();
    }
    public static void main(String[] args) {
        Scanner s = new Scanner(new AdaptedRandomDoubles(7));
        while(s.hasNextDouble())
            System.out.print(s.nextDouble() + " ");
    }
} /* Output:
      0.7271157860730044 0.5309454508634242 0.16020656493302599\
      0.18847866977771732 0.5166020801268457 0.2678662084200585\
      0.2613610344283964
   *///:~
```

[49] Da Sie jede Klasse um ein Interface ergänzen können, läßt sich jede Methode, die ein Argument eines Interfacetyps erwartet, per Adapter auf jede beliebige Klasse anwenden. Dies zeigt die Mächtigkeit von Interfaces anstelle von Klassen.

Übungsaufgabe 16: (3) Schreiben Sie eine Klasse, die eine Reihe von `char`-Werten erzeugt. Adaptieren Sie diese Klasse, so daß Sie einem `Scanner`-Objekt als Eingabe übergeben werden kann. ■

10.7 Deklaration von Feldern in Interfaces

[50] Da jedes in einem Interface deklarierte Feld automatisch statisch und final ist, sind Interfaces ein komfortables Hilfsmittel zur Deklaration von Gruppen von Konstanten. Vor der SE 5 war dies die einzige Möglichkeit, um den Effekt eines Aufzählungstyp von C/C++ zu erhalten. Eventuell stoßen Sie auf älteren Java-Quelltexte vor der SE 5, wie in diesem Beispiel:

```
//: interfaces/Months.java
// Using interfaces to create groups of constants.
package interfaces;

public interface Months {
    int
```

```
        JANUARY = 1, FEBRUARY = 2, MARCH = 3,
        APRIL = 4, MAY = 5, JUNE = 6, JULY = 7,
        AUGUST = 8, SEPTEMBER = 9, OCTOBER = 10,
        NOVEMBER = 11, DECEMBER = 12;
} ///:~
```

[51] Beachten Sie die Benennung statischer finaler Felder gemäß der Formatierungsrichtlinie durchgängig in Großbuchstaben (mehrere Worte in einem Bezeichner werden durch Unterstriche getrennt). Die Formatierungsrichtlinie gilt gleichermaßen bei konstanten und nicht-konstanten Initialisierungsausdrücken. In einem Interface deklarierte Felder sind automatisch öffentlich, so daß der Modifikator nicht explizit angegeben ist.

[52] Ab der SE 5 stehen Ihnen die viel mächtigeren und flexibleren Aufzählungstypen zur Verfügung, so daß die Verwendung von Interfaces zur Deklaration von Konstanten in den Hintergrund tritt. Allerdings werden Sie die veraltete Schreibweise wahrscheinlich noch häufig in älteren Quelltexten vorfinden. (In den Online-Anhängen zu diesem Buch, unter der Webadresse *http://www.mindview.net*, finden Sie eine vollständige Beschreibung des Ansatzes zur Definition von Aufzählungstypen per Interface vor der SE 5.) Weitere Einzelheiten über die Verwendung von Aufzählungstypen finden Sie in Kapitel 20.

Übungsaufgabe 17: (2) Zeigen Sie, daß die in einem Interface deklarierten Felder implizit statisch und final sind. ■

10.7.1 Initialisierung mit nicht-konstanten Ausdrücken

[53–54] Jedes in einem Interface deklarierte Feld muß initialisiert werden. Dabei können nicht-konstante Ausdrücke verwendet werden, zum Beispiel:

```
//: interfaces/RandVals.java
// Initializing interface fields with
// non-constant initializers.
import java.util.*;

public interface RandVals {
    Random RAND = new Random(47);
    int RANDOM_INT = RAND.nextInt(10);
    long RANDOM_LONG = RAND.nextLong() * 10;
    float RANDOM_FLOAT = RAND.nextLong() * 10;
    double RANDOM_DOUBLE = RAND.nextDouble() * 10;
} ///:~
```

Da diese Felder statisch sind, werden Sie beim erstmaligen Laden der Klasse initialisiert, wobei das Laden beim ersten Zugriff auf eines der Felder ausgelöst wird. Ein einfaches Testprogramm für die zuvor deklarierten Konstanten:

```
//: interfaces/TestRandVals.java
import static net.mindview.util.Print.*;

public class TestRandVals {
    public static void main(String[] args) {
        print(RandVals.RANDOM_INT);
        print(RandVals.RANDOM_LONG);
        print(RandVals.RANDOM_FLOAT);
        print(RandVals.RANDOM_DOUBLE);
    }
} /* Output:
      8
```

```
        -32032247016559954
        -8.5939291E18
        5.779976127815049
    *///:~
```

Die Felder sind selbstverständlich kein Teil der Schnittstelle, sondern werden im statischen Speicherbereich für dieses Interface deponiert.

10.8 Schachtelung von Interfaces

[55–56] Interfaces können in Klassen und anderen Interfaces definiert werden.[4] Dabei zeigen sich einige interessante Eigenschaften:

```
//: interfaces/nesting/NestingInterfaces.java
package interfaces.nesting;

class A {
    interface B {
        void f();
    }
    public class BImp implements B {
        public void f() {}
    }
    private class BImp2 implements B {
        public void f() {}
    }
    public interface C {
        void f();
    }
    class CImp implements C {
        public void f() {}
    }
    private class CImp2 implements C {
        public void f() {}
    }
    private interface D {
        void f();
    }
    private class DImp implements D {
        public void f() {}
    }
    public class DImp2 implements D {
        public void f() {}
    }
    public D getD() { return new DImp2(); }
    private D dRef;
    public void receiveD(D d) {
        dRef = d;
        dRef.f();
    }
}

interface E {
    interface G {
        void f();
```

[4] Danke an Martin Danner, der diese Frage während einer Schulung gestellt hat.

```
        }
        // Redundant ''public'':
        public interface H {
            void f();
        }
        void g();
        // Cannot be private within an interface:
        //! private interface I {}
    }

    public class NestingInterfaces {
        public class BImp implements A.B {
            public void f() {}
        }
        class CImp implements A.C {
            public void f() {}
        }
        // Cannot implement a private interface except
        // within that interface's defining class:
        //! class DImp implements A.D {
        //! public void f() {}
        //! }
        class EImp implements E {
            public void g() {}
        }
        class EGImp implements E.G {
            public void f() {}
        }
        class EImp2 implements E {
            public void g() {}
            class EG implements E.G {
                public void f() {}
            }
        }
        public static void main(String[] args) {
            A a = new A();
            // Can't access A.D:
            //! A.D ad = a.getD();
            // Doesn't return anything but A.D:
            //! A.DImp2 di2 = a.getD();
            // Cannot access a member of the interface:
            //! a.getD().f();
            // Only another A can do anything with getD():
            A a2 = new A();
            a2.receiveD(a.getD());
        }
    } ///:~
```

Die Syntax zur Definition eines Interfaces in einer Klasse ist selbstverständlich. Geschachtelte Interfaces können, wie nicht-geschachtelte Interfaces, unter öffentlichem Zugriff und unter Packagezugriff stehen.

[57] Die Schachtelung von Interfaces in Klassen hat noch eine überraschende Eigenschaft: Derartige Interfaces können als privat deklariert werden, siehe `A.D` (für geschachtelte Klassen und geschachtelte Interfaces gilt dieselbe Qualifizierungssyntax). Doch wozu taugt ein geschachteltes privates Interface? Eventuell vermuten Sie, daß ein geschachteltes privates Interface nur in Form einer privaten inneren Klasse implementiert werden kann (wie `A.DImp`). Aber `A.DImp2` zeigt, daß das Interface

ebenso als öffentliche Klasse implementiert werden kann. ~~However, A.DImp2 can only be used as itself.~~ Sie dürfen die Tatsache, daß `DImp2` das private Interface *`D`* implementiert, nicht erwähnen. Das Implementieren eines privaten Interfaces ist eine Möglichkeit, um die Definition aller dort deklarierten Methoden zu erzwingen, ~~without adding any type information (that is, without allowing any upcasting)~~.

[58] Die Methode `getD()` verdeutlicht ein anderes Dilemma bei geschachtelten privaten Interfaces: `getD()` ist eine öffentliche Methode, die eine Referenz vom Typ eines privaten Interfaces zurückgibt. Was können Sie mit dem Rückgabewert dieser Methode anfangen? Die `main()`-Methode zeigt einige Versuche, den Rückgabewert zu verwenden, die aber alle scheitern. Der einzige funktionierende Fall ist, daß der Rückgabewert einem Objekt übergeben wird, welches berechtigt ist, ihn zu nutzen, hier über die `receiveD()`-Methode einem `A`-Objekt.

[59] Das Interface *`E`* zeigt, daß Interfaces ineinander verschachtelt werden können. Die Regeln für Interfaces, besonders die, daß alle Komponenten eines Interfaces öffentlich sein müssen, werden strikt eingefordert. Ein in einem Interface verschachteltes Interface ist somit automatisch öffentlich und kann nicht als privat deklariert werden.

[60] Die Klasse `NestingInterfaces` zeigt die verschiedenen Möglichkeiten zur Implementierung eines geschachtelten Interfaces. Beachten Sie, daß Sie beim Implementieren eines Interfaces nicht gezwungen sind, darin geschachtelte Interfaces ebenfalls zu implementieren. Als privat deklarierte Interfaces können außerhalb ihrer definierenden Klasse nicht implementiert werden.

[61] Auf den ersten Blick scheinen diese Eigenschaften nur um der strikten syntaktischen Konsistenz Willen vorhanden zu sein. Ich finde allerdings, daß Sie, nachdem Sie sich mit einer Eigenschaft vertraut gemacht haben, häufig Situationen bemerken, in denen sie sich verwerten läßt.

10.9 Interfaces und das Entwurfsmuster Factory-Method

[62–63] Ein Interface ist als Verzweigungspunkt zu mehreren Implementierungen gedacht. Das Entwurfmuster *Factorymethod* ist ein typisches Verfahren, um Objekte zu erzeugen, die zu einer Schnittstelle passen. Anstelle eines direkten Konstruktoraufrufes, rufen Sie eine spezielle Methode eines Fabrikobjektes auf, die die Referenz auf ein Objekt zurückgibt, welches das Interface implementiert. Damit ist Ihr Programm theoretisch vollständig von der Implementierung des Interfaces isoliert, so daß eine Implementierung transparent durch eine andere ersetzt werden kann. Das folgende Beispiel zeigt die Struktur des Entwurfsmusters *Factorymethod*:

```
//: interfaces/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    Implementation1() {} // Package access
    public void method1() { print("Implementation1 method1"); }
    public void method2() { print("Implementation1 method2"); }
}

class Implementation1Factory implements ServiceFactory {
```

```
    public Service getService() {
        return new Implementation1();
    }
}

class Implementation2 implements Service {
    Implementation2() {} // Package access
    public void method1() { print("Implementation2 method1"); }
    public void method2() { print("Implementation2 method2"); }
}

class Implementation2Factory implements ServiceFactory {
    public Service getService() {
        return new Implementation2();
    }
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(new Implementation1Factory());
        // Implementations are completely interchangeable:
        serviceConsumer(new Implementation2Factory());
    }
} /* Output:
      Implementation1 method1
      Implementation1 method2
      Implementation2 method1
      Implementation2 method2
   *///:~
```

Ohne die Fabrikmethode müßte der Quelltext irgendwo den exakten Typ des erzeugten *Service*-Objektes angeben, um den entsprechenden Konstruktor aufrufen zu können.

[64–65] In welchen Situationen sollten Sie diese zusätzliche Indirektionsstufe einführen? Das Anlegen eines Frameworks ist ein häufiger Grund. Stellen Sie sich ein System vor, das Spiele spielt, beispielsweise Schach und Dame auf demselben Brett:

```
//: interfaces/Games.java
// A Game framework using Factory Methods.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
}

class CheckersFactory implements GameFactory {
    public Game getGame() { return new Checkers(); }
}
```

```
class Chess implements Game {
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
        return ++moves != MOVES;
    }
}

class ChessFactory implements GameFactory {
    public Game getGame() { return new Chess(); }
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while(s.move())
            ;
    }
    public static void main(String[] args) {
        playGame(new CheckersFactory());
        playGame(new ChessFactory());
    }
} /* Output:
      Checkers move 0
      Checkers move 1
      Checkers move 2
      Chess move 0
      Chess move 1
      Chess move 2
      Chess move 3
   *///:~
```

Repräsentiert die Klasse `Games` komplexe Funktionalität, so gestattet Ihnen dieser Ansatz die Wiederverwendung dieser Funktionalität mit verschiedenen Spielen. Sie können sich sorgfältiger ausgearbeitete Spiele vorstellen, die von diesem Ansatz profitieren.

[66] Im nächsten Kapitel lernen Sie einen eleganteren Ansatz für die Implementierung von Fabrikklassen und -methoden mit inneren Klassen kennen.

Übungsaufgabe 18: (2) Definieren Sie ein Interface *`Cycle`* und Implementierungen `UniCycle`, `BiCycle` und `TriCycle`. Legen Sie zu jeder Implementierung eine Fabrikmethode und eine Anweisung an, die die Fabrikmethode aufruft. ■

Übungsaufgabe 19: (3) Entwickeln Sie ein Framework, das über Fabrikmethoden das Werfen von Münzen und Würfeln simuliert. ■

10.10 Zusammenfassung

[67] Die Entscheidung, Interfaces für gut zu befinden und daher stets Interfaces anstelle konkreter Basisklassen zu wählen ist nicht unattraktiv. In fast jeder Situation, in der Sie eine Basisklasse anlegen, könnten Sie statt dessen ein Interface und eine Fabrikklasse wählen.

[68] Viele Programmierer haben dieser Versuch nachgegeben und Interfaces mit Fabrikklassen angelegt, wann immer es möglich war. Der Hintergedanke besteht vermutlich darin, daß eventuell einmal eine andere Implementierung erforderlich werden könnte, so daß es stets ratsam ist, die zusätzliche

Abstraktionsstufe einzuführen. Dieser Ansatz hat sich zu einer Art verfrühter Design-Optimierung entwickelt.

[69] Jede Form von Abstraktion sollte aber durch den tatsächlichen Bedarf motiviert sein. Interfaces sind Komponenten, die gegebenenfalls im Rahmen einer Refaktorierung eingeführt werden können, statt überall eine zusätzliche Indirektionsebene mit der zugehörigen Komplexität zu installieren. Dieser Zusatzaufwand ist beträchtlich. Stellen Sie sich einen Programmier vor, der sich durch all diese Komplexität arbeiten muß, und schließlich erkennt, daß Sie Interfaces „nur für den Fall“ und nicht etwa aus einem zwingenden Grund verwendet haben. Wenn ich auf einen solches Programm stoße, fange ich an, alle Entwürfe dieses Autors in Frage zu stellen.

[70] Eine geeignete Richtlinie ist es, Klassen Vorrang vor Interface zu geben. Beginnen Sie mit Klassen und refaktorieren Sie Ihr Programm, wenn sich zeigt, daß Interfaces benötigt werden. Interfaces sind großartige Hilfsmittel, ihre Verwendung kann aber leicht übertrieben werden.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 11

Innere Klassen

Inhaltsübersicht

[0] Im Körper einer Klasse können wiederum Klassen definiert werden. Solche Klassen heißen innere Klassen.

[1] Innere Klassen sind ein nützliches Konzept, da sie Ihnen gestatten, logisch zusammengehörige Klassen zu gruppieren und die Sichtbarkeit der inneren Klasse in der äußeren Klassen zu kontrollieren. Es ist aber wichtig, zu verstehen, daß sich innere Klassen konzeptionell von der Komposition unterscheiden.

[2] Auf den ersten Blick wirken innere Klassen wie ein einfacher Mechanismus, um Funktionalität zu verbergen, indem eine Klasse in eine andere Klasse eingesetzt wird. In diesem Kapitel lernen Sie, daß innere Klassen noch andere Eigenschaften und Fähigkeiten haben, beispielsweise „kennt“ eine innere Klasse ihre äußere Klasse und ist in der Lage mit dieser zu kommunizieren. Überdies gestatten innere Klassen eleganteren und verständlicheren Quelltext, obwohl sie diese Eigenschaften natürlich nicht garantieren können.

[3] Zu Anfang wirken innere Klassen recht sonderbar und es vergeht einige Zeit, bis Sie sich daran gewöhnt haben und sie in Ihren Entwürfen verwenden. Die Notwendigkeit innerer Klassen ist nicht immer offensichtlich, aber nach der Beschreibung der grundlegenden Syntax und Bedeutung dokumentiert Abschnitt 11.8 „Motivation zur Anwendung innerer Klassen“ die Vorzüge ihrer Verwendung.

[4] Der Rest des Kapitel, nach Abschnitt 11.8, widmet sich der Untersuchung der Syntax innerer Klassen in allen Einzelheiten. Die dort vorgestellten Eigenschaften und Fähigkeiten sind zur vollständigen Beschreibung des Konzeptes gedacht, so daß Sie sie wenigstens zu Anfang noch nicht einsetzen werden. Die ersten Abschnitte dieses Kapitel mögen für Ihre gegenwärtigen Bedürfnisse ausreichen. Behandeln Sie die detaillierten Betrachtung als Referenz zum Nachschlagen.

11.1 Definition innerer Klassen

[5–6] Sie legen eine innerer Klasse an, indem Sie ihre Definition in die Definition der umgebenden Klasse einsetzen:

```
//: innerclasses/Parcel1.java
// Creating inner classes.

public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    // Using inner classes looks just like
    // using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();
        Destination d = new Destination(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel1 p = new Parcel1();
        p.ship(''Tasmania'');
    }
} /* Output:
      Tasmania
   *///:~
```

Die in der `ship()`-Methode verwendeten inneren Klassen sehen wie gewöhnliche Klassen aus. Der einzige praktische Unterschied besteht darin, daß sie in der Klasse `Parcel1` verschachtelt sind. In Abschnitt 11.2 werden Sie erkennen, daß dies nicht der einzige Unterschied ist.

[7–8] Typischerweise definiert die äußere Klasse eine Methode, welche die Referenz auf ein Objekt einer inneren Klasse zurückgibt, wie die Methoden `to()` und `contents()` im folgenden Beispiel:

```
//: innerclasses/Parcel2.java
// Returning a reference to an inner class.
```

```
public class Parcel2 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {
            label = whereTo;
        }
        String readLabel() { return label; }
    }
    public Destination to(String s) {
        return new Destination(s);
    }
    public Contents contents() {
        return new Contents();
    }
    public void ship(String dest) {
        Contents c = contents();
        Destination d = to(dest);
        System.out.println(d.readLabel());
    }
    public static void main(String[] args) {
        Parcel2 p = new Parcel2();
        p.ship(''Tasmania'');
        Parcel2 q = new Parcel2();
        // Defining references to inner classes:
        Parcel2.Contents c = q.contents();
        Parcel2.Destination d = q.to(''Borneo'');
    }
} /* Output:
      Tasmania
   *///:~
```

Wenn Sie außerhalb einer nicht-statischen Methode der äußeren Klasse ein Objekt einer inneren Klassen erzeugen wollen, müssen Sie den Typ dieses Objektes in der Schreibweise `OuterClassName.InnerClassName` deklarieren, wie in der `main()`-Methode.

Übungsaufgabe 1: (1) Schreiben Sie eine Klasse namens `Outer`, die eine Klasse namens `Inner` beinhaltet. Legen Sie in `Outer` eine Methode an, die ein Objekt der Klasse `Inner` erzeugt und die Referenz auf dieses Objekt zurückgibt. Definieren und Bewerten Sie in der `main()`-Methode eine Referenzvariable vom Typ `Inner`. ■

11.2 Referenz auf das Objekt der äußeren Klasse

[9] Bis jetzt scheinen innere Klassen nur ein Schema zu sein, um Bezeichner verbergen und den Quelltext organisieren zu können, das heißt sie sind nützlich aber nicht unwiderstehlich. Die Geschichte nimmt aber eine unerwartete Wendung: Ein Objekt einer inneren Klasse verfügt nämlich über eine Verknüpfung zu dem Objekt der äußeren Klasse, mit dessen Hilfe es erzeugt wurde und ist in der Lage, die Komponenten „seines“ äußeren Objektes ohne spezielle Qualifizierung zu erreichen. Innere Klassen haben uneingeschränkte Zugriffsberechtigung auf alle Bestandteile der äußeren

Klasse,[1] wie das folgende Beispiel demonstriert:

```
//: innerclasses/Sequence.java
// Holds a sequence of Objects.

interface Selector {
    boolean end();
    Object current();
    void next();
}

public class Sequence {
    private Object[] items;
    private int next = 0;
    public Sequence(int size) { items = new Object[size]; }
    public void add(Object x) {
        if(next < items.length)
            items[next++] = x;
    }
    private class SequenceSelector implements Selector {
        private int i = 0;
        public boolean end() { return i == items.length; }
        public Object current() { return items[i]; }
        public void next() { if (i < items.length) i++; }
    }
    public Selector selector() {
        return new SequenceSelector();
    }
    public static void main(String[] args) {
        Sequence sequence = new Sequence(10);
        for(int i = 0; i < 10; i++)
            sequence.add(Integer.toString(i));
        Selector selector = sequence.selector();
        while(!selector.end()) {
            System.out.print(selector.current() + " ");
            selector.next();
        }
    }
} /* Output:
      0 1 2 3 4 5 6 7 8 9
   *///:~
```

[10] Die Klasse `Sequence` repräsentiert eine endliche Folge von ganzen Zahlen und ist eine Wrapperklasse um ein Array vom Typ `Object`. Sie können mit Hilfe der `add()`-Methode ein neues `Object`-Objekt hinter dem zuvor gespeicherten Element einsetzen, sofern die Kapazität des Arrays noch nicht erschöpft ist. Das Interface *`Selector`* deklariert die zum Abfragen der Elemente des Arrays benötigten Methoden und ist ein Beispiel für das Entwurfsmuster *Iterator*, über das Sie in diesem Buch noch mehr erfahren werden. Die Methode `end()` gibt an, ob das Ende der Folge erreicht ist, die Methode `current()` liefert die Referenz auf das gegenwärtig ausgewählte Element und die Methode `next()` bewegt den Selektor vorwärts zum nächsten Element. Als Interface gestattet *`Selector`* universeller verwendbaren Quelltext, da Klassen das Interface individuell implementieren und Methoden Argumente dieses Typs definieren können.

[1] Dies ist ein deutlicher Unterschied zu den geschachtelten Klassen von C++, welche nur ein Mechanismus sind, um Namen zu verbergen. Die geschachtelten Klassen von C++ haben weder eine Verbindung zu einem Objekt der äußeren Klasse noch implizierte Berechtigungen.

[11] Die Implementierung `SequenceSelector` ist eine private Klasse, welche die in *`Selector`* deklarierte Funktionalität zur Verfügung stellt. Die `main()`-Methode erzeugt ein `Sequence`-Objekt und setzt einige Elemente vom Typ `String` ein. Danach fordert `main()` über die `selector()`-Methode die Referenz auf ein *`Selector`*-Objekt an und verwendet es, um die Folge elementweise zu durchlaufen.

[12] Die Definition der Klasse `SequenceSelector` wirkt auf den ersten Blick wie die vorigen Beispiele für innere Klassen, aber sehen Sie genau hin: Jede der drei Methoden `end()`, `current()` und `next()` bezieht sich auf die Referenzvariable `items`, die nicht in `SequenceSelector` definiert, sondern ein privates Feld der äußeren Klasse `Sequence` ist. Die innere Klase ist in der Lage, auf Methoden und Felder der äußeren Klasse zuzugreifen, als ob sie in der inneren Klasse selbst definiert wären. Diese Eigenschaft stellt sich als sehr komfortabel heraus, wie Sie im obigen Beispiel bereits erkennen können.

[13] Ein Objekt einer inneren Klasse hat also automatisch Zugriff auf die Komponenten „seines“ Objektes der äußeren Klasse. Wie geht das vor sich? Ein Objekt einer inneren Klasse speichert während seiner Erzeugung die Referenz auf das Objekt der äußeren Klasse, das die Objekterzeugung verursacht hat. Diese Referenz wird anschließend bei Zugriffen auf die Komponenten des äußeren Objektes verwendet, um die entsprechende Komponente auszuwählen. Der Compiler kümmert sich um alle erforderlichen Einzelheiten. Sie sehen nun, daß ein Objekt einer inneren Klasse stets nur in Verbindung mit einem Objekt der äußeren Klassen erzeugt werden kann (bei nicht-statischen inneren Klassen). Das Erzeugen eines Objektes einer inneren Klasse setzt die Referenz auf ein Objekt der äußeren Klasse voraus und der Compiler beschwert sich, wenn diese Referenz nicht verfügbar ist. Die meiste Zeit über funktioniert dieser Mechanismus ohne Eingriff des Programmierers.

Übungsaufgabe 2: (1) Schreiben Sie eine Klasse mit einem `String`-Feld und einer `toString()`-Methode, um den Inhalt des `String`-Feldes anzuzeigen. Setzen Sie mehrere Objekte dieser Klasse als Elemente in ein `Sequence`-Objekt ein und geben Sie die Elemente anschließend aus. ■

Übungsaufgabe 3: (1) Ändern Sie Übungsaufgabe 1, so daß die äußere Klasse ein privates `String`-Feld (Initialisierung per Konstruktor) und die innere Klasse eine `toString()`-Methode definiert, um den Inhalt des `String`-Feldes anzuzeigen. Erzeugen Sie ein Objekt der inneren Klasse und zeigen Sie es an. ■

11.3 Die .this- und .new-Syntax

[14] Sie erhalten die Referenz auf das Objekt der äußeren Klasse, indem Sie den Namen der äußeren Klasse und das Schlüsselwort `this` über einen Punkt miteinander verbinden. Die erhaltene Referenz hat automatisch den korrekten Typ. Die Typprüfung erfolgt zur Übersetzungszeit, verursacht also zur Laufzeit keine Unkosten. Das folgende Beispiel zeigt die Verwendung von `.this`:

```
//: innerclasses/DotThis.java
// Qualifying access to the outer-class object.

public class DotThis {
    void f() { System.out.println("DotThis.f()"); }
    public class Inner {
        public DotThis outer() {
            return DotThis.this;
            // A plain "this" would be Inner's "this"
        }
    }
    public Inner inner() { return new Inner(); }
```

```
    public static void main(String[] args) {
        DotThis dt = new DotThis();
        DotThis.Inner dti = dt.inner();
        dti.outer().f();
    }
} /* Output:
      DotThis.f()
   *///:~
```

[15] Gelegentlich möchten Sie, daß ein Objekt einer äußeren Klasse ein Objekt einer seiner inneren Klassen direkt erzeugt (im Gegensatz zu einer Hilfsmethode, die ein Objekt erzeugt und die Objektreferenz zurückgibt). Sie müssen hierzu beim Aufruf des `new`-Operators, per `.new`-Syntax die Referenz auf das Objekt der äußeren Klasse übergeben, zum Beispiel:

```
//: innerclasses/DotNew.java
// Creating an inner class directly using the .new syntax.

public class DotNew {
    public class Inner {}
    public static void main(String[] args) {
        DotNew dn = new DotNew();
        DotNew.Inner dni = dn.new Inner();
    }
} ///:~
```

[16] Beim direkten Erzeugen eines Objektes einer inneren Klasse, beziehen Sie sich *nicht auf den Namen* (wie Sie nach dem vorigen Beispiel *DotThis.java* eventuell erwarten), sondern auf ein *Objekt* der äußeren Klasse. Damit sind auch die Namensraumprobleme hinsichtlich der inneren Klasse gelöst. Die Syntax `dn.new DotNew.Inner()` ist *nicht* erlaubt.

[17] Es ist nicht möglich, ein Objekt einer inneren Klasse zu erzeugen, wenn nicht bereits ein Objekt der äußeren Klasse existiert, da zwischen beiden eine stillschweigende Verbindung besteht. Eine *geschachtelte Klasse* (eine statische innere Klasse, siehe Abschnitt 11.7) benötigt dagegen keine Referenz auf ein Objekt der äußeren Klasse.

[18] Das folgende Beispiel zeigt die Anwendung der `.new`-Syntax im Paket-Beispiel aus Abschnitt 11.1:

```
//: innerclasses/Parcel3.java
// Using .new to create instances of inner classes.

public class Parcel3 {
    class Contents {
        private int i = 11;
        public int value() { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) { label = whereTo; }
        String readLabel() { return label; }
    }
    public static void main(String[] args) {
        Parcel3 p = new Parcel3();
        // Must use instance of outer class
        // to create an instance of the inner class:
        Parcel3.Contents c = p.new Contents();
        Parcel3.Destination d = p.new Destination("Tasmania");
    }
} ///:~
```

Übungsaufgabe 4: (2) Legen Sie in der inneren Klasse `Sequence.SequenceSelector` eine Methode an, welche die Referenz auf das äußere `Sequence`-Objekt liefert. ■

Übungsaufgabe 5: (1) Schreiben Sie eine Klasse, die eine innere Klasse enthält. Erzeugen Sie in einer separaten Klasse ein Objekt der inneren Klasse. ■

11.4 Referenzen auf Objekte innerer Klassen vom Typ einer Basisklasse oder eines Interfaces

[19] Innere Klassen kommen erst voll zur Geltung, wenn Sie die Referenz auf ein Objekt einer inneren Klasse in den Typ einer Basisklasse, speziell eines Interfaces umwandeln. (Die Umwandlung einer Objektreferenz in den Typ des implementierten Interfaces entspricht effektiv im wesentlichen der Umwandlung in den Typ der Basisklasse.) Die innere Klasse, also die Implementierung des Interfaces, kann in diesem Fall nämlich unsichtbar und unerreichbar bleiben, ein komfortabler Ansatz, um die Implementierung zu verbergen. Sie bekommen lediglich ein Objektreferenz vom Typ der Basisklasse beziehungsweise des Interfaces.

[20–21] Wir definieren die beiden folgenden Interfaces für das Paket-Beispiel:

```
//: innerclasses/Destination.java
public interface Destination {
    String readLabel();
} ///:~
```

```
//: innerclasses/Contents.java
public interface Contents {
    int value();
} ///:~
```

`Destination` und *`Contents`* sind für den Clientprogrammierer zugängliche Interfaces. Denken Sie daran, daß die Komponenten eines Interfaces implizit öffentlich sind.

[22] Wenn Sie (nur) eine Referenz eines Basisklassen- oder Interfacestyps zur Verfügung haben, können Sie eventuell den exakten Typ nicht herausfinden:

```
//: innerclasses/TestParcel.java
class Parcel4 {
    private class PContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected class PDestination implements Destination {
        private String label;
        private PDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
    }
    public Destination destination(String s) {
        return new PDestination(s);
    }
    public Contents contents() {
        return new PContents();
    }
}
```

```
public class TestParcel {
    public static void main(String[] args) {
        Parcel4 p = new Parcel4();
        Contents c = p.contents();
        Destination d = p.destination("Tasmania");
        // Illegal - can't access private class:
        //! Parcel4.PContents pc = p.new PContents();
    }
} ///:~
```

[23] Die Klasse `Parcel4` zeigt eine weitere Eigenschaft innerer Klassen: Die innere Klasse `PContents` ist als `private` definiert, so daß nur die Komponenten von `Parcel4` Zugriff erhalten. Normale (nicht-innere) Klassen können dagegen nicht mit den Modifikatoren `private` oder `protected` definiert werden, sondern nur unter öffentlichem Zugriff oder Packagezugriff stehen. Die zweite innere Klasse, `PDestination`, ist als `protected` deklariert, so daß nur die Komponenten von `Parcel4`, Klassen im gleichen Package (`protected` schließt Packagezugriff ein) und von `Parcel4` abgeleitete Klassen Zugriff auf `PDestination` erhalten. Der Clientprogrammierer verfügt somit nur über begrenzte Informationen und eingeschränkten Zugriff über beziehungsweise auf die Komponenten von `PDestination`. Sie können eine Referenz auf ein Objekt einer als `private` deklarierten inneren Klasse nicht abwärts in ihren eigentlichen Typ umwandeln, wie Sie in der `main()`-Methode der Klasse `TestParcel` sehen (analog bei einer als `protected` deklarierten inneren Klasse, außer in einer von der äußeren Klasse abgeleiteten Klasse). Eine als `private` deklarierte innere Klasse stellt eine Möglichkeit dar, über die der Designer einer Klasse, Typabhängigkeiten im clientseitigen Quelltext sicher verhindern und Implementierungsdetails vollständig verbergen kann. Die Erweiterung des Interfaces ist aus der Perspektive des Clientprogrammierers nutzlos, da er keinen Zugriff auf Methoden erhält, die nicht im öffentlichen Interface deklariert sind. Darüber hinaus gestattet dieser Ansatz dem Java-Compiler, effizienteren Code zu generieren.

Übungsaufgabe 6: (2) Definieren Sie ein Interface mit wenigstens einer Methode in einem eigenen Package. Schreiben Sie eine Klasse, die einem anderen Package angehört. Legen Sie eine als `protected` definierte innere Klasse an, die das Interface implementiert. Leiten Sie in einem dritten Package eine Klasse von Ihrer Klasse ab und geben Sie über eine Methode eine in den Typ des Interfaces umgewandelte Referenz auf ein Objekt der geschützten inneren Klasse zurück. ■

Übungsaufgabe 7: (2) Schreiben Sie eine Klasse mit einem privaten Feld und einer privaten Methode. Definieren Sie eine innere Klasse mit einer Methode, die den Inhalt des Feldes der äußeren Klasse modifiziert und die Methode der äußeren Klasse aufruft. Erzeugen Sie in einer zweiten Methode der äußeren Klasse ein Objekt der inneren Klasse, rufen Sie seine Methode auf und zeigen Sie die Wirkung auf das Objekt der äußeren Klasse. ■

Übungsaufgabe 8: (2) Ermitteln Sie, ob die äußere Klasse Zugriff auf private Komponenten einer inneren Klasse hat. ■

11.5 Innere Klassen in Methoden und beliebigen Geltungsbereichen

[24] Die bisherigen Beispiele decken die typische Verwendung innerer Klassen ab. In der Regel sind die Implementierungen innerer Klassen die Sie vorfinden und selbst schreiben werden einfach gehalten und mühelos zu verstehen. Die Syntax der inneren Klassen gestattet allerdings auch eine Reihe anderer und etwas undurchsichtigerer Ansätze. Innere Klasse können in Methodenkörpern und sogar in beliebigen Geltungsbereichen definiert werden. Es gibt zwei Gründe, um von diesen

Möglichkeiten Gebrauch zu machen:

- Sie implementieren ein Interface, wie im vorigen Abschnitt, damit Sie ein Objekt einer inneren Klasse erzeugen und die Referenz auf dieses Objekt zurückgegeben können.
- Sie lösen ein kompliziertes Problem und brauchen eine Hilfsklasse, die Sie nicht öffentlich zur Verfügung stellen möchten.

In den folgenden Beispielen wird das Paket-Beispiel verändert, um die folgenden Varianten vorzustellen:

- Innere Klasse im Körper einer Methode (lokale innere Klasse): *Parcel5.java* auf Seite 283.
- Innere Klasse in einem Geltungsbereich innerhalb eines Methodenkörpers: *Parcel6.java* auf Seite 284.
- Anonyme innere Klasse implementiert ein Interface: *Parcel7.java* auf Seite 284.
- Anonyme innere Klasse von einer Klasse abgeleitet, die einen parameterbehafteten Konstruktor (Nicht-Standardkonstruktor) besitzt: *Parcel8.java* auf Seite 285.
- Anonyme innere Klasse zur Initialisierung von Feldern: *Parcel9.java* auf Seite 286.
- Anonyme innere Klasse mit dynamischem Initialisierungsblock (anonyme innere Klasse können keine Konstruktoren haben): *Parcel10.java* auf Seite 287.

[25–26] Das erste Beispiel zeigt die Definition einer ganzen Klasse im Geltungsbereich einer Methode (statt im Geltungsbereich einer äußeren Klasse), einer sogenannten **lokalen inneren Klasse**:

```
//: innerclasses/Parcel5.java
// Nesting a class within a method.

public class Parcel5 {
    public Destination destination(String s) {
        class PDestination implements Destination {
            private String label;
            private PDestination(String whereTo) {
                label = whereTo;
            }
            public String readLabel() { return label; }
        }
        return new PDestination(s);
    }
    public static void main(String[] args) {
        Parcel5 p = new Parcel5();
        Destination d = p.destination("Tasmania");
    }
} ///:~
```

Die innere Klasse `PDestination` ist nun Teil der Methode `destination()`, statt der äußeren Klasse `Parcel5`. Folglich ist `PDestination` außerhalb des Methodenkörpers von `destination()` nicht definiert. Beachten Sie die Umwandlung der von `destination()` zurückgegebenen Referenz in den Typ *`Destination`* (Interface). Die Platzierung der Klasse `PDestination` im Körper der `destination()`-Methode bedeutet keineswegs, daß das `PDestination`-Objekt nach der Rückkehr aus `destination()` nicht mehr gültig ist.

[27] Sie können den Bezeichner `PDestination` in jeder Klasse innerhalb eines Unterverzeichnisses einmal für eine innere Klasse vergeben, ohne eine Namenskollision zu verursachen.

[28–29] Das nächste Beispiel zeigt, daß Sie in jedem beliebigen Geltungsbereich eine innere Klasse definieren können:

```
//: innerclasses/Parcel6.java
// Nesting a class within a scope.

public class Parcel6 {
    private void internalTracking(boolean b) {
        if(b) {
            class TrackingSlip {
                private String id;
                TrackingSlip(String s) {
                    id = s;
                }
                String getSlip() { return id; }
            }
            TrackingSlip ts = new TrackingSlip("slip");
            String s = ts.getSlip();
        }
        // Can't use it here! Out of scope:
        //! TrackingSlip ts = new TrackingSlip("x");
    }
    public void track() { internalTracking(true); }
    public static void main(String[] args) {
        Parcel6 p = new Parcel6();
        p.track();
    }
} ///:~
```

Die innere Klasse `TrackingSlip` ist im Geltungsbereich einer `if`-Anweisung definiert. Dies bedeutet *nicht*, daß die Klasse in Abhängigkeit von einer Bedingung *übersetzt* wird (der Compiler übersetzt den gesamten Quelltext). Allerdings ist die Klasse außerhalb des Geltungsbereiches in dem sie definiert ist, nicht vorhanden. Davon abgesehen, ist `TrackingSlip` eine gewöhnliche Klasse.

Übungsaufgabe 9: (1) Definieren Sie ein Interface mit wenigstens einer Methode und implementieren Sie das Interface mittels einer lokalen inneren Klasse in einer Methode, die eine Referenz vom Typ Ihres Interfaces zurückgibt. ■

Übungsaufgabe 10: (1) Wiederholen Sie Übungsaufgabe 9. Definieren Sie die innere Klasse aber diesesmal in einem Geltungsbereich innerhalb des Methodenkörpers. ■

Übungsaufgabe 11: (2) Definieren Sie eine private innere Klasse, die ein öffentliches Interface implementiert. Legen Sie eine Methode an, welche die Referenz auf ein Objekt der privaten inneren Klasse, umgewandelt in den Typ des Interfaces zurückgibt. Zeigen Sie, durch Versuchen einer abwärts gerichteten Typumwandlung, daß die innere Klasse vollständig verborgen ist. ■

11.6 Anonyme innere Klassen

[30–31] Das folgende Beispiel wirkt ein wenig sonderbar:

```
//: innerclasses/Parcel7.java
// Returning an instance of an anonymous inner class.

public class Parcel7 {
    public Contents contents() {
        return new Contents() { // Insert a class definition
```

```
                private int i = 11;
                public int value() { return i; }
            }; // Semicolon required in this case
    }
    public static void main(String[] args) {
        Parcel7 p = new Parcel7();
        Contents c = p.contents();
    }
} ///:~
```

Die Methode `contents()` kombiniert das Erzeugen des Rückgabewertes mit der Definition einer Klasse, deren Objekte diesen Rückgabewert darstellen. Darüber hinaus ist die Klasse *anonym* (hat keinen Namen). Außerdem scheint die Methode ein *`Contents`*-Objekt (Interface!) zu erzeugen. Aber an der Stelle, an der üblicherweise das Semikolon steht, ist hier eine Klassendefinition „zwischengeschaltet“.

[32–35] Diese Notation erzeugt ein Objekt einer anonymen inneren Klasse, die von *`Contents`* abgeleitet ist beziehungsweise `Contents` implementiert. Die bei der Objekterzeugung zurückgegebene Referenz wird automatisch in den Typ *`Contents`* umgewandelt. Die anonyme innere Klasse ist eine abgekürzte Schreibweise für:

```
//: innerclasses/Parcel7b.java
// Expanded version of Parcel7.java

public class Parcel7b {
    class MyContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    public Contents contents() { return new MyContents(); }
    public static void main(String[] args) {
        Parcel7b p = new Parcel7b();
        Contents c = p.contents();
    }
} ///:~
```

Ist der Basistyp `Contents` eine Klasse, so wird dessen Unterobjekt mit Hilfe des Standardkonstruktors erzeugt. Das nächste Beispiel zeigt, wie Sie vorgehen müssen, um einen parameterbehafteten Konstruktor der Basisklasse aufzurufen:

```
//: innerclasses/Parcel8.java
// Calling the base-class constructor.

public class Parcel8 {
    public Wrapping wrapping(int x) {
        // Base constructor call:
        return new Wrapping(x) { // Pass constructor argument.
                public int value() {
                    return super.value() * 47;
                }
            }; // Semicolon required
    }
    public static void main(String[] args) {
        Parcel8 p = new Parcel8();
        Wrapping w = p.wrapping(10);
    }
} ///:~
```

Sie übergeben dem Konstruktor der Basisklasse also einfach das passende Argument, wie `x` bei `new Wrapping(x)`. Die Basisklasse `Wrapping` ist eine gewöhnliche Klasse, definiert aber in diesem Fall auch die Schnittstelle für die von ihr abgeleiteten Klassen:

```
//: innerclasses/Wrapping.java
public class Wrapping {
    private int i;
    public Wrapping(int x) { i = x; }
    public int value() { return i; }
} ///:~
```

Beachten Sie, daß die Klasse `Wrapping` einen Konstruktor mit Parameter definiert, um das Beispiel etwas interessanter zu machen.

[36] Das Semikolon am Ende der anonymen inneren Klasse kennzeichnet nicht das Ende des Klassenkörpers, sondern das Ende des Ausdrucks, der die anonyme Klasse beinhaltet, unterscheidet sich also nicht von den übrigen Semikolons.

[37–38] Eine anonyme innere Klasse kann Felder definieren und initialisieren:

```
//: innerclasses/Parcel9.java
// An anonymous inner class that performs
// initialization. A briefer version of Parcel5.java.

public class Parcel9 {
    // Argument must be final to use inside
    // anonymous inner class:
    public Destination destination(final String dest) {
        return new Destination() {
                private String label = dest;
                public String readLabel() { return label; }
            };
    }
    public static void main(String[] args) {
        Parcel9 p = new Parcel9();
        Destination d = p.destination("Tasmania");
    }
} ///:~
```

Verwendet eine anonyme innere Klasse ein Objekt einer außerhalb dieser inneren Klasse definierten Klasse, so verlangt der Compiler, daß die entsprechende Referenzvariable als final definiert wird (siehe Argument der Methode `destination()`). Vergessen Sie `final`, so meldet der Compiler einen Fehler.

[39–40] Der Ansatz im vorigen Beispiel genügt, wenn Sie nur Felder zu initialisieren brauchen. Was aber, wenn Sie darüber hinausgehendes Konstruktorverhalten benötigen? Sie können in einer anonymen inneren Klasse keinen benannten Konstruktor anlegen, da die Klasse keinen Namen hat. Sie können aber mit Hilfe eines *dynamischen Initialisierungsblocks* (*instance initializer*) konstruktorähnliches Verhalten definieren:

```
//: innerclasses/AnonymousConstructor.java
// Creating a constructor for an anonymous inner class.
import static net.mindview.util.Print.*;

abstract class Base {
    public Base(int i) {
        print("Base constructor, i = " + i);
    }
    public abstract void f();
```

```
}

public class AnonymousConstructor {
    public static Base getBase(int i) {
        return new Base(i) {
                { print("Inside instance initializer"); }
                public void f() {
                    print("In anonymous f()");
                }
            };
    }
    public static void main(String[] args) {
        Base base = getBase(47);
        base.f();
    }
} /* Output:
      Base constructor, i = 47
      Inside instance initializer
      In anonymous f()
   *///:~
```

Die lokale Variable `i` braucht in diesem Fall nicht final zu sein. Der Konstruktor der Basisklasse der anonymen Klasse erhält zwar ein Argument, aber es wird innerhalb der anonymen Klasse nicht direkt verwendet.

[41–42] Das folgende Beispiel zeigt nochmals das Paket-Beispiel, diesesmal mit einem dynamischen Initialisierungsblock. Beachten Sie, daß die Parameter der `destination()`-Methode wiederum finale Parameter sein müssen, da sie in der anonymen inneren Klasse verwendet werden:

```
//: innerclasses/Parcel10.java
// Using "instance initialization" to perform
// construction on an anonymous inner class.

public class Parcel10 {
    public Destination
        destination(final String dest, final float price) {
        return new Destination() {
                private int cost;
                // Instance initialization for each object:
                {
                    cost = Math.round(price);
                    if(cost > 100)
                        System.out.println("Over budget!");
                }
                private String label = dest;
                public String readLabel() { return label; }
            };
    }
    public static void main(String[] args) {
        Parcel10 p = new Parcel10();
        Destination d = p.destination("Tasmania", 101.395F);
    }
} /* Output:
      Over budget!
   *///:~
```

Der dynamische Initialisierungsblock enthält eine `if`-Anweisung, die nicht als Teil eines Feldinitialisierungsausdrucks verarbeitet werden kann. Der dynamische Initialisierungsblock ist somit effektiv der Konstruktor der obigen anonymen inneren Klasse. Ihre Möglichkeiten sind natürlich begrenzt.

Ein dynamischer Initialisierungsblock kann nicht überladen werden, so daß Sie nur einen solchen Konstruktor zur Verfügung haben.

[43] Anonyme innere Klassen sind auch hinsichtlich der normalen Ableitung eingeschränkt, da sie entweder von einer Klasse abgeleitet werden oder ein Interface implementieren können, aber nicht beides zugleich. Eine anonyme Klasse kann höchstens ein Interface implementieren.

Übungsaufgabe 12: (1) Wiederholen Sie Übungsaufgabe 7 (Seite 282) mit einer anonymen inneren Klasse. ■

Übungsaufgabe 13: (1) Wiederholen Sie Übungsaufgabe 9 (Seite 284) mit einer anonymen inneren Klasse. ■

Übungsaufgabe 14: (1) Ändern Sie das Beispiel *interfaces/HorrorShow.java* (Abschnitt 10.5, Seite 263), so daß die beiden Interfaces `DangerousMonster` und `Vampire` als anonyme Klassen implementiert werden. ■

Übungsaufgabe 15: (2) Schreiben Sie eine Klasse mit einem Nicht-Standardkonstruktor (Konstruktor mit Parametern) und einem Standardkonstruktor. Schreiben Sie eine zweite Klasse mit einer Methode, die die Referenz auf ein Objekt der ersten Klasse zurückgibt. Definieren Sie das zurückgegebene Objekt in Form einer anonymen inneren Klasse, die von der ersten Klasse abgeleitet ist. ■

11.6.1 Überarbeitung der Fabrikmethoden-Beispiele aus Abschnitt 10.9

[44–45] Sehen Sie sich an, wie viel schöner sich das Beispiel *interfaces/Factories.java* aus Abschnitt 10.9 mit Hilfe anonymer innerer Klassen implementieren läßt:

```
//: innerclasses/Factories.java
import static net.mindview.util.Print.*;

interface Service {
    void method1();
    void method2();
}

interface ServiceFactory {
    Service getService();
}

class Implementation1 implements Service {
    private Implementation1() {}
    public void method1() { print("Implementation1 method1"); }
    public void method2() { print("Implementation1 method2"); }
    public static ServiceFactory factory =
        new ServiceFactory() {
            public Service getService() {
                return new Implementation1();
            }
        };
}

class Implementation2 implements Service {
    private Implementation2() {}
    public void method1() { print("Implementation2 method1"); }
    public void method2() { print("Implementation2 method2"); }
    public static ServiceFactory factory =
        new ServiceFactory() {
```

```
            public Service getService() {
                return new Implementation2();
            }
        };
}

public class Factories {
    public static void serviceConsumer(ServiceFactory fact) {
        Service s = fact.getService();
        s.method1();
        s.method2();
    }
    public static void main(String[] args) {
        serviceConsumer(Implementation1.factory);
        // Implementations are completely interchangeable:
        serviceConsumer(Implementation2.factory);
    }
} /* Output:
      Implementation1 method1
      Implementation1 method2
      Implementation2 method1
      Implementation2 method2
   *///:~
```

Die Konstruktoren der Klassen `Implementation1` und `Implementation2` können nun als privat definiert werden und es ist nicht mehr nötig, die Fabrikklasse als benannte Klasse anzulegen. Häufig genügt ein einziges Fabrikobjekt. Daher hat jede der beiden Implementierungen des Interfaces *`Service`* ein statisches Feld, welches ein Fabrikobjekt vom Typ *`ServiceFactory`* referenziert. Insgesamt ergibt sich eine aussagekräftigere Syntax.

[46–47] Auch das Beispiel *interfaces/Games.java*, ebenfalls aus Abschnitt 10.9, läßt sich mit anonymen inneren Klasse verbessern:

```
//: innerclasses/Games.java
// Using anonymous inner classes with the Game framework.
import static net.mindview.util.Print.*;

interface Game { boolean move(); }
interface GameFactory { Game getGame(); }

class Checkers implements Game {
    private Checkers() {}
    private int moves = 0;
    private static final int MOVES = 3;
    public boolean move() {
        print("Checkers move " + moves);
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
            public Game getGame() { return new Checkers(); }
        };
}

class Chess implements Game {
    private Chess() {}
    private int moves = 0;
    private static final int MOVES = 4;
    public boolean move() {
        print("Chess move " + moves);
```

```
        return ++moves != MOVES;
    }
    public static GameFactory factory = new GameFactory() {
            public Game getGame() { return new Chess(); }
        };
}

public class Games {
    public static void playGame(GameFactory factory) {
        Game s = factory.getGame();
        while(s.move())
            ;
    }
    public static void main(String[] args) {
        playGame(Checkers.factory);
        playGame(Chess.factory);
    }
} /* Output:
      Checkers move 0
      Checkers move 1
      Checkers move 2
      Chess move 0
      Chess move 1
      Chess move 2
      Chess move 3
   *///:~
```

Denken Sie an den Rat vom Ende des letzten Kapitels: *Geben Sie Klassen den Vorzug vor Interfaces.* Sie erkennen, wenn Ihr Design ein Interface verlangt. Vermeiden Sie andernfalls Interfaces, bis Sie gezwungen sind, eines zu verwenden.

Übungsaufgabe 16: (1) Ändern Sie Übungsaufgabe 18 aus Abschnitt 10.9 (Seite 273), so daß die Lösung anonyme innere Klassen verwendet. ■

Übungsaufgabe 17: (1) Ändern Sie Übungsaufgabe 19 aus Abschnitt 10.9 (Seite 273), so daß die Lösung anonyme innere Klassen verwendet. ■

11.7 Geschachtelte Klassen (statische innere Klassen)

[48] Wenn Sie die Verbindung zwischen dem Objekt einer inneren Klasse und dem Objekt der äußeren Klasse nicht brauchen, können Sie die innere Klasse als statisch definieren. Diese Sorte innerer Klassen wird im allgemeinen als *geschachtelte Klasse* (*nested class*)[2] bezeichnet. Um die Bedeutung des Modifikators `static` zu verstehen, müssen Sie daran denken, daß ein Objekt einer gewöhnlichen inneren Klasse implizit eine Referenz auf das Objekt der äußeren Klasse speichert, über das es erzeugt wurde. Dies gilt nicht, wenn Sie eine innere Klasse als statisch deklarieren. Für eine geschachtelte Klasse gilt:

- Sie brauchen kein Objekt der äußeren Klasse, um ein Objekt einer geschachtelten Klasse zu erzeugen.
- Eine geschachtelte Klasse hat keinen Zugriff auf die Komponenten eines nicht-statischen äußeren Objektes.

[2] Annähernd wie die geschachtelten Klassen bei C++, die allerdings, im Gegensatz zu Java, keine privaten Komponenten ihrer äußeren Klasse erreichen können.

[49–50] Geschachtelte Klassen unterscheiden sich in noch einer weiteren Hinsicht von gewöhnlichen inneren Klassen: Die Komponenten einer inneren Klasse können nur dann als statisch deklariert werden, wenn auch die äußere Klasse statisch ist. Geschachtelte Klassen können dagegen sowohl statische als auch nicht-statische Komponenten haben:

```
//: innerclasses/Parcel11.java
// Nested classes (static inner classes).

public class Parcel11 {
    private static class ParcelContents implements Contents {
        private int i = 11;
        public int value() { return i; }
    }
    protected static class ParcelDestination implements Destination {
        private String label;
        private ParcelDestination(String whereTo) {
            label = whereTo;
        }
        public String readLabel() { return label; }
        // Nested classes can contain other static elements:
        public static void f() {}
        static int x = 10;
        static class AnotherLevel {
            public static void f() {}
            static int x = 10;
        }
    }
    public static Destination destination(String s) {
        return new ParcelDestination(s);
    }
    public static Contents contents() {
        return new ParcelContents();
    }
    public static void main(String[] args) {
        Contents c = contents();
        Destination d = destination("Tasmania");
    }
} ///:~
```

Die `main()`-Methode braucht kein `Parcel11`-Objekt, sondern verwendet die für statische Komponenten übliche Syntax, um die Methoden aufzurufen, welche die Referenzen auf die *`Contents`*- und *`Destination`*-Objekte zurückgeben.

[51] In einer normalen (nicht-statischen) inneren Klasse ist die Referenz auf das Objekt der äußeren Klasse über die `.this`-Syntax verfügbar (siehe Abschnitt 11.3). Eine geschachtelte innere Klasse hat die `.this`-Syntax dagegen nicht zur Verfügung, analog zu einer statischen Methode (Fehlermeldung des Compilers: „non-static variable `this` cannot be referenced from a static context“).

Übungsaufgabe 18: (1) Schreiben Sie eine Klasse, die eine geschachtelte Klasse enthält und erzeugen Sie in `main()`-Methode ein Objekt der geschachtelten Klasse. ■

Übungsaufgabe 19: (2) Schreiben Sie eine Klasse, die eine innere Klasse enthält, welche wiederum eine innere Klasse definiert. Wiederholen Sie dies mit geschachtelten Klassen. Beachten Sie die Namen der vom Compiler erzeugten Klassendateien(*.class* Dateien). ■

11.7.1 Geschachtelte Klassen in Interfaces

[52–53] Normalerweise können Sie keine Anweisungen in ein Interface einsetzen. Es ist aber erlaubt in einem Interface eine geschachtelte Klasse zu definieren. Jedes Element eines Interfaces, also auch jede Klasse, ist automatisch öffentlich und statisch. Da eine geschachtelte Klasse statisch ist, verletzt sie die Regeln für Interfaces nicht, sondern wird nur dem durch das Interface definierten Namensraum zugeordnet. Sie können das umgebende Interface sogar in der geschachtelten Klasse implementieren:

```
//: innerclasses/ClassInInterface.java
// {main: ClassInInterface$Test}

public interface ClassInInterface {
    void howdy();
    class Test implements ClassInInterface {
        public void howdy() {
            System.out.println("Howdy!");
        }
        public static void main(String[] args) {
            new Test().howdy();
        }
    }
} /* Output:
      Howdy!
   *///:~
```

Die Definition einer geschachtelten Klasse in einem Interface ist komfortabel, um eine allgemeine Funktionalität zu hinterlegen, die von allen Implementierungen des Interfaces verwendet wird.

[54–55] In Abschnitt 8.2 habe ich auf Seite 196 angeregt, in jeder Klasse eine `main()`-Methode als Testumgebung für diese Klasse anzulegen. Ein Nachteil dieser Vorgehensweise ist das Ausmaß an zusätzlich übersetztem Code *pro Objekt*. Ist diese Situation problematisch, so können Sie die zum Testen benötigten Anweisungen in einer geschachtelten Klasse unterbringen:

```
//: innerclasses/TestBed.java
// Putting test code in a nested class.
// {main: TestBed$Tester}

public class TestBed {
    public void f() { System.out.println("f()"); }
    public static class Tester {
        public static void main(String[] args) {
            TestBed t = new TestBed();
            t.f();
        }
    }
} /* Output:
      f()
   *///:~
```

Sie erhalten eine separate Klasse namens `TestBed$Tester` (das Programm wird per `java TestBed$Tester` aufgerufen; beachten Sie, daß Sie das Dollarzeichen unter Unix/Linux schützen müssen). Sie können diese Hilfsklasse zum Testen verwenden, brauchen sie aber nicht zusammen mit dem Softwareprodukt auszuliefern, sondern können *TestBed$Tester.class* einfach vor dem Zusammenstellen der Distribution löschen.

Übungsaufgabe 20: (1) Definieren Sie ein Interface mit einer geschachtelten Klasse. Implementieren Sie das Interface und erzeugen Sie ein Objekt der geschachtelten Klasse. ■

Übungsaufgabe 21: (2) Definieren Sie ein Interface mit einer geschachtelten Klasse, die eine statische Methode beinhaltet. Die statische Methode ruft die im Interface deklarierten Methoden auf und zeigt die Ergebnisse an. Implementieren Sie das Interface und übergeben Sie der statischen Methode ein Objekt Ihrer Implementierung. ■

11.7.2 Erreichbarkeit bei mehrfach geschachtelten inneren Klassen

[56–57] Gleichgültig wie tief eine nicht-statische innere Klasse verschachtelt ist, hat sie stets transparenten Zugriff die Komponenten aller Klassen der Schachtelungsreihenfolge:[3]

```
//: innerclasses/MultiNestingAccess.java
// Nested classes can access all members of all
// levels of the classes they are nested within.

class MNA {
    private void f() {}
    class A {
        private void g() {}
        public class B {
            void h() {
                g();
                f();
            }
        }
    }
}

public class MultiNestingAccess {
    public static void main(String[] args) {
        MNA mna = new MNA();
        MNA.A mnaa = mna.new A();
        MNA.A.B mnaab = mnaa.new B();
        mnaab.h();
    }
} ///:~
```

Sie sehen, daß die Methoden `g()` und `f()` innerhalb der Klasse `MNA.A.B` ohne Qualifizierung aufgerufen werden können (abgesehen davon, daß es private Methoden sind). Das Beispiel zeigt außerdem die benötigte Syntax zum Erzeugen von Objekten mehrfach verschachtelter innerer Klassen in einer außenstehenden Klasse. Die `.new`-Syntax liefert den korrekten Geltungsbereich, so daß der Klassenname beim Aufruf des Konstruktors nicht qualifiziert werden muß.

11.8 Motivation zur Anwendung innerer Klassen

[58] Sie haben die Syntax und Semantik der inneren Klassen bis zu diesem Abschnitt an einer Reihe von Beispielen kennengelernt. Die Frage, zu welchem Zweck dieses Konzept existiert, wird durch diese Beispiele jedoch nicht beantwortet. Warum haben die Designer von Java soviel Mühe auf sich genommen, um diese fundamentale Eigenschaft in den Sprachumfang aufzunehmen.

[59] Ein innere Klasse ist typischerweise von einer anderen Klasse abgeleitet oder implementiert ein Interface. Die Anweisungen der inneren Klasse bedienen die Komponenten der äußeren Klasse, in der die innere Klasse definiert ist. Eine innere Klasse ist also eine Art Fenster zur äußeren Klasse.

[3] Nochmals meinen Dank an Martin Danner.

[60] Fundamentale Frage: Angenommen, Sie brauchen lediglich eine Referenz vom Typ eines Interfaces. Genügt es dann nicht, wenn die äußere Klasse dieses Interface implementiert? Antwort: Wenn Ihre Anforderungen damit erfüllt sind, sollten Sie diesen Weg wählen. Worin besteht also der Unterschied zwischen einer inneren Klasse, die ein Interface implementiert und einer äußeren Klasse, die dasselbe Interface implementiert? Die Antwort lautet, daß Sie nicht immer den Komfort von Interfaces in Anspruch nehmen können, sondern gelegentlich mit Implementierungen arbeiten müssen. Der wichtigste Grund, sich für innere Klassen zu entscheiden, besteht darin, daß jede innere Klasse *unabhängig von der äußeren Klasse* von einer Implementierung abgeleitet werden kann. Eine innere Klasse wird also nicht dadurch eingeschränkt, wenn ihre äußere Klasse bereits von einer Implementierung abgeleitet ist.

[61] Ohne die Möglichkeit, mehrere innere Klassen von unterschiedlichen konkreten oder abstrakten Klassen ableiten zu können, wären einige Design- und Programmierprobleme schwer zu lösen. Innere Klassen können als Rest der Lösung des Problems der Mehrfachvererbung betrachtet werden. Ein Teil dieses Problems wird durch Interfaces gelöst, aber erst die inneren Klassen gestatten effektiv die Mehrfachvererbung von Implementierungen, also *das Ableiten von mehr als einer Klasse*.

[62–63] Das folgende Beispiel definiert zwei Interfaces, die von einer Klasse implementiert werden sollen. Die Aufgabe, zwei *Interfaces* zu implementieren (im Gegensatz zur Ableitung von zwei verschiedenen Klassen, siehe unten) gestattet zwei Wahlmöglichkeiten: Eine Klasse implementiert beide Interfaces oder die Implementierung wird auf eine äußere und eine innere Klasse verteilt:

```
//: innerclasses/MultiInterfaces.java
// Two ways that a class can implement multiple interfaces.
package innerclasses;

interface A {}
interface B {}

class X implements A, B {}

class Y implements A {
    B makeB() {
        // Anonymous inner class:
        return new B() {};
    }
}

public class MultiInterfaces {
    static void takesA(A a) {}
    static void takesB(B b) {}
    public static void main(String[] args) {
        X x = new X();
        Y y = new Y();
        takesA(x);
        takesA(y);
        takesB(x);
        takesB(y.makeB());
    }
} ///:~
```

Beide Vorgehensweisen setzen natürlich voraus, daß die Strukturierung des Quelltextes logisch schlüssig ist. In der Regel liefern aber die Eigenschaften des Problems Anhaltspunkte für die Entscheidung, ob Sie eine einzelne oder eine innere Klasse verwenden sollten. Ohne weitere Einschränkungen unterscheiden sich die beiden Ansätze im obigen Beispiel kaum und beide funktionieren.

[64] Ist dagegen die Ableitung von abstrakten oder konkreten Klassen, statt der Implementierung von Interfaces notwendig, so müssen Sie auf innere Klassen zurückgreifen, falls Ihre Klasse von mehr

als einer Klasse abgeleitet werden muß:

```
//: innerclasses/MultiImplementation.java
// With concrete or abstract classes, inner
// classes are the only way to produce the effect
// of ''multiple implementation inheritance.''
package innerclasses;

class D {}
abstract class E {}

class Z extends D {
    E makeE() { return new E() {}; }
}

public class MultiImplementation {
    static void takesD(D d) {}
    static void takesE(E e) {}
    public static void main(String[] args) {
        Z z = new Z();
        takesD(z);
        takesE(z.makeE());
    }
} ///:~
```

[65] Sofern Sie nicht gezwungen sind, „eine Klasse von mehr als einer anderen Klasse abzuleiten", können Sie die Notwendigkeit innerer Klassen unter Umständen vermeiden. Ein Ansatz mit inneren Klassen bietet aber die folgenden Eigenschaften und Fähigkeiten:

- Sie können mehrere Objekt einer inneren Klasse erzeugen, von denen jedes einzelne einen individuellen, vom Objekt der äußeren Klasse unabhängigen Zustand hat.
- Ein äußere Klasse kann mehrere innere Klassen beinhalten, von denen jede dasselbe Interface auf individuelle Weise implementiert beziehungsweise auf individuelle Weise von derselben Klasse abgeleitet ist. Der folgende Abschnitt beschreibt ein Beispiel.
- Die Stelle an der ein Objekt einer inneren Klasse erzeugt wird, ist unabhängig von der Stelle an das Objekt der äußeren Klasse erzeugt wird.
- Es gibt keine potentiell verwirrende „ist-ein"-Beziehung. Eine innere Klasse definiert eine separate Entität.

[66] Ohne innere Klassen, müßte die Klasse `Sequence` im Beispiel *Sequence.java* auf Seite 277 das Interface *`Selector`* implementieren und wäre an diesen Selektor gebunden. Mit inneren Klassen können Sie dagegen einfach ein zweite Methode anlegen, zum Beispiel `reverseSelector()`, die einen Selektor rückwärts durch die Folge bewegt. Diese Flexibilität bekommen Sie nur mit inneren Klassen.

Übungsaufgabe 22: (2) Implementieren Sie die `reverseSelector()`-Methode im Beispiel *Sequence.java* (Seite 277). ■

Übungsaufgabe 23: (4) Definieren Sie ein Interface *`U`*, das drei Methoden deklariert. Schreiben Sie eine Klasse `A` mit einer Methode, die mittels einer anonymen inneren Klasse ein Objekt erzeugt und die Referenz darauf als Typ *`U`* zurückgibt. Schreiben Sie eine weitere Klasse `B`, die ein Array vom Typ *`U`* enthält und drei Methoden besitzt: Die erste Methode erwartet eine Objektreferenz vom Typ *`U`* und speichert sie im Array. Die zweite Methode setzt die per Argument definierte Referenz im Array auf den Wert `null`. Die dritte Methode bewegt sich durch das Array und ruft die in *`U`* deklarierten Methoden auf. Die `main()`-Methode erzeugt mehrere `A`-Objekte und ein einzelnes `B`-Objekt. Füllen Sie das Array im `B`-Objekt mit den von der Methode in `A` gelieferten *`U`*-Referenzen. Verwenden Sie

das B-Objekt, um to call back into all the A-Objekte. Löschen Sie einige der *U*-Referenzen aus dem Array des B-Objektes. ■

11.8.1 Funktionsabschlüsse und Rückruffunktionen

[67] Ein *Funktionsabschluß* (*closure*) ist ein aufrufbares Objekt, das Informationen über den Geltungsbereich konserviert, in dem es erzeugt wurde. Mit dieser Definition können Sie nachvollziehen, daß ein Objekt einer inneren Klasse einen objektorientierten Funktionsabschluß repräsentiert. Ein solches Objekt beinhaltet nicht nur sämtliche Informationen über das Objekt der äußeren Klasse (Geltungsbereich in dem das Objekt der inneren Klasse erzeugt wurde), sondern speichert automatisch eine Referenz auf dieses Objekt und ist berechtigt, auf alle seine Komponenten zuzugreifen, insbesondere auf private Komponenten.

[68] Ein zwingendes Argument für eine Art von Zeigermechanismus bei Java war, *Rückruffunktionen* (*callbacks*) zu ermöglichen. Bei diesem Konzept wird einem Objekt eine Informationen übergeben, die diesem Objekt gestattet, zu einem späteren Zeitpunkt to call back into das ursprüngliche Objekt. Wie Sie in diesem Buch noch sehen werden, ist dieses Konzept sehr mächtig. Beim Implementieren einer Rückruffunktion über einen Zeiger müssen Sie sich allerdings darauf verlassen, daß sich der Clientprogrammierer anständig verhält und den Zeiger nicht mißbraucht. Java ist in dieser Hinsicht sorgfältiger, so daß Zeiger nicht in den Sprachumfang aufgenommen wurden.

[69] Der Funktionsabschluß mit Hilfe einer inneren Klasse ist eine gute Lösung, das heißt die Lösung ist flexibler und deutlich sicherer als ein Ansatz mit einem Zeiger. Ein Beispiel:

```
//: innerclasses/Callbacks.java
// Using inner classes for callbacks
package innerclasses;
import static net.mindview.util.Print.*;

interface Incrementable {
    void increment();
}

// Very simple to just implement the interface:
class Callee1 implements Incrementable {
    private int i = 0;
    public void increment() {
        i++;
        print(i);
    }
}

class MyIncrement {
    public void increment() { print("Other operation"); }
    static void f(MyIncrement mi) { mi.increment(); }
}

// If your class must implement increment() in
// some other way, you must use an inner class:
class Callee2 extends MyIncrement {
    private int i = 0;
    public void increment() {
        super.increment();
        i++;
        print(i);
    }
    private class Closure implements Incrementable {
```

```
        public void increment() {
            // Specify outer-class method, otherwise
            // you'd get an infinite recursion:
            Callee2.this.increment();
        }
    }
    Incrementable getCallbackReference() {
        return new Closure();
    }
}

class Caller {
    private Incrementable callbackReference;
    Caller(Incrementable cbh) { callbackReference = cbh; }
    void go() { callbackReference.increment(); }
}

public class Callbacks {
    public static void main(String[] args) {
        Callee1 c1 = new Callee1();
        Callee2 c2 = new Callee2();
        MyIncrement.f(c2);
        Caller caller1 = new Caller(c1);
        Caller caller2 = new Caller(c2.getCallbackReference());
        caller1.go();
        caller1.go();
        caller2.go();
        caller2.go();
    }
} /* Output:
      Other operation
      1
      1
      2
      Other operation
      2
      Other operation
      3
   *///:~
```

[70] Dieses Beispiel zeigt einen weiteren Unterschied zwischen der Implementierung eines Interfaces in einer äußeren Klasse und der Implementierung in einer inneren Klasse. Vom Programmieraufwand her, ist die Klasse `Callee1` offensichtlich die einfachere Lösung. Die Klasse `Callee2` ist dagegen von `MyIncrement` abgeleitet, welche bereits eine eigene `increment()`-Methode definiert, die nichts mit der durch *`Incrementable`* beschriebenen Funktionalität zu tun hat. Wenn `Callee2` von `MyIncrement` abgeleitet wird, kann die `increment()`-Methode nicht überschrieben werden, so daß sie zugleich dem Interface *`Incrementable`* genügt. Sie sind vielmehr gezwungen, eine separate Implementierung mittels einer inneren Klasse anzulegen. Beachten Sie, daß die Schnittstelle der äußeren Klasse durch die Definition einer innerer Klassen *nicht* ändert.

[71] Mit Ausnahme der Methode `getCallbackReference()` sind alle Komponenten der Klasse `Callee2` privat. Das Interface *`Incrementable`* ist wesentlich, um eine Verbindung nach außen zu ermöglichen. Dieses Beispiel demonstriert, wie Interfaces die vollständige Trennung von Schnittstelle und Implementierung ermöglichen.

[72] Die innere Klasse `Closure` implementiert *`Incrementable`*, um einen (sicheren) Eintrittspunkt in `Callee2` zu definieren. Der Empfänger der *`Incrementable`*-Referenz kann selbstverständlich nur

die `increment()`-Methode aufrufen und erhält keine anderen Zugriffsmöglichkeiten (im Gegensatz zu einem Zeiger).

[73] Der Konstruktor der Klasse `Caller` erwartet eine Referenz auf ein Objekt vom Typ *`Incrementable`* (die Referenz auf das Rückrufobjekt kann aber jederzeit gespeichert werden).

[74] Der Wert des Rückrufobjektes besteht in der Flexibilität: Sie können zur Laufzeit dynamisch entscheiden, welche Methoden aufgerufen werden. Die Vorteile dieses Ansatzes werden in Kapitel 23 klarer. Dort werden Rückruffunktionen überall verwendet, um GUI-Funktionalität zu implementieren.

11.8.2 Innere Klassen bei Kontrollframeworks

[75] Das Konzept, das ich in diesem Abschnitt über die Motivation zur Verwendung innerer Klassen als *Kontrollframework* bezeichnen möchte, liefert ein konkreteres Beispiel für die Anwendung innerer Klassen.

[76] Ein *Applikationsframework* besteht aus einer oder mehreren Klassen, die zur Lösung einer bestimmten Sorte von Problemen entworfen wurde(n). Die Anwendung eines solchen Frameworks besteht darin, daß Sie typischerweise eine oder mehrere seiner Klasse ableiten und einen Teil der dort definierten Methoden überschreiben. Ihre Anweisungen in den überschreibenden Methoden passen die durch das Framework gegebene allgemeine Lösung an Ihr spezifisches Problem an. Dies ist ein Beispiel für das Entwurfsmuster *Templatemethod*, siehe *Thinking in Patterns, Problem-Solving Techniques using Java*. Die Schablonenmethode implementiert die grundsätzliche Struktur des Algorithmus' und ruft eine oder mehrere überschreibbare Methoden auf, welche die Funktionsweise des Algorithmus' vervollständigen. Ein Entwurfsmuster trennt die veränderlichen Bestandteile von den unveränderlichen. Beim Entwurfsmuster *Templatemethod* stellt die Schablonenmethode den invarianten Anteil dar, die überschreibbaren Methoden dagegen den veränderlichen Anteil.

[77] Ein Kontrollframework ist ein Applikationsframework mit der Aufgabe, auf Ereignisse zu reagieren. Ein System heißt *ereignisgetrieben*, wenn es primär auf Ereignisse reagiert. Das Programmieren einer, in der Regel fast vollständig ereignisgetriebenen graphischen Benutzeroberfläche ist eine häufige Aufgabe in der Anwendungsentwicklung. Wie Sie in Kapitel 23 lernen werden, ist die Swing-Bibliothek von Java ein Kontrollframework, das den ereignisorientierten Charakter graphischer Benutzeroberflächen elegant abbildet und in erheblichem Ausmaß innere Klassen gebraucht.

[78] Das Beispiel in diesem Unterabschnitt (Anlagensteuerung in einem Gewächshaus) zeigt die einfache Entwicklung und Verwendung von Kontrollframeworks mit Hilfe innerer Klassen, ausgehend von einem allgemeineren Framework, das Aufgabe hat, Ereignisse auszulösen, nach dem sie „fertig" sind. Der Ereigniszustand „fertig" kann beliebig definiert werden. Die Anlagensteuerung bezieht sich hierbei auf Uhrzeiten. Das Kontrollframework beinhaltet keinerlei spezifische Information darüber, worauf es angewendet wird. Diese Informationen wird im Rahmen der Ableitung von der Klasse `Event` (siehe unten) übergeben, wenn die ereignisspezifischen `action()`-Methoden des Algorithmus implementiert werden.

[79] Die folgende Klasse `Event` repräsentiert die Schnittstelle zur Beschreibung der kontrollierten Ereignisse. Die Schnittstelle ist in Form einer abstrakten Klasse, anstelle eines tatsächlichen Interfaces gegeben, da die zeitabhängige Steuerung als Standardverhalten gewählt wurde und zum Teil bereits hier implementiert werden kann:

```
//: innerclasses/controller/Event.java
// The common methods for any control event.
package innerclasses.controller;
```

```
public abstract class Event {
    private long eventTime;
    protected final long delayTime;
    public Event(long delayTime) {
        this.delayTime = delayTime;
        start();
    }
    public void start() { // Allows restarting
        eventTime = System.nanoTime() + delayTime;
    }
    public boolean ready() {
        return System.nanoTime() >= eventTime;
    }
    public abstract void action();
} ///:~
```

[80] Der Konstruktor erwartet die zeitliche Verzögerung relativ zum Zeitpunkt der Erzeugung des `Event`-Objektes, nach der das Ereignis ausgelöst werden soll und ruft die `start()`-Methode auf. Die `start()`-Methode ermittelt die aktuelle Uhrzeit und addiert die angeforderte Verzögerung, um den Zeitpunkt zu bestimmen, an dem das Ereignis ausgelöst wird. Die Berechnung des Auslösezeitpunktes wird nicht im Konstruktor ausgeführt, sondern als separate Methode ausgelagert, um den Mechanismus nach dem Auslösen eines Ereignisses neu starten, das `Event`-Objekt also wiederverwenden zu können. Ein sich wiederholendes Ereignis kann durch Aufrufen der `start()`-Methode in der entsprechenden `action()`-Methode definiert werden.

[81] Die `ready()`-Methode gibt an, ob das Ereignis „fertig“ ist, das heißt ob die `action()`-Methode aufgerufen werden muß. Die Methode kann selbstständlich in einer von `Event` abgeleiteten Klasse überschrieben werden, um eine andere Bezugsgröße als die Uhrzeit zu verwenden.

[82–83] Die folgende Klasse *`Controller`* repräsentiert das eigentliche Kontrollframework, welches die Ereignisse verwaltet und auslöst. Die `Event`-Objekte werden in einem Container vom Typ *`List<Event>`* gespeichert (siehe Kapitel 12). Im Augenblick genügt es, wenn Sie sich die folgenden Eigenschaften merken: Die `add()`-Methode fügt ein `Event`-Objekt an das Ende der Liste im Container an. Die `size()`-Methode gibt die Anzahl der im Container gespeicherten Elemente zurück. Die erweiterte `for`-Schleife durchläuft den Inhalt des Containers elementweise. Die `remove()`-Methode entfernt das `Event`-Objekt mit dem angeforderten Index aus dem Container:

```
//: innerclasses/controller/Controller.java
// The reusable framework for control systems.
package innerclasses.controller;
import java.util.*;

public class Controller {
    // A class from java.util to hold Event objects:
    private List<Event> eventList = new ArrayList<Event>();
    public void addEvent(Event c) { eventList.add(c); }
    public void run() {
        while(eventList.size() > 0)
            // Make a copy so you're not modifying the list
            // while you're selecting the elements in it:
            for(Event e : new ArrayList<Event>(eventList))
                if(e.ready()) {
                    System.out.println(e);
                    e.action();
                    eventList.remove(e);
                }
    }
```

```
} ///:~
```

Die `run()`-Methode durchläuft eine Kopie des von `eventList` referenzierten *`List<Event>`*-Containers auf der Suche nach „fertigen“ `Event`-Objekten. Findet die Methode ein „fertiges“ `Event`-Objekt, so gibt sie über dessen `toString()`-Methode Auskunft über dieses Objekt, ruft seine `action()`-Methode auf und entfernt es anschließend aus dem Container.

[84] Beachten Sie im Hinblick auf dieses Design, daß Sie noch nichts darüber wissen, worin ein Ereignis eigentlich besteht. Dies ist genau der Kernpunkt des Designs: Die Trennung der veränderlichen Dinge von den unveränderlichen Dingen. Der „Änderungsvektor“ (*vector of change*), meine persönliche Bezeichnung, besteht aus den verschiedenen Funktionen der einzelnen `Event`-Objekte. Verschiedene Funktionen manifestieren sich in unterschiedlichen von `Event` abgeleiteten Klassen.

[85] Hier kommen die inneren Klassen ins Spiel. Sie gestatten zwei Dinge:

- Das gesamte Kontrollframework kann in einer einzigen Klasse implementiert werden, so daß alle implementierungsspezifischen Eigenschaften und Fähigkeiten gekapselt sind. Innere Klassen werden verwendet, um die unterschiedlichen, zur Lösung des Problem benötigten Funktionen ausdrücken (hier in `action()`-Methoden).
- Innere Klassen verhindern, daß die Implementierung des Kontrollframeworks unübersichtlichg wird, da Sie mühelos jede Komponente in der äußeren Klasse erreichen können. Ohne diese Fähigkeit würde der Quelltext unter Umständen unbequem genug werden, um sich nach einer alternativen Lösung umzusehen.

[86] Wir konzentrieren uns nun auf die angekündigte spezifische Implementierung unseres Kontrollframeworks zur Anlagensteuerung in einem Gewächshaus.[4] Die Steuerung umfaßt die folgenden, völlig unterschiedlichen Ereignisse: Beleuchtung ein- und ausschalten, Bewässerung und Thermostaten ein- und ausschalten, eine Glocke läuten sowie einen Neustart des Systems auslösen. Innere Klasse ermöglichen Ihnen, viele von ein und derselben Basisklasse (`Event`) abgeleitete Versionen in einer einzigen Klasse unterzubringen. Sie leiten für jedes Ereignis eine neue innere Klasse von `Event` ab und implementieren die Steuerfunktionen in den `action()`-Methoden.

[87] Wie für ein Kontrollframework typisch, ist die Klasse `GreenhouseControls` von *`Controller`* abgeleitet:

```
//: innerclasses/GreenhouseControls.java
// This produces a specific application of the
// control system, all in a single class. Inner
// classes allow you to encapsulate different
// functionality for each type of event.
import innerclasses.controller.*;

public class GreenhouseControls extends Controller {
    private boolean light = false;
    public class LightOn extends Event {
        public LightOn(long delayTime) { super(delayTime); }
        public void action() {
            // Put hardware control code here to
            // physically turn on the light.
            light = true;
        }
        public String toString() { return ”Light is on”; }
    }
    public class LightOff extends Event {
```

[4] Dieses Problem hat mich stets angesprochen. Es stammt aus meinem früheren Buch *C++ Inside & Out*. Java gestattet aber eine elegantere Lösung.

```
    public LightOff(long delayTime) { super(delayTime); }
    public void action() {
        // Put hardware control code here to
        // physically turn off the light.
        light = false;
    }
    public String toString() { return ''Light is off''; }
}
private boolean water = false;
public class WaterOn extends Event {
    public WaterOn(long delayTime) { super(delayTime); }
    public void action() {
        // Put hardware control code here.
        water = true;
    }
    public String toString() {
        return ''Greenhouse water is on'';
    }
}
public class WaterOff extends Event {
    public WaterOff(long delayTime) { super(delayTime); }
    public void action() {
        // Put hardware control code here.
        water = false;
    }
    public String toString() {
        return ''Greenhouse water is off'';
    }
}
private String thermostat = ''Day'';
public class ThermostatNight extends Event {
    public ThermostatNight(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Put hardware control code here.
        thermostat = ''Night'';
    }
    public String toString() {
        return ''Thermostat on night setting'';
    }
}
public class ThermostatDay extends Event {
    public ThermostatDay(long delayTime) {
        super(delayTime);
    }
    public void action() {
        // Put hardware control code here.
        thermostat = ''Day'';
    }
    public String toString() {
        return ''Thermostat on day setting'';
    }
}
// An example of an action() that inserts a
// new one of itself into the event list:
public class Bell extends Event {
```

```
        public Bell(long delayTime) { super(delayTime); }
        public void action() {
            addEvent(new Bell(delayTime));
        }
        public String toString() { return "Bing!"; }
    }
    public class Restart extends Event {
        private Event[] eventList;
        public Restart(long delayTime, Event[] eventList) {
            super(delayTime);
            this.eventList = eventList;
            for(Event e : eventList)
                addEvent(e);
        }
        public void action() {
            for(Event e : eventList) {
                e.start(); // Rerun each event
                addEvent(e);
            }
            start(); // Rerun this Event
            addEvent(this);
        }
        public String toString() {
            return "Restarting system";
        }
    }
    public static class Terminate extends Event {
        public Terminate(long delayTime) { super(delayTime); }
        public void action() { System.exit(0); }
        public String toString() { return "Terminating"; }
    }
} ///:~
```

[88] Beachten Sie, daß die Felder `light`, `water` und `thermostat` zur äußeren Klasse gehören (`GreenhouseControls`), die inneren Klassen diese Felder aber dennoch ohne Qualifizierung oder spezielle Berechtigung erreichen können. Die `action()`-Methoden enthalten in der Regel Anweisungen zur Hardwaresteuerung.

[89] Die von `Event` abgeleiteten Klassen sind größtenteils gleich. Nur die Klassen `Bell` und `Restart` unterscheiden sich von den übrigen Ereignissen. Die Klasse `Bell` läutet die Glocke und setzt anschließend ein neues `Bell`-Objekt in den Ereigniscontainer ein, so daß die Glocke später noch einmal geläutet wird. Beachten Sie, wie die inneren Klassen Mehrfachvererbung zu implementieren scheinen: Die Klassen `Bell` und `Restart` verfügen über alle Methoden aus `Event` und zusätzlich über sämtliche Methoden der äußeren Klasse `GreenhouseControls`.

[90] Der Konstruktor der inneren Klasse `Restart` erwartet ein `Event`-Array, dessen Elemente in den Container der Anlagensteuerung eingetragen werden. Da auch die Klasse `Restart` von `Event` abgeleitet ist, können Sie über deren `action()`-Methode auch ein `Restart`-Objekt in den Ereigniscontainer einsetzen, um das System regelmäßig neu zu starten.

[91] Die folgende Klasse `GreenhouseController` konfiguriert das System, indem sie ein `GreenhouseControls`-Objekt erzeugt und eine Reihe verschiedener `Event`-Objekte registriert. Dies ist ein Beispiel für das Entwurfsmuster *Command*: Jedes Element in dem von `eventList` referenzierten *`List<Event>`*-Container repräsentiert eine als Objekt gekapselte Anfrage:

```
//: innerclasses/GreenhouseController.java
// Configure and execute the greenhouse system.
```

```
// {Args: 5000}
import innerclasses.controller.*;

public class GreenhouseController {
    public static void main(String[] args) {
        GreenhouseControls gc = new GreenhouseControls();
        // Instead of hard-wiring, you could parse
        // configuration information from a text file here:
        gc.addEvent(gc.new Bell(900));
        Event[] eventList = {
            gc.new ThermostatNight(0),
            gc.new LightOn(200),
            gc.new LightOff(400),
            gc.new WaterOn(600),
            gc.new WaterOff(800),
            gc.new ThermostatDay(1400)
        };
        gc.addEvent(gc.new Restart(2000, eventList));
        if(args.length == 1)
            gc.addEvent(new GreenhouseControls.Terminate(new Integer(args[0])));
        gc.run();
    }
} /* Output:
      Bing!
      Thermostat on night setting
      Light is on
      Light is off
      Greenhouse water is on
      Greenhouse water is off
      Thermostat on day setting
      Restarting system
      Terminating
   *///:~
```

[92] Die Klasse `GreenhouseController` initialisiert die Anlagensteuerung des Gewächshauses und legt die verschiedenen Ereignisse zeitlich fest. Das `Restart`-Ereignis wird wiederholt ausgeführt und übergibt dem `GreenhouseControls`-Objekt jedesmal ein Array von Ereignissen. Sie können über die Kommandzeile (zum Testen) eine Zeitdauer in Millisekunden übergeben, nach der das Programm die Steuerung der Anlage beendet.

[93] Eine flexiblere Lösung wäre, die Ereignisse nicht hart zu kodieren, sondern aus einer Datei einzulesen. Übungsaufgabe 11 in Unterabschnitt 19.6.1 (Seite 719) fordert Sie auf, das Gewächshausbeispiel entsprechend zu ändern.

[94] Dieses Beispiel sollte Ihre Einschätzung des Wertes innerer Klassen positiv beinflussen, vor allem hinsichtlich der Verwendung in einem Kontrollframework. In Kapitel 23 werden Sie sehen, wie elegant sich innere Klassen zur Ereignisbehandlung bei graphischen Benutzeroberflächen eignen.

Übungsaufgabe 24: (2) Definieren Sie im Beispiel *GreenhouseControls.java* zwei neue Unterklassen von `Event`, die eine Belüftungsanlage ein- und ausschalten. Konfigurieren Sie *GreenhouseController.java*, so daß die neuen Ereignisse ausgelöst werden. ■

Übungsaufgabe 25: (3) Leiten Sie eine Klasse von `GreenhouseControls` ab, um zwei neue Unterklassen von `Event` zu definieren, die einen Nebelgenerator ein- und ausschalten. Schreiben Sie eine geänderte Version des Beispiels *GreenhouseController.java*, so daß die neuen Ereignisse ausgelöst werden. ■

11.9 Ableitung von einer inneren Klasse

[95] Die Ableitung von einer inneren Klasse ist ein wenig komplizierter, da der Konstruktor einer inneren Klasse das erzeugte Objekt mit einer Referenz auf das Objekt der entsprechenden äußeren Klasse verknüpfen muß. Das Problem besteht darin, daß die „geheime“ Referenzvariable mit einer Referenz initialisiert werden *muß*, eine von einer inneren Klasse abgeleitete Klasse aber kein „Standardobjekt“ zur Verfügung hat, mit dem eines ihrer Objekte verknüpft werden kann. Sie müssen diese Verknüpfung mit Hilfe einer speziellen Syntax explizit angeben:

```
//: innerclasses/InheritInner.java
// Inheriting an inner class.

class WithInner {
    class Inner {}
}

public class InheritInner extends WithInner.Inner {
    //! InheritInner() {} // Won't compile
    InheritInner(WithInner wi) {
        wi.super();
    }
    public static void main(String[] args) {
        WithInner wi = new WithInner();
        InheritInner ii = new InheritInner(wi);
    }
} ///:~
```

Die Klasse `InheritInner` ist nur von der inneren Klasse `Inner` abgeleitet, nicht von der äußeren. Der Compiler akzeptiert keinen Standardkonstruktor und es genügt auch nicht, eine Referenz auf ein Objekt der äußeren Klasse zu übergeben. Sie müssen *außerdem* im Konstruktor die folgende Syntax verwenden:

```
enclosingClassReference.super();
```

Diese Notation gewährleistet die erforderliche Referenz auf ein Objekt der äußeren Klasse und das Programm läßt sich übersetzen.

Übungsaufgabe 26: (2) Schreiben Sie eine Klasse mit einer inneren Klasse, die einen Nicht-Standardkonstruktor hat (einen parameterbehafteten Konstruktor). Schreiben Sie eine zweite Klasse mit einer inneren Klasse, die von der inneren Klasse der ersten Klasse abgeleitet ist. ■

11.10 Können innere Klassen überschrieben werden?

[96–97] Was geschieht, wenn Sie eine innere Klasse definieren, eine Klasse von der äußeren Klasse ableiten und die innere Klasse erneut definieren? Ist es möglich, die gesamte innere Klasse zu „überschreiben“? Dies könnte ein mächtiges Konzept sein. Es zeigt sich allerdings, daß das „Überschreiben“ einer inneren Klasse, als wäre sie eine Methode der äußeren Klasse, tatsächlich wirkungslos ist:

```
//: innerclasses/BigEgg.java
// An inner class cannot be overriden like a method.
import static net.mindview.util.Print.*;

class Egg {
    private Yolk y;
    protected class Yolk {
        public Yolk() { print("Egg.Yolk()"); }
```

```
    }
    public Egg() {
        print("New Egg()");
        y = new Yolk();
    }
}

public class BigEgg extends Egg {
    public class Yolk {
        public Yolk() { print("BigEgg.Yolk()"); }
    }
    public static void main(String[] args) {
        new BigEgg();
    }
} /* Output:
      New Egg()
      Egg.Yolk()
   *///:~
```

Der Standardkonstruktor von `BigEgg` wird vom Compiler automatisch erzeugt und ruft den Standardkonstruktor der Basisklasse `Egg` auf. Sie haben eventuell vermutet, daß beim Erzeugen des `BigEgg`-Objektes die „überschriebene“ Version von `Yolk` verwendet wird, aber dies trifft nicht zu, wie die Ausgabe dokumentiert.

[98–99] Das Beispiel zeigt, daß beim Ableiten der äußeren Klasse keine ~~extra inner-class magic~~ stattfindet. Die beiden inneren Klassen sind völlig voneinander getrennte Entitäten, jede in ihrem eigenen Namensraum. Es ist aber dennoch möglich, die innere Klasse der abgeleiteten äußeren Klasse von der inneren Klasse der äußeren Basisklasse abzuleiten:

```
//: innerclasses/BigEgg2.java
// Proper inheritance of an inner class.
import static net.mindview.util.Print.*;

class Egg2 {
    protected class Yolk {
        public Yolk() { print("Egg2.Yolk()"); }
        public void f() { print("Egg2.Yolk.f()");}
    }
    private Yolk y = new Yolk();
    public Egg2() { print("New Egg2()"); }
    public void insertYolk(Yolk yy) { y = yy; }
    public void g() { y.f(); }
}

public class BigEgg2 extends Egg2 {
    public class Yolk extends Egg2.Yolk {
        public Yolk() { print("BigEgg2.Yolk()"); }
        public void f() { print("BigEgg2.Yolk.f()"); }
    }
    public BigEgg2() { insertYolk(new Yolk()); }
    public static void main(String[] args) {
        Egg2 e2 = new BigEgg2();
        e2.g();
    }
} /* Output:
      Egg2.Yolk()
      New Egg2()
      Egg2.Yolk()
      BigEgg2.Yolk()
```

```
    BigEgg2.Yolk.f()
*///:~
```

Nun ist `BigEgg2.Yolk` explizit von `Egg2.Yolk` abgeleitet und überschreibt die Methoden ihrer Basisklasse. Die `insertYolk()`-Methode der Klasse `Egg2` gestattet der Klasse `BigEgg2`, die Referenzvariable `y` in `Egg2` mit einer aufwärts umgewandelten Referenz auf eines ihrer eigenen `Yolk`-Objekte zu bewerten, so daß die `g()`-Methode bei `y.f()` die in `BigEgg2` überschriebene Version von `f()` aufruft. Der zweite Aufruf des Konstruktors von `Egg2.Yolk` rührt vom Aufruf des Konstruktors der Basisklasse im Konstruktor von `BigEgg2` her. An der Ausgabe sehen sie, daß beim Aufruf von `g()` die überschriebene Version von `f()` gewählt wird.

11.11 Lokale innere Klassen

[100–101] Wie in Abschnitt 11.5 bereits beschrieben, können innere Klassen auch in Anweisungsblöcken definiert werden, typischerweise in Methodenkörpern. Eine lokale innere Klasse kann nicht mit Zugriffsmodifikatoren deklariert werden, da sie keine Komponente der äußeren Klasse ist, hat aber Zugriff auf die finalen Variablen des aktuellen Anweisungsblocks und alle Komponenten der äußeren Klasse. Das folgende Beispiel vergleicht die Definition einer lokalen mit der Definition einer anonymen inneren Klasse:

```
//: innerclasses/LocalInnerClass.java
// Holds a sequence of Objects.
import static net.mindview.util.Print.*;

interface Counter {
    int next();
}

public class LocalInnerClass {
    private int count = 0;
    Counter getCounter(final String name) {
        // A local inner class:
        class LocalCounter implements Counter {
            public LocalCounter() {
                // Local inner class can have a constructor
                print("LocalCounter()");
            }
            public int next() {
                printnb(name); // Access local final
                return count++;
            }
        }
        return new LocalCounter();
    }
    // The same thing with an anonymous inner class:
    Counter getCounter2(final String name) {
        return new Counter() {
                // Anonymous inner class cannot have a named
                // constructor, only an instance initializer:
                {
                    print("Counter()");
                }
                public int next() {
                    printnb(name); // Access local final
                    return count++;
```

```
                }
            };
        }
        public static void main(String[] args) {
            LocalInnerClass lic = new LocalInnerClass();
            Counter
                c1 = lic.getCounter("Local inner "),
                c2 = lic.getCounter2("Anonymous inner ");
            for(int i = 0; i < 5; i++)
                print(c1.next());
            for(int i = 0; i < 5; i++)
                print(c2.next());
        }
    } /* Output:
          LocalCounter()
          Counter()
          Local inner 0
          Local inner 1
          Local inner 2
          Local inner 3
          Local inner 4
          Anonymous inner 5
          Anonymous inner 6
          Anonymous inner 7
          Anonymous inner 8
          Anonymous inner 9
       *///:~
```

Das Interface *Counter* beschreibt die Rückgabe des nächsten Wertes einer Folge und wird je einmal als lokale und als anonyme innere Klasse implementiert, beide mit identischen Eigenschaften und Fähigkeiten. Da der Name einer lokalen inneren Klasse außerhalb der Methoden unsichtbar ist, liegt die einzige Rechtfertigung für die Wahl einer lokalen statt einer anonymen inneren Klasse vor, wenn Sie einen benannten und/oder einen überladenen Konstruktor brauchen, da eine anonyme innere Klasse nur einen dynamischen Initialisierungsblock zur Verfügung hat.

[102] Ein weiterer Grund für eine lokale anstelle einer anonymen inneren Klasse besteht, wenn Sie mehr als ein Objekt dieser Klasse brauchen.

11.12 Namensschema für Klassendateien

[103] Da zu jeder Klasse eine Klassendatei (*.class* Datei) gehört, die sämtliche zum Erzeugen eines Objektes dieser Klasse benötigten Informationen enthält (diese Informationen liefern eine „Meta-Klasse“, ein sogenanntes Klassenobjekt (`Class`-Objekt)), liegt die Vermutung nahe, daß auch zu jeder inneren Klasse eine Klassendatei gehört, welche die entsprechenden Informationen für ihr Klassenobjekt beinhaltet. Die Namen dieser Klassendateien folgen einem strikten Schema: Dem Namen der äußeren Klasse folgt ein $-Zeichen und anschließend der Name der inneren Klasse. Beispielsweise gehören die folgenden Klassendateien zum Beispiel *LocalInnerClass.java* aus dem vorigen Abschnitt:

```
Counter.class
LocalInnerClass$1.class
LocalInnerClass$1LocalCounter.class
LocalInnerClass.class
```

[104] Bei anonymen inneren Klassen vergibt der Compiler einfach Nummern als Bezeichner. Bei mehrfach geschachtelten inneren Klassen schließt sich der Name einer inneren Klasse nach einem $-Zeichen dem Namen ihrer unmittelbaren äußeren Klasse an.

[105] Dieses Namensschema ist ebenso einfach und geradlinig wie robust und bewältigt die meisten Situationen.[5] Da es sich um das Standardnamensschema von Java handelt, sind die erzeugten Dateien automatisch plattformunabhängig. ~~(Note that the Java compiler is changing your inner classes in all sorts of other ways in order to make them work.)~~

11.13 Zusammenfassung

[106] Interfaces und innere Klassen sind anspruchsvollere Konzepte, als die Dinge, die Sie in vielen anderen objektorientierten Sprachen finden. C++ kennt beispielsweise nicht vergleichbares. Kombiniert lösen Sie dasselbe Problem, das C++ mit der Fähigkeit zur Mehrfachvererbung zu lösen versucht. Die Mehrfachvererbung bei C++ stellt sich in der Anwendung allerdings als schwierig heraus, während die Interfaces und inneren Klassen von Java im Vergleich viel leichter zugänglich sind.

[107] Auch wenn beide Konzepte relativ einfach sind, ist ihre Verwendung, wie bei der Polymorphie, eine Designangelegenheit. Ihre Fähigkeit, zu erkennen in welcher Situation Sie ein Interface oder eine innere Klasse oder beides verwenden sollten, wird sich im Laufe der Zeit verbessern. An dieser Stelle des Buches sollte Ihnen zumindest die Syntax und Semantik vertraut sein. Wenn Sie Interfaces und innere Klassen in Anwendungen sehen, prägen sie sich schließlich ein.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

[5] Andererseits ist das Dollarzeichen ($) ein Metazeichen der Unixshell und kann beim Anzeigen der *.class* Dateien Schwierigkeiten verursachen. Das ist merkwürdig, da Sun Microsystems ein unixbasiertes Unternehmen ist. Ich habe den Verdacht, daß dieser Aspekt bei Sun Microsystems nicht beachtet wurde und man statt dessen dachte, daß Sie sich natürlich auf die Quelltextdateien konzentrieren würden.

Kapitel 12

Einführung in die Containerbibliothek

Inhaltsübersicht

[0] Nur ein sehr einfaches Programm hat eine feste Anzahl von Objekten mit bestimmter Lebensdauer.

[1] Die Anzahl erzeugter Objekte hängt bei Ihren Programmen in der Regel von Kriterien ab, die erst zur Laufzeit ausgewertet werden können. Vor dem Programmstart haben Sie keine Informationen über die Anzahl der benötigten Objekte zur Verfügung, eventuell nicht einmal über deren exakten Typ. Das Problem lautet, allgemein formuliert, eine beliebige Anzahl von Objekten zu einem beliebigen Zeitpunkt an einer beliebigen Stelle im Programm zu erzeugen. Es genügt also nicht, für jedes benötigte Objekt eine benannte Referenzvariable zu deklarieren:

```
MyType aReference;
```

da Sie nicht wissen, wieviele solcher Referenzvariablen tatsächlich erforderlich sind.

[2] Die meisten Programmiersprachen haben eine Lösung für diese grundlegende Anforderung parat. Java verfügt über mehrere Möglichkeiten zur Speicherung von Objekten (eigentlich von Referenzen

auf Objekte). Der Compiler unterstützt die in den vorausgegangenen Kapiteln behandelten Arrays, siehe zum Beispiel Unterabschnitt 3.2.3 und Abschnitt 6.8. Ein Array ist die effizienteste Möglichkeit, um eine Anzahl von Objekten zu speichern und die bevorzugte Wahl, wenn Sie primitive Werte speichern müssen. Ein Array hat allerdings eine feste Länge und während Sie das Programm schreiben, wissen Sie im allgemeinen nicht, wieviele Objekte Sie benötigen oder ob Sie eine anspruchsvollere Möglichkeit brauchen, um Ihre Objekte zu speichern. Die Begrenzung von Arrays auf eine feste Länge kann eine zu wesentliche Einschränkung sein.

[3] Das Package `java.util` umfaßt eine einigermaßen komplette Sammlung sogenannter *Containerklassen* oder *Container* zur Lösung dieses Problems, deren Basistypen die Interfaces `List`, `Set`, `Queue` und `Map` sind. Die Klassen unter diesen Interfaces werden in der Java-Bibliothek auch als *Kollektionsklassen* oder *Kollektionen* bezeichnet. Ich verwende den umfassenderen Begriff „Container“. Container bieten technisch ausgereifte Möglichkeiten, Ihre Objekte zu verwalten und Sie können eine überraschende Anzahl von Problemen mit Hilfe der Containerklassen lösen.

[4] Kennzeichnende Eigenschaften sind beispielsweise, daß Objekte vom Typ `Set` keine Duplikate enthalten, während sich Objekte vom Typ `Map` wie *assoziative Arrays* verhalten, also Objekte mit anderen Objekten verknüpfen. Die Objekte der Containerklassen von Java passen ihre Länge automatisch an. Im Gegensatz zu Arrays können Sie beliebig viele Objekte in einem Container speichern, ohne sich beim Schreiben des Programmes über die Länge des Containers Gedanken machen zu müssen.

[5] Obwohl Java[1] keine direkten Schlüsselwörter für die Containerklassen hat, sind sie ein fundamentaler Bestandteil der Sprache, der Ihre Möglichkeiten deutlich verbessert. Dieses Kapitel vermittelt grundlegende, für die Arbeit erforderliche Kenntnisse der Containerbibliothek von Java und betont dabei die typische Verwendungsweise. Wir konzentrieren uns auf die Container, die Sie im Alltag brauchen. In Kapitel 18 lernen Sie die übrigen Container und weitere Einzelheiten über ihre Funktionalität und Anwendung kennen.

12.1 Generische Typen und typsichere Container

[6] Eines der Probleme an den Containern vor Version 5 der Java Standard Edition (SE 5) war, daß der Compiler das Eintragen typfremder Elemente gestattete. Nehmen Sie zum Beispiel einen `ArrayList`-Container (das wichtigste Arbeitspferd unter den Containerklassen) mit `Apple`-Objekten. Fürs erste können Sie ein `ArrayList`-Objekt wie ein Array vorstellen, das sein Speichervermögen automatisch vergrößert. Die Klasse `ArrayList` ist leicht zu bedienen: Sie erzeugen ein Objekt, tragen per `add()` Elemente ein und fragen per Elemente per `get()` mit einem Index als Argument ab; wie bei Arrays, aber ohne eckige Klammern.[2] Die Klasse `ArrayList` verfügt außerdem über eine `size()`-Methode, die angibt, wieviele Elemente die Liste enthält, damit Sie nicht versehentlich einen Index jenseits des Endes wählen und einen Fehler verursachen (das heißt eine Ausnahme vom Typ `RuntimeException` hervorrufen; Ausnahmen werden in Kapitel 13 eingeführt.)

[7] Im folgenden Beispiel werden `Apple`- und `Orange`-Objekte in einem Container gespeichert und wieder entnommen. Normalerweise gibt der Java-Compiler eine Warnung aus, da kein generischer Container mit Parametertyp verwendet wird. Wir deklarieren eine spezielle *Annotation*, die seit der SE 5 existiert. Annotationen beginnen mit dem Zeichen `@` und können ein Argument haben. In diesem Beispiel hat die Annotation `@SuppressWarnings` das Argument `''unchecked''`, wodurch nur „unchecked“-Warnungen unterdrückt werden:

[1] Mehrere Sprachen, darunter Perl, Python und Ruby, unterstützen Container in nativer Weise.

[2] Hier wäre Operatorüberladung wünschenswert. Die Containerklassen von C++ und C# haben durch Operatorüberladung eine sauberere Syntax.

```
//: holding/ApplesAndOrangesWithoutGenerics.java
// Simple container example (produces compiler warnings).
// {ThrowsException}
import java.util.*;

class Apple {
    private static long counter;
    private final long id = counter++;
    public long id() { return id; }
}

class Orange {}

public class ApplesAndOrangesWithoutGenerics {
    @SuppressWarnings(''unchecked'')
    public static void main(String[] args) {
        ArrayList apples = new ArrayList();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // Not prevented from adding an Orange to apples:
        apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            ((Apple) apples.get(i)).id();
        // Orange is detected only at run time
    }
} /* (Execute to see output) *////:~
```

In Kapitel 21 lernen Sie mehr über die Annotationen der SE 5.

[8] Die Klassen `Apple` und `Orange` sind voneinander verschieden und haben keine Gemeinsamkeit außer, daß beide von `Object` abgeleitet sind. (Erinnern Sie sich: Wenn Sie nicht explizit angeben, von welcher Klasse Sie abgeleiten, erbt Ihre Klasse automatisch von `Object`.) Da ein `ArrayList`-Objekt Elemente vom Typ `Object` enthält, können Sie per `add()`-Methode nicht nur `Apple`-, sondern auch `Orange`-Objekte in den Container einsetzen, ohne daß zur Übersetzungs- oder Laufzeit Beschwerden gemeldet werden. Wenn Sie per `get()`-Methode ein vermeintliches `Apple`-Objekt aus dem Container entnehmen, erhalten Sie eine Referenz vom Typ `Object`, die Sie in den Typ `Apple` umwandeln müssen. Sie müssen den ganzen Ausdruck einklammern, um zu erzwingen, daß die Typumwandlung vor dem Aufruf der `Apple`-Methode `id()` ausgeführt wird. Andernfalls meldet der Compiler einen Syntaxfehler.

[9] Der Versuch, ein `Orange`-Objekt in ein `Apple`-Objekt umzuwandeln, wird zur Laufzeit mit einem Fehler in Gestalt der oben erwähnten Ausnahme quittiert.

[10] In Kapitel 16 werden Sie lernen, daß das Schreiben von generischen Klassen in Java kompliziert sein kann. Die Anwendung einer vordefinierten generischen Klasse ist dagegen in der Regel einfach. Beispielsweise definieren Sie ein `ArrayList`-Objekt, welches `Apple`-Objekte enthält, in dem Sie `ArrayList<Apple>` statt `ArrayList` wählen. Bei der Definition der generischen Klasse beinhalten die spitzen Klammern den sogenannten *Typparameter* (es gibt auch generische Klassen mit mehr als einem Typparameter), der beim Erzeugen eines Objektes der generischen Klasse durch den *Parametertyp* ersetzt wird, das heißt bei einem Container durch den Typ seiner Elemente.

[11] Generische Container schützen Sie *bereits zur Übersetzungszeit* vor dem Einsetzen typfremder Elemente.[3] Hier nochmals das obige Beispiel, diesesmal aber mit einem generischen Container:

```
//: holding/ApplesAndOrangesWithGenerics.java
```

[3] Am Ende von Kapitel 16 (Abschnitt 16.19) finden Sie eine Diskussion zu der Frage, ob das Einsetzen typfremder Elemente in einen Container tatsächlich ein wesentliches Problem ist. Kapitel 16 zeigt auch, daß die generischen Typen von Java neben typsicheren Containern andere nützliche Anwendungsfelder haben.

```
import java.util.*;

public class ApplesAndOrangesWithGenerics {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        for(int i = 0; i < 3; i++)
            apples.add(new Apple());
        // Compile-time error:
        // apples.add(new Orange());
        for(int i = 0; i < apples.size(); i++)
            System.out.println(apples.get(i).id());
        // Using foreach:
        for(Apple c : apples)
            System.out.println(c.id());
    }
} /* Output:
      0
      1
      2
      0
      1
      2
   *///:~
```

[12] Nun verhindert der Compiler das Einsetzen eines `Orange`-Objektes in die von `apples` referenzierte Liste, verlagert die Erkennung des Fehlers also von der Laufzeit hin zur Übersetzungszeit.

[13] Beachten Sie außerdem, daß die Typumwandlung beim Abfragen eines Objektes aus dem `ArrayList`-Objekt nicht mehr länger notwendig ist. Da das `ArrayList`-Objekt den Typ seiner Elemente kennt, führt es die Typumwandlung für Sie aus, wenn Sie die `get()`-Methode aufrufen. Der generische Container bewirkt nicht nur, daß der Compiler den Typ der eingesetzten Objekte prüft, sondern führt auch zu einer übersichtlicheren Syntax beim Abfragen der Elemente.

[14] Das Beispiel zeigt insbesondere, daß Sie die Liste mit Hilfe der erweiterten `for`-Schleife („Foreach-Syntax“) elementweise verarbeiten können, falls Sie einzelne Elemente nicht per Index referenzieren müssen.

[15] Sie sind bei einem generischen Container nicht an den exakten Parametertyp gebunden, sondern können Elemente jedes vom Parametertyp abgeleiteten Typs übergeben:

```
//: holding/GenericsAndUpcasting.java
import java.util.*;

class GrannySmith extends Apple {}
class Gala extends Apple {}
class Fuji extends Apple {}
class Braeburn extends Apple {}

public class GenericsAndUpcasting {
    public static void main(String[] args) {
        ArrayList<Apple> apples = new ArrayList<Apple>();
        apples.add(new GrannySmith());
        apples.add(new Gala());
        apples.add(new Fuji());
        apples.add(new Braeburn());
        for(Apple c : apples)
            System.out.println(c);
    }
} /* Output: (Sample)
```

```
    GrannySmith@7d772e
    Gala@11b86e7
    Fuji@35ce36
    Braeburn@757aef
  *///:~
```

[16] Sie können einem für `Apple`-Objekt eingerichteten Container also Objekte eines beliebigen von `Apple` abgeleiteten Typs übergeben.

[17] Das Ausgabeformat, das heißt der Klassenname mit vorzeichenloser Darstellung des Hashwertes des Objektes (generiert von der `hashCode()`-Methode), stammt von der Standardversion der `toString()`-Methode aus der Klasse `Object`. Sie lernen die Einzelheiten über Hashwerte in Kapitel 18.

Übungsaufgabe 1: (2) Schreiben Sie eine neue Klasse namens `Gerbil` (Wüstenrennmaus) mit einem `int`-Feld namens `gerbilNumber`, das per Konstruktor initialisiert wird. Legen Sie eine `hop()`-Methode an, welche die Kennummer der Wüstenrennmaus und die Information ausgibt, daß sie springt. Erzeugen Sie ein `ArrayList`-Objekt und setzen Sie einige `Gerbil`-Objekte ein. Verarbeiten Sie die Liste elementweise per `get()`-Methode und rufen Sie für jedes Objekt die `hop()`-Methode auf. ■

12.2 Grundlegende Konzepte

[18] Die Containerbibliothek von Java untergliedert das Speichern von Objekten in zwei verschiedene Konzepte, die sich in den beiden grundlegenden Interfaces der Bibliothek niederschlagen:

- *`Collection`*: Eine Reihe individueller Elemente für die wenigstens eine Regel gilt. Ein Container (der Begriff „Container“ schließt „Kollektionen“ ein; siehe Seite 310) vom Typ *`List`* muß ihre Elemente in der Reihenfolge speichern, in der sie eingesetzt wurden. Ein Container vom Typ *`Set`* darf keine Duplikate enthalten. Ein Container vom Typ *`Queue`* gibt seine Elemente in der durch ein bestimmtes Anordnungskriterium gegebenen Reihenfolge heraus (in der Regel die Einsetzungsreihenfolge).
- *`Map`*: Eine Reihe von Schlüssel/Wert-Paaren, die das Nachschlagen eines Wertes anhand seines Schlüssels gestattet. Ein Container vom Typ `ArrayList` gestattet das Abfragen eines Objektes mit Hilfe eines ganzzahligen Indexes, verknüpft („assoziiert“) also Indizes mit Objektes. Eine Container vom Typ *`Map`* ermöglicht dagegen das Nachschlagen eines Objektes anhand eines *anderen* Objektes und wird daher auch als *assoziatives Array* oder *Dictionary* bezeichnet, da nach Nachschlagen eines Wertobjektes mit Hilfe eines Schlüsselobjektes dem Nachschlagen einer Begriffsdefinition in einem Wörterbuch ähnelt. Container vom Typ *`Map`* sind mächtige Programmierwerkzeuge.

[19] Idealerweise beziehen Sie sich stets auf diese Interfaces, auch wenn es nicht immer möglich ist. Die Stelle, an der Sie den Bezugs- und den eigentlichen Objekttyp festlegen, ist die Deklaration der Referenzvariablen mit gleichzeitiger Objekterzeugung. Beispielsweise können Sie eine Liste so definieren:

```
List<Apple> apples = new ArrayList<Apple>();
```

Beachten Sie, im Vergleich mit den beiden vorangegangenen Beispielen, daß die Referenz auf das `ArrayList`-Objekt in den Interfacetyp *`List`* umgewandelt wird. Die Motivation, sich auf das Interface zu beziehen, besteht darin, daß Sie, falls Sie sich zu einer Änderung Ihrer Implementierung entscheiden, nur die Stelle der Objekterzeugung ändern müssen, zum Beispiel:

```
List<Apple> apples = new LinkedList<Apple>();
```

Sie erzeugen typischerweise ein Objekt einer konkreten Klasse, wandeln die Referenz auf das Objekt in den entsprechenden Interfacetyp um und verwenden im restlichen Quelltext stets das Interface für den Zugriff auf das Objekt.

[20] Dieser Ansatz läßt sich nicht immer befolgen, da manche Klassen zusätzliche Funktionalität haben. Beispielsweise hat die Klasse `LinkedList` (`TreeMap`) Methoden, die im Interface *`List`* (*`Map`*) nicht deklariert sind. Wenn Sie diese Methoden brauchen, scheidet die Typumwandlung zum allgemeineren Interface hin aus.

[21] Das Interface *`Collection`* verallgemeinert die Idee einer *Aufeinanderfolge*, also der Speicherung einer Anzahl von Objekten. Das folgende einfache Beispiel füllt eine Kollektion (hier einen `ArrayList`-Container) mit `Integer`-Objekten und gibt alle Elemente des resultierenden Containers aus:

```
//: holding/SimpleCollection.java
import java.util.*;

public class SimpleCollection {
    public static void main(String[] args) {
        Collection<Integer> c = new ArrayList<Integer>();
        for(int i = 0; i < 10; i++)
            c.add(i); // Autoboxing
        for(Integer i : c)
            System.out.print(i + ", ");
    }
} /* Output:
      0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
   *////:~
```

[22] Da dieses Beispiel nur Methoden aus dem Interface *`Collection`* aufruft, kann ein Objekt jeder Klasse gewählt werden, die dieses Interface implementiert. `ArrayList` ist der einfachste Listentyp.

[23] Der Name der `add()`-Methode erweckt den Anschein, daß sie der Kollektion ein neues Element hinzufügt. Die Dokumentation stellt dagegen sorgfältig fest, daß `add()` „gewährleistet, daß die Kollektion das jeweilige Element enthält.“ Diese Beschreibung berücksichtigt die Eigenschaft des Containertyps *`Set`*, ein Element nur aufzunehmen, wenn es noch nicht vorhanden ist. Bei `ArrayList` oder jedem anderen *`List`*-Typ bewirkt das Aufrufen der `add()`-Methode stets, daß das Element in die Liste eingetragen wird, da eine Liste Duplikate enthalten darf.

[24] Jeder Container vom Typ *`Collection`* kann, wie oben gezeigt, mit Hilfe der erweiterten `for`-Schleife elementweise verarbeitet werden. In Abschnitt 12.6 lernen Sie das flexiblere Konzept des Iterators kennen.

Übungsaufgabe 2: (1) Ändern Sie das Beispiel *SimpleCollection.java* so, daß `c` einen Container vom Typ *`Set`* referenziert. ■

Übungsaufgabe 3: (2) Ändern Sie das Beispiel *innerclasses/Sequence.java* so, daß Sie beliebig viele Elemente hinzufügen können. ■

12.3 Hinzufügen mehrerer Elemente

[25] Die Hilfsklassen `Arrays` und `Collections` im Package `java.util` verfügen über Methoden, die einem Container vom Typ *`Collection`* mehrere Elemente auf einmal übergeben. Die statische `Arrays`-Methode `asList()` erwartet entweder ein Array oder eine kommaseparierte Liste von

Elementen (per Argumentliste variabler Länge) und gibt eine Referenz auf einen *List*-Container zurück. Die statische `Collections`-Methode `addAll()` erwartet einen Container vom Typ *Collection* sowie entweder ein Array oder eine kommaseparierte Liste von Elementen und setzt es beziehungsweise sie in den Container ein. Das nächste Beispiel zeigt sowohl diese beiden Methoden als auch die gebräuchlichere `addAll()`-Methode, die jeder Container vom Typ *Collection* besitzt:

```
//: holding/AddingGroups.java
// Adding groups of elements to Collection objects.
import java.util.*;

public class AddingGroups {
    public static void main(String[] args) {
        Collection<Integer> collection =
            new ArrayList<Integer>(Arrays.asList(1, 2, 3, 4, 5));
        Integer[] moreInts = { 6, 7, 8, 9, 10 };
        collection.addAll(Arrays.asList(moreInts));
        // Runs significantly faster, but you can't
        // construct a Collection this way:
        Collections.addAll(collection, 11, 12, 13, 14, 15);
        Collections.addAll(collection, moreInts);
        // Produces a list ''backed by'' an array:
        List<Integer> list = Arrays.asList(16, 17, 18, 19, 20);
        list.set(1, 99); // OK - modify an element
        // list.add(21); // Runtime error because the
        // underlying array cannot be resized.
    }
} ///:~
```

[26] Der Konstruktor einer Containerklasse vom Typ *Collection* kann einen anderen Container vom Typ *Collection* entgegennehmen, um das gerade erzeugte Objekt zu initialisieren. Somit können Sie per `Arrays.asList()` eine Eingabe für den Konstruktor erzeugen. Andererseits wird `Collections.addAll()` erheblich schneller verarbeitet. Da es ebenso einfach ist, einen leeren *Collection*-Container zu erzeugen und anschließend `Collections.addAll()` aufzurufen, ist dies die bevorzugte Vorgehensweise.

[27] Die *Collection*-Methode (*ohne* „s“) `addAll()` kann nur mit einem Argument umgehen, welches wiederum einen *Collection*-Container referenziert und ist daher weniger flexibel als `Arrays.asList()` oder `Collections.addAll()`, die Argumentlisten variabler Länge haben.

[28] Sie können den von `Arrays.asList()` zurückgegebenen *List*-Container auch direkt verwenden, der allerdings an sein unterliegendes Array gebunden ist, also eine unveränderliche Länge hat. Der Versuch durch die Methoden `add()` oder `delete()` einer solchen Liste ein Element zu übergeben oder daraus zu entfernen, wird als Versuch interpretiert, die Länge des Arrays zu ändern und ruft zur Laufzeit eine Ausnahme vom Typ `UnsupportedOperationException` hervor.

[29] Eine Einschränkung der Methode `Arrays.asList()` besteht darin, daß sie den resultierenden Parametertyp des *List*-Container bestmöglich schätzt und den für die Referenzvariable festgelegten Parametertyp nicht beachtet. Das kann problematisch sein:

```
//: holding/AsListInference.java
// Arrays.asList() makes its best guess about type.
import java.util.*;

class Snow {}
class Powder extends Snow {}
class Light extends Powder {}
class Heavy extends Powder {}
class Crusty extends Snow {}
```

```
class Slush extends Snow {}

public class AsListInference {
    public static void main(String[] args) {
        List<Snow> snow1 =
            Arrays.asList(new Crusty(), new Slush(), new Powder());

        // Won't compile:
        // List<Snow> snow2 = Arrays.asList(
        // new Light(), new Heavy());
        // Compiler says:
        // found : java.util.List<Powder>
        // required: java.util.List<Snow>

        // Collections.addAll() doesn't get confused:
        List<Snow> snow3 = new ArrayList<Snow>();
        Collections.addAll(snow3, new Light(), new Heavy());

        // Give a hint using an
        // explicit type argument specification:
        List<Snow> snow4 = Arrays.<Snow>asList(new Light(), new Heavy());
    }
} ///:~
```

[30] Beim Erzeugen des von `snow2` referenzierten Objektes hat die Methode `Arrays.asList()` nur Untertypen von `Powder` zur Verfügung und erzeugt einen *`List<Powder>`*- anstelle eines *`List<Snow>`*-Containers, während die Methode `Collections.addAll()` den Zieltyp aus ihrem ersten Argument ermittelt.

[31] Wie Sie an `snow4` sehen, können Sie dem Compiler in der Mitte von `Arrays.asList()` einen Hinweis auf den tatsächlichen Parametertyp des erzeugten *`List`*-Containers liefern. Diese Syntaxkonstruktion wird als **explizite Angabe des Parametertyps** (*explicit type argument specification*) bezeichnet.

[32] Container vom Typ *`Map`* sind komplizierter und die Standardbibliothek von Java bietet keine Möglichkeit, einen solchen Container automatisch zu initialisieren, außer mit dem Inhalt eines anderen Containers vom Typ *`Map`*.

12.4 Ausgeben des Containerinhaltes

[33] Sie müssen die statische `Arrays`-Methode `toString()` aufrufen, um ein Array in eine druckbare Darstellung umzuwandeln. Der Inhalt eines Containers läßt sich dagegen ohne Hilfsmethode ansprechend ausgeben. Das folgende Beispiel stellt auch die wichtigsten Java-Container vor:

```
//: holding/PrintingContainers.java
// Containers print themselves automatically.
import java.util.*;
import static net.mindview.util.Print.*;

public class PrintingContainers {
    static Collection fill(Collection<String> collection) {
        collection.add("rat");
        collection.add("cat");
        collection.add("dog");
        collection.add("dog");
        return collection;
    }
    static Map fill(Map<String,String> map) {
```

```
        map.put(''rat'', ''Fuzzy'');
        map.put(''cat'', ''Rags'');
        map.put(''dog'', ''Bosco'');
        map.put(''dog'', ''Spot'');
        return map;
    }
    public static void main(String[] args) {
        print(fill(new ArrayList<String>()));
        print(fill(new LinkedList<String>()));
        print(fill(new HashSet<String>()));
        print(fill(new TreeSet<String>()));
        print(fill(new LinkedHashSet<String>()));
        print(fill(new HashMap<String,String>()));
        print(fill(new TreeMap<String,String>()));
        print(fill(new LinkedHashMap<String,String>()));
    }
} /* Output:
      [rat, cat, dog, dog]
      [rat, cat, dog, dog]
      [dog, cat, rat]
      [cat, dog, rat]
      [rat, cat, dog]
      {dog=Spot, cat=Rags, rat=Fuzzy}
      {cat=Rags, dog=Spot, rat=Fuzzy}
      {rat=Fuzzy, cat=Rags, dog=Spot}
   *///:~
```

[34] Das Beispiel verdeutlicht die zwei Hauptkategorien in der Containerbibliothek von Java. Die Unterscheidung richtet sich nach den Anzahl der Einträge pro „Speicherplatz“ eines Containers. In der Kategorie unter dem Interface *`Collection`* enthält jeder Speicherplatz nur ein Element. Diese Kategorie umfaßt die Containertypen *`List`*, der eine Anzahl von Elementen in einer bestimmten Reihenfolge enthält, *`Set`*, der nur Elemente aufnimmt, die noch nicht enthalten sind und *`Queue`*, bei dem Sie ein Element am einen „Ende“ einsetzen und am anderen „Ende“ wieder entnehmen (im obigen Beispiel wäre dieser Containertyp lediglich eine andere Perspektive auf eine Reihe von Elementen und entfällt daher). Die Containertypen in der zweiten Kategorie unter dem Interface *`Map`* enthalten dagegen zwei Objekte je Speicherplatz, nämlich einen *Schlüssel* und einen damit verknüpften *Wert*.

[35] Die Ausgabe zeigt, daß das durch die `toString()`-Methode des jeweiligen Containers gegebene Standardverhalten annehmbar zu lesende Ergebnisse liefert. Der Inhalt eines Containers vom Typ *`Collection`* wird in eckigen Klammern als kommaseparierte Liste ausgegeben. Der Inhalt eines Containers vom Typ *`Map`* wird in geschweiften Klammern als kommaseparierte Liste von Schlüssel/Wert-Paaren ausgegeben (Schlüssel links und Wert rechts eines Gleichheitszeichens).

[36] Die erste Version der `fill()`-Methode funktioniert bei jedem Container vom Typ *`Collection`*. Die `add()`-Methode übergibt dem Container ein neues Element.

[37] Sie sehen an der Ausgabe, daß beide verwendeten *`List`*-Containertypen (`ArrayList` und `LinkedList`) die Einsetzungsreihenfolge beibehalten. Beide Containertypen unterscheiden sich nicht nur bei bestimmten Operationen hinsichtlich ihrer Performanz, sondern `LinkedList` verfügt auch über mehr Operationen als `ArrayList`, siehe Abschnitte 12.7, 12.8 und 12.11.

[38] Die Containertypen `HashSet`, `TreeSet` und `LinkedHashSet` liegen unter dem Interface *`Set`*. Die Ausgabe zeigt, daß ein Container vom Typ *`Set`* ein Element höchstens einmal enthält und daß die verschiedenen Implementierungen ihre Elemente unterschiedlich speichern. Der `HashSet`-Container speichert seine Elemente nach einem ziemlich komplizierten Verfahren, das in Kapitel 18 diskutiert

wird. Im Augenblick genügt es, wenn Sie sich merken, daß es das schnellste Verfahren beim Abfragen von Elementen ist und die Reihenfolge der Speicherung sinnlos wirken kann (häufig genügt die Information, ob ein Container ein bestimmtes Element enthält und seine Position ist unwichtig). Wenn es auf die Reihenfolge der gespeicherten Elemente ankommt, können Sie einen `TreeSet`-Container wählen, welcher die Elemente miteinander vergleicht und in aufsteigender Reihenfolge anordnet. Ein `LinkedHashSet`-Container speichert seine Elemente in der Reihenfolge, in der sie übergeben wurden.

[39] Ein Container vom Typ *`Map`* („assoziatives Array") gestattet das Abfragen von Einträgen mit Hilfe eines *Schlüssels*, wie eine Datenbank. Das mit dem Schlüssel verknüpfte Objekt wird als *Wert* bezeichnet. Stellen Sie sich einen *`Map`*-Container vor, der Bundesstaaten auf ihre Hauptstädte abbildet. Wenn Sie wissen möchten, wie die Hauptstadt von Ohio heißt, fragen Sie den Container ab, indem Sie „Ohio" als Schlüssel angeben, analog zur Angabe des Index bei einem Array. Aufgrund dieses Verhaltens akzeptiert ein Container vom Typ *`Map`* höchstens ein Exemplar jedes Schlüssels.

[40] Die *`Map`*-Methode `put(key, value)` speichert den Wert `value` (die Information, die Sie nachschlagen) und verknüpft ihn mit einem Schlüssel `key` (die Information, die Sie zur Suche eingeben). Die *`Map`*-Methode `get(key)` fragt den mit dem Schlüssen `key` verknüpften Wert ab. Das obige Beispiel legt nur Schlüssel/Wert-Paare an, ohne Werte abzufragen. Das Abfragen von Werten wird in Abschnitt 12.10 gezeigt.

[41] Beachten Sie, daß Sie bei einem Container vom Typ *`Map`* keine Größe angeben oder sich darüber Gedanken machen müssen, da er seine Kapazität automatisch anpaßt. Ein Container vom Typ *`Map`* „weiß" wie sein Inhalt beim Ausdrucken dargestellt werden muß, um die Verbindung zwischen Schlüsseln und Werten zu zeigen. Die Anordnung der Schlüssel und Werte in einem Container vom Typ *`Map`* hat nichts mit der Einsetzungsreihenfolge zu tun, da der Containertyp `HashMap` die Anordnung über einen sehr schnellen Algorithmus steuert.

[42] Das obige Beispiel verwendet die drei wichtigsten Varianten des Containertyps *`Map`*: `HashMap`, `TreeMap` und `LinkedHashMap`. Ein `HashMap`-Container implementiert (wie `HashSet`) das schnellste Abfrageverfahren und speichert seine Elemente ebenfalls in keiner erkennbaren Reihenfolge. Ein `TreeMap`-Container speichert seine Schlüssel in aufsteigender Reihenfolge. Ein `LinkedHashMap`-Container erhält die Einsetzungsreihenfolge der Schlüssel und die Zugriffsgeschwindigkeit des Typs `HashMap`.

Übungsaufgabe 4: (3) Schreiben Sie eine Generatorklasse, die bei jedem Aufruf ihrer `next()`-Methode den Namen einer Figur aus ihrem Lieblingsfilm in Form eines `String`-Objektes zurückgibt (nehmen Sie „Schneewittchen" oder „Star Wars", wenn Ihnen kein Film einfällt) und erneut beim Anfang der Liste beginnt, wenn der letzte Eintrag zurückgegeben wurde. Verwenden Sie diese Generatorklasse, um ein Array sowie Container der Typen `ArrayList`, `LinkedList`, `HashSet`, `LinkedHashSet` und `TreeSet` zu füllen und geben Sie jeden Container aus. ■

12.5 Listen: Das Interface List

[43] Listen speichern ihre Elemente in einer bestimmten Reihenfolge. Das Interface *`List`* erweitert *`Collection`* um eine Anzahl von Methoden, die das Eintragen und Entfernen von Elementen aus der Mitte einer Liste ermöglichen. Es gibt zwei Listentypen:

- Der grundlegende Typ `ArrayList` zeichnet sich dadurch aus, daß Sie auf jedes beliebige Element zugreifen können, ist aber beim Hinzufügen oder Entfernen von Elementen aus der Mitte der Liste langsamer.

- Der Typ `LinkedList` ist für den Zugriff auf ein Element nach dem anderen optimiert und gestattet effizientes Hinzufügen beziehungsweise Entfernen von Elementen in der Mitte der Liste. Beim Zugriff auf beliebige Elemente ist `LinkedList` relativ langsam, hat dafür aber einen größeren Funktionsumfang als `ArrayList`.

[44] Das folgende Beispiel greift auf Kapitel 15 vor, indem es das Package `typeinfo.pets` importiert. Dieses Package enthält eine Hierarchie von Klassen für Haustiere (`Pet`) sowie Hilfsmittel, um Objekte von zufällig gewählten Unterklassen von `Pet` erzeugen zu können. Die Einzelheiten sind an dieser Stelle noch nicht von Bedeutung. Es genügt, wenn Sie sich merken, daß es eine Klasse `Pet` sowie verschiedene davon abgeleitete Klassen gibt und daß die statische `Pets`-Methode `arrayList()` einen Container vom Typ `ArrayList` zurückgibt, der mit Objekten zufällig gewählter Unterklassen von `Pet` gefüllt ist:

```
//: holding/ListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ListFeatures {
    public static void main(String[] args) {
        Random rand = new Random(47);
        List<Pet> pets = Pets.arrayList(7);
        print("1: " + pets);
        Hamster h = new Hamster();
        pets.add(h); // Automatically resizes
        print("2: " + pets);
        print("3: " + pets.contains(h));
        pets.remove(h); // Remove by object
        Pet p = pets.get(2);
        print("4: " + p + " " + pets.indexOf(p));
        Pet cymric = new Cymric();
        print("5: " + pets.indexOf(cymric));
        print("6: " + pets.remove(cymric));
        // Must be the exact object:
        print("7: " + pets.remove(p));
        print("8: " + pets);
        pets.add(3, new Mouse()); // Insert at an index
        print("9: " + pets);
        List<Pet> sub = pets.subList(1, 4);
        print("subList: " + sub);
        print("10: " + pets.containsAll(sub));
        Collections.sort(sub); // In-place sort
        print("sorted subList: " + sub);
        // Order is not important in containsAll():
        print("11: " + pets.containsAll(sub));
        Collections.shuffle(sub, rand); // Mix it up
        print("shuffled subList: " + sub);
        print("12: " + pets.containsAll(sub));
        List<Pet> copy = new ArrayList<Pet>(pets);
        sub = Arrays.asList(pets.get(1), pets.get(4));
        print("sub: " + sub);
        copy.retainAll(sub);
        print("13: " + copy);
        copy = new ArrayList<Pet>(pets); // Get a fresh copy
        copy.remove(2); // Remove by index
        print("14: " + copy);
        copy.removeAll(sub); // Only removes exact objects
```

```
        print("15: " + copy);
        copy.set(1, new Mouse()); // Replace an element
        print("16: " + copy);
        copy.addAll(2, sub); // Insert a list in the middle
        print("17: " + copy);
        print("18: " + pets.isEmpty());
        pets.clear(); // Remove all elements
        print("19: " + pets);
        print("20: " + pets.isEmpty());
        pets.addAll(Pets.arrayList(4));
        print("21: " + pets);
        Object[] o = pets.toArray();
        print("22: " + o[3]);
        Pet[] pa = pets.toArray(new Pet[0]);
        print("23: " + pa[3].id());
    }
} /* Output:
      1: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug]
      2: [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Hamster]
      3: true
      4: Cymric 2
      5: -1
      6: false
      7: true
      8: [Rat, Manx, Mutt, Pug, Cymric, Pug]
      9: [Rat, Manx, Mutt, Mouse, Pug, Cymric, Pug]
      subList: [Manx, Mutt, Mouse]
      10: true
      sorted subList: [Manx, Mouse, Mutt]
      11: true
      shuffled subList: [Mouse, Manx, Mutt]
      12: true
      sub: [Mouse, Pug]
      13: [Mouse, Pug]
      14: [Rat, Mouse, Mutt, Pug, Cymric, Pug]
      15: [Rat, Mutt, Cymric, Pug]
      16: [Rat, Mouse, Cymric, Pug]
      17: [Rat, Mouse, Mouse, Pug, Cymric, Pug]
      18: false
      19: []
      20: true
      21: [Manx, Cymric, Rat, EgyptianMau]
      22: EgyptianMau
      23: 14
   *///:~
```

[45] Die Ausgabezeilen sind nummeriert, damit Sie die Anweisung im Quelltext und die zugehörige Ausgabe identifizieren können. Die erste Ausgabezeile zeigt den Inhalt der ursprünglichen `Pet`-Liste. Im Gegensatz zu einem Array können Sie eine Liste nach ihrer Erzeugung um zusätzliche Elemente erweitern oder Elemente entfernen, wobei sich die Länge der Liste automatisch anpaßt. Die Fundamentaleigenschaft einer Liste besteht darin, eine veränderliche Aneinanderreihung von Elementen zu sein. Ausgabezeile 2 zeigt was beim Eintragen eines `Hamster`-Objektes geschieht: Das Objekt wird an das Ende der Liste angefügt.

[46] Die `contains()`-Methode ermittelt, ob eine Liste ein bestimmtes Element enthält. Wenn Sie ein Objekt entfernen möchten, können Sie der Methode `remove()` die Referenz auf dieses Objekt übergeben. Die `indexOf()`-Methode gibt die Position eines Elementes in der Liste an, wenn sie mit

der Referenz auf das entsprechende Objekt aufgerufen wird (Ausgabezeile 4).

[47] Wenn Sie eine Referenz übergeben, um zu ermitteln, ob eine Liste ein bestimmtes Element enthält, an welcher Position ein Element in einer Liste steht oder um ein Element aus einer Liste zu entfernen, kommt die von der Wurzelklasse `Object` ererbte Methode `equals()` ins Spiel. Jedes `Pet`-Objekt ist ein eindeutiges Objekt, das heißt auch wenn die Liste bereits zwei `Cymric`-Objekte enthält, ich ein weiteres `Cymric`-Objekt erzeuge und seine Referenz der `indexOf()`-Methode übergebe, gibt die Methode -1 zurück (der Rückgabewert -1 zeigt an, daß das gesuchte Element nicht gefunden wurde) und die `remove()`-Methode liefert `false`. Bei anderen Klassen kann die `equals()`-Methode abweichend definiert sein, beispielsweise sind zwei `String`-Objekt identisch, wenn sie inhaltlich übereinstimmen. Um Überraschungen zu vermeiden, ist es wichtig, zu verstehen, daß das Verhalten einer Liste vom Verhalten der `equals()`-Methode ihrere Elemente abhängt.

[48] Die Ausgabezeile 7 und 8 belegen, daß ein Objekt, welches exakt mit einem Element der Liste übereinstimmt aus der Liste entfernt werden kann.

[49] Es ist möglich, ein Element in der Mitte der Liste einzusetzen (Ausgabezeile 9) und stellt eine Eigenschaft von Listen in den Vordergrund: Bei einer Liste vom Typ `LinkedList` ist das Einsetzen und Entfernen von Elementen aus der Mitte der Liste eine effiziente Operation (nicht aber der Zugriff auf Elemente *in der Mitte*), bei einer Liste vom Typ `ArrayList` dagegen eine teuere Operation. Sollten Sie also niemals Elemente in der Mitte einer Liste vom Typ `ArrayList` einsetzen und im Bedarfsfall auf den Typ `LinkedList` umsteigen? Nein. Sie sollten sich aber dieser Eigenschaft bewußt sein. Wenn Sie aber bei vielen Einsetzungsoperationen in der Mitte einer Liste vom Typ `ArrayList` beobachten, daß Ihr Programm langsam wird, könnte sich der gewählte Listentyp als Ursache herausstellen (die beste Vorgehensweise, um einen solchen Engpaß zu entdecken besteht darin, einen Profiler zu verwenden; siehe *http://www.mindview.net/Books/BetterJava*). Optimierung ist eine komplizierte Angelegenheit und der beste Ansatz lautet, solange darauf zu verzichten, bis sich herausstellt, daß Sie sich darüber Gedanken machen müssen (es ist trotzdem sinnvoll, sich mit dem Thema auseinanderzusetzen).

[50] Die Methode `subList()` erzeugt eine Teilliste einer größeren Liste, für welche die `containsAll()`-Methode natürlich `true` liefert. Interessanterweise ist dabei die Reihenfolge der Elemente unwichtig: Die Ausgabezeilen 11 und 12 zeigen, daß die intuitiv benannten statischen `Collections`-Methoden `sort()` und `shuffle()`, auf die von `sub` referenzierte Teilliste angewandt, den Rückgabewert von `containsAll()` nicht beeinflussen. Die `subList()`-Methode erzeugt eine Liste, die direkt mit der ursprünglichen Liste verknüpft ist. Änderungen an der Teilliste schlagen sich in der ursprünglichen Liste nieder und umgekehrt.

[51] Die Methode `retainAll()` implementiert die Bildung einer Schnittmenge. Hier bleiben alle Elemente aus `copy` erhalten, die auch in der von `sub` referenzierten Liste liegen. Auch hier ist das resultierende Verhalten von der `equals()`-Methode abhängig.

[52] Ausgabezeile 14 zeigt das Entfernen eines Elementes aus der Liste anhand seines Index. Diese Vorgehensweise ist einfacher, als das Entfernen eines Elementes per Objektreferenz, da Sie sich beim Index keine Gedanken über das Verhalten der `equals()`-Methode machen müssen.

[53] Auch das Verhalten der Methode `removeAll()` hängt von `equals()` ab. Die Methode entfernt alle Elemente ihrer Liste, die zu der als Argument übergebenen Liste gehören.

[54] Die Benennung der `set()`-Methode ist aufgrund der möglichen Verwechslung mit dem Containertyp *`Set`* unglücklich. Der Name „`replace()`“ wäre besser gewesen, da die `set()`-Methode das Element an der Indexposition (erstes Argument) durch ihr zweites Argument ersetzt.

[55–56] Ausgabezeile 17 zeigt, daß die `addAll()`-Methode bei Containern vom Typ *`List`* überladen ist, wobei eine Version gestattet, eine neue Liste in der Mitte der ursprünglichen Liste einzusetzen,

statt wie bei der `addAll()`-Version aus *`Collection`* nur an das Ende anzufügen. Die Ausgabezeilen 18–20 demonstrieren die Wirkung der Methoden `isEmpty()` und `clear()`.

[57] Die Ausgabezeilen 22 und 23 zeigen, wie Sie einen beliebigen Container vom Typ *`Collection`* per `toArray()` in ein Array umwandeln können. Die `toArray()`-Methode ist überladen. Die argumentlose Version gibt ein `Object`-Array zurück. Wenn Sie der Methode ein Array des Zieltyps übergeben, liefert die Methode ein Array des angegebenen Typs (vorausgesetzt, die Typprüfung nicht scheitert). Ist das als Argument übergebene Array zu klein, um alle Elemente der Liste aufzunehmen (wie im obigen Beispiel), so erzeugt `toArray()` ein neues Array passender Länge. `Pet`-Objekte haben eine `id()`-Methode, die hier auf allen Elementen des resultierenden Arrays aufgerufen wird.

Übungsaufgabe 5: (3) Ändern Sie das Beispiel *ListFeatures.java* so, daß es `Integer`- statt `Pet`-Objekten verwendet werden (denken Sie an den Autoboxingmechanismus) und erläutern Sie die Unterschiede in den Ergebnissen. ■

Übungsaufgabe 6: (2) Ändern Sie das Beispiel *ListFeatures.java* so, daß `String`- statt `Pet`-Objekten verwendet werden und erläutern Sie die Unterschiede in den Ergebnissen. ■

Übungsaufgabe 7: (3) Legen Sie eine Klasse und ein initialisiertes Array von Objekten dieser Klasse an. Füllen Sie eine Liste aus diesem Array. Erzeugen Sie per `subList()`-Methode eine Teilliste der ursprünglichen Liste und entfernen Sie die Teilliste als der ursprünglichen Liste. ■

12.6 Iteratoren: Das Interface Iterator

[58] Ein Container muß Funktionalität bieten, um Elemente einzusetzen und wieder zu entnehmen, schließlich besteht seine Aufgabe hauptsächlich darin, Elemente zu enthalten. Ein Container vom Typ *`List`* bietet zum Beispiel die Methoden `add()` zum Einsetzen und `get()` zum Entnehmen von Elementen.

[59] Von einer höheren Ebene aus betrachtet, hat dieser Ansatz einen Nachteil: Sie beziehen sich stets auf den exakten Interfacetyp des Containers. Der Nachteil ist auf den ersten Blick vielleicht nicht klar zu erkennen, aber was tun Sie, wenn Sie einen Container vom Typ *`List`* implementiert haben und später entdecken, daß es praktisch wäre, dieselbe Funktionalität auch auf einen Container vom Typ *`Set`* anwenden zu können? Oder stellen Sie sich vor, Sie entwickeln von Anfang an ein Stück universellen Quelltext, der nicht „weiß“ beziehungsweise sich nicht darum kümmert, auf welchen Containertyp er angewendet werden wird, sich also ohne Änderungen für verschiedenartige Container eignet?

[60] Das Konzept des Iterators (Entwurfmuster *Iterator*) liefert diese Abstraktionsstufe. Ein Iterator ist ein Objekt, dessen Aufgabe darin besteht, eine Reihe von Elementen zu überstreichen und ein Element nach dem anderen auszuwählen, ohne daß der Programmierer die dieser Reihe unterliegenden Struktur kennen oder sich darum kümmern muß. Iteratoren gehören zu den üblicherweise als *leichtgewichtig* bezeichneten Objekten, das heißt ihre Erzeugung erfordert wenig Aufwand. Iteratoren unterliegen daher häufig scheinbar seltsamen Einschränkungen. Ein Iterator vom Java-Typ *`Iterator`* kann sich beispielsweise nur in eine Richtung bewegen. Die Funktionalität von Iteratoren ist beschränkt:

- Ein Container vom Typ *`Collection`* gestattet Ihnen, über die Methode `iterator()` einen Iterator anzufordern. Der Iterator ist bereit, um das erste Element der Reihe zu überstreichen.
- Die `next()`-Methode gibt das zuletzt überstrichene Objekt der Reihe zurück.
- Die `hasNext()`-Methode gibt an, ob die Reihe noch ein weiteres Objekt enthält.

- Die `remove()`-Methode entfernt das zuletzt überstrichene Element aus der Reihe.

[61] Das folgende Beispiel führt die Funktionsweise eines Iterators vor und bedient sich wiederum der Hilfsklasse `Pets` aus Kapitel 15:

```
//: holding/SimpleIteration.java
import typeinfo.pets.*;
import java.util.*;

public class SimpleIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(12);
        Iterator<Pet> it = pets.iterator();
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
        // A simpler approach, when possible:
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
        // An Iterator can also remove elements:
        it = pets.iterator();
        for(int i = 0; i < 6; i++) {
            it.next();
            it.remove();
        }
        System.out.println(pets);
    }
} /* Output:
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat
        10:EgyptianMau 11:Hamster
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx 8:Cymric 9:Rat
        10:EgyptianMau 11:Hamster
      [Pug, Manx, Cymric, Rat, EgyptianMau, Hamster]
   *///:~
```

[62] Mit einem Iterator brauchen Sie sich nicht um die um die Anzahl der Elemente eines Containers zu kümmern. Das übernehmen die Methoden `hasNext()` und `next()` für Sie.

[63] Wenn Sie die Liste einfach nur in Vorwärtsrichtung durchlaufen und den Container selbst nicht modifizieren müssen, ist die erweitere `for`-Schleife kürzer und bündiger.

[64] Die im Interface *`Iterator`* deklarierte Methode `remove()` entfernt das zuletzt überstrichene Element, das heißt Sie müssen erst `next()` aufrufen und danach `remove()`.[4]

[65] Die Idee, einen Container von Objekten zu nehmen und zu durchlaufen, wobei auf jedem Element eine Operation ausgeführt wird, ist mächtig und tritt in diesem Buch an vielen Stellen auf.

[66] Das folgende Beispiel zeigt eine `display()`-Methode, die den unterliegenden Container nicht wahrnimmt:

```
//: holding/CrossContainerIteration.java
import typeinfo.pets.*;
```

[4] `remove()` ist eine sogenannte „optionale“ Methode (es gibt weitere dieser Methoden), das heißt, nicht jede Implementierung des *`Iterator`*-Interfaces muß diese Methode *mit Funktionalität* ausprogrammieren (siehe Beispiel *CollectionSequence.java*, Seite 340). Dieses Thema wird in Kapitel 18 behandelt. Die Containerklassen der Standardbibliothek von Java implementieren `remove()`, aber Sie brauchen sich vor Kapitel 18 nicht den Kopf zu zerbrechen.

```
import java.util.*;

public class CrossContainerIteration {
    public static void display(Iterator<Pet> it) {
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }
    public static void main(String[] args) {
        ArrayList<Pet> pets = Pets.arrayList(8);
        LinkedList<Pet> petsLL = new LinkedList<Pet>(pets);
        HashSet<Pet> petsHS = new HashSet<Pet>(pets);
        TreeSet<Pet> petsTS = new TreeSet<Pet>(pets);
        display(pets.iterator());
        display(petsLL.iterator());
        display(petsHS.iterator());
        display(petsTS.iterator());
    }
} /* Output:
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
      4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
      5:Cymric 2:Cymric 7:Manx 1:Manx 3:Mutt 6:Pug 4:Pug 0:Rat
   *///:~
```

[67] Beachten Sie, daß die Methode `display()` keinerlei Information über den Typ der traversierten Reihe enthält und somit die wahre Kraft des Iteratorkonzeptes zeigt: Die Fähigkeit, die Operation des Traversierens einer Reihe von Elementen von der unterliegenden Struktur dieser Reihe zu lösen. Daher sagen wir zuweilen, daß Iteratoren den *Zugriff auf Container vereinheitlichen*.

Übungsaufgabe 8: (1) Ändern Sie Übungsaufgabe 1 (Seite 313) so, daß das Programm einen Iterator verwendet, um die Liste zu durchlaufen und die `hop()`-Methode auf jedem Element aufzurufen. ■

Übungsaufgabe 9: (4) Ändern Sie das Beispiel *innerclasses/Sequence.java* (Seite **??**) so, daß die Klasse `Sequence` statt eines Selektors (Klasse *`Selector`*) mit einem Iterator arbeitet. ■

Übungsaufgabe 10: (2) Ändern Sie Übungsaufgabe 9 aus Kapitel 9 (Seite 231) so, daß das Programm einen Container vom Typ `ArrayList` verwendet, um die `Rodent`-Objekt zu speichern und einen Iterator, um die Reihe der `Rodent`-Objekte zu durchlaufen. ■

Übungsaufgabe 11: (2) Schreiben Sie eine Methode, die einen Iterator verwendet, um einen Container vom Typ *`Collection`* elementweise zu verarbeiten und die `toString()`-Methode jedes in dem Container gespeicherten Objektes aufzurufen. Füllen Sie Container der Typen unter dem Interface *`Collection`* mit Elementen und wenden Sie die Methode auf jeden Container an. ■

12.6.1 Bidirektionale Iteratoren (Listeniteratoren): Das Interface ListIterator

[68] Ein Iterator vom Typ *`ListIterator`* („Listeniterator“) ist leistungsfähiger als ein Iterator dessen Klasse nur das Interface *`Iterator`* implementiert und kann nur bei Containern vom Typ *`List`* angefordert werden. Während sich Iteratoren vom Typ *`Iterator`* nur vorwärts bewegen können, sind Listeniteratoren in der Lage, bidirektional zu arbeiten. Ein Listeniterator kann zusätzlich die Indices des nächsten und des vorigen Elementes relativ zu seiner aktuellen Position zurückgeben und das zuletzt überstrichene Element austauschen. Die Methode `listIterator()` fordert einen Listen-

iterator an, der auf den Anfang der Liste (*vor* dem ersten Element) zeigt, während `listIterator(n)` den Listeniterator vor das Element mit dem Index n setzt. Das folgende Beispiel demonstriert diese Fähigkeiten:

```
//: holding/ListIteration.java
import typeinfo.pets.*;
import java.util.*;

public class ListIteration {
    public static void main(String[] args) {
        List<Pet> pets = Pets.arrayList(8);
        ListIterator<Pet> it = pets.listIterator();
        while(it.hasNext())
            System.out.print(it.next() + ", " + it.nextIndex() +
                             ", " + it.previousIndex() + "; ");
        System.out.println();
        // Backwards:
        while(it.hasPrevious())
            System.out.print(it.previous().id() + " ");
        System.out.println();
        System.out.println(pets);
        it = pets.listIterator(3);
        while(it.hasNext()) {
            it.next();
            it.set(Pets.randomPet());
        }
        System.out.println(pets);
    }
} /* Output:
      Rat, 1, 0; Manx, 2, 1; Cymric, 3, 2; Mutt, 4, 3; Pug, 5, 4; Cymric, 6, 5; Pug,
7, 6; Manx, 8, 7;
      7 6 5 4 3 2 1 0
      [Rat, Manx, Cymric, Mutt, Pug, Cymric, Pug, Manx]
      [Rat, Manx, Cymric, Cymric, Rat, EgyptianMau, Hamster, EgyptianMau]
   *///:~
```

[69] Die statische `Pets`-Methode `randomPet()` wird aufgerufen, um alle `Pet`-Objekte ab Position 3 in der Liste zu ersetzen.

Übungsaufgabe 12: (3) Erzeugen und füllen Sie einen Container vom Typ *`List<Integer>`*. Erzeugen Sie einen zweiten Container dieses Typs derselben Größe und verwenden Sie einen Listeniterator, um die Elemente der ersten Liste abzufragen und in umgekehrter Reihenfolge in die zweite Liste einzusetzen. (Probieren Sie verschiedene Möglichkeiten aus, um diese Aufgabe zu lösen.) ■

12.7 Verkettete Listen: Die Klasse LinkedList

[70] Der Containertyp `LinkedList` implementiert wie `ArrayList` das grundlegende *`List`*-Interface und ist bei bestimmten Operationen (Einsetzen und Entfernen von Elementen aus der Mitte der Liste) effizienter als `ArrayList`. Andererseits ist `LinkedList` beim Zugriff auf beliebige Positionen langsamer als `ArrayList`.

[71–76] Die Klasse `LinkedList` definiert außerdem Methoden, mit deren Hilfe ein Container dieses Typs als Stapelspeicher (*stack*), Warteschlange (*queue*) oder als doppelköpfige Warteschlange (*deque*) verwendet werden kann. Einige dieser Methoden sind funktionell identisch oder unterscheiden sich nur geringfügig voneinander, haben aber verschiedene, an den jeweiligen Anwendungskontext

angepaßte Namen (besonders beim Containertyp *Queue*). Beispielsweise sind die Methoden `getFirst()` und `element()` identisch, geben das Kopfelement (das erste Element) der Liste zurück, ohne es zu entfernen und werfen eine Ausnahme vom Typ `NoSuchElementException` aus, wenn die Liste leer ist. Die Methoden `removeFirst()` und `remove()` sind ebenfalls identisch, entfernen das Kopfelement der Liste und geben es zurück. Enthält die Liste keine Elemente so rufen beide Methoden eine Ausnahme vom Typ `NoSuchElementException` hervor. Die Methode `poll()` ist eine kleine Variation der Methoden `removeFirst()` und `remove()` und gibt bei leerer Liste `null` zurück. Die Methode `addFirst()` setzt ein Element am Anfang der Liste ein. Die Methode `offer()` ist mit `add()` und `addLast()` identisch und fügt ein Element an das Ende der Liste an. Die Methode `removeLast()` entfernt das letzte Element der Liste und gibt es zurück.

[77] Das folgende Beispiel zeigt die grundsätzlichen Ähnlichkeiten beziehungsweise Unterschiede zwischen diesen Methoden. Das im Beispiel *ListFeatures.java* vorgeführte Verhalten wird nicht wiederholt:

```
//: holding/LinkedListFeatures.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class LinkedListFeatures {
    public static void main(String[] args) {
        LinkedList<Pet> pets = new LinkedList<Pet>(Pets.arrayList(5));
        print(pets);
        // Identical:
        print("pets.getFirst(): " + pets.getFirst());
        print("pets.element(): " + pets.element());
        // Only differs in empty-list behavior:
        print("pets.peek(): " + pets.peek());
        // Identical; remove and return the first element:
        print("pets.remove(): " + pets.remove());
        print("pets.removeFirst(): " + pets.removeFirst());
        // Only differs in empty-list behavior:
        print("pets.poll(): " + pets.poll());
        print(pets);
        pets.addFirst(new Rat());
        print("After addFirst(): " + pets);
        pets.offer(Pets.randomPet());
        print("After offer(): " + pets);
        pets.add(Pets.randomPet());
        print("After add(): " + pets);
        pets.addLast(new Hamster());
        print("After addLast(): " + pets);
        print("pets.removeLast(): " + pets.removeLast());
    }
} /* Output:
      [Rat, Manx, Cymric, Mutt, Pug]
      pets.getFirst(): Rat
      pets.element(): Rat
      pets.peek(): Rat
      pets.remove(): Rat
      pets.removeFirst(): Manx
      pets.poll(): Cymric
      [Mutt, Pug]
      After addFirst(): [Rat, Mutt, Pug]
      After offer(): [Rat, Mutt, Pug, Cymric]
      After add(): [Rat, Mutt, Pug, Cymric, Pug]
```

```
        After addLast(): [Rat, Mutt, Pug, Cymric, Pug, Hamster]
        pets.removeLast(): Hamster
    *///:~
```

[78] Der Rückgabewert der statischen `Pets`-Methode `arrayList()` wird dem Konstruktor der Klasse `LinkedList` übergeben, um das erzeugte Objekt zu füllen. Das Interface *`Queue`* deklariert die Methoden `element()`, `offer()`, `peek()`, `poll()` und `remove()`, welche von der Klasse `LinkedList` implementiert werden, um einen Container dieses Typs auch als Warteschlange nutzen zu können. Sie finden in Abschnitt 12.11 komplette Beispiele für Warteschlangen.

Übungsaufgabe 13: (3) Die Klasse *`Controller`* im Beispiel *innerclasses/GreenhouseController-.java* (Seite 302) verwendet einen Container vom Typ `ArrayList`. Ändern Sie das Programm, so daß es mit einem Container vom Typ `LinkedList` arbeitet und einen Iterator verwendet, um die Ereignisse eines nach dem anderen auszuwählen. ■

Übungsaufgabe 14: (3) Legen Sie einen leeren Container vom Typ `LinkedList<Integer>` an und verwenden Sie einen Listeniterator, um einige `Integer`-Objekte stets in der Mitte der Liste einzusetzen. ■

12.8 Stapelspeicher: Die Klasse Stack

[79] Der Stapelspeicher wird gelegentlich als „last-in, first-out"-Speicher (LIFO-Speicher) bezeichnet. In der englischsprachigen Literatur ist auch der Begriff „Pushdown Stack" gebräuchlich, da das letzte im Speicher deponierte Element zugleich das erste ist, welches Sie dem Speicher wieder entnehmen können. Ein häufig verwendeter Vergleich ist der Tellerstapel in einer Kantine oder Cafeteria: Der letzte eingelegte Teller wird zuerst entnommen.

[80] Ein Container vom Typ `LinkedList` implementiert die Funktionalität eines Stapelspeichers direkt. Sie können also einen solchen Container wählen, statt eine eigene Stapelspeicherklasse zu implementieren. Andererseits drückt eine echte `Stack`-Klasse unter Umständen besser aus, worum es geht:

```
//: net/mindview/util/Stack.java
// Making a stack from a LinkedList.
package net.mindview.util;
import java.util.LinkedList;

public class Stack<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void push(T v) { storage.addFirst(v); }
    public T peek() { return storage.getFirst(); }
    public T pop() { return storage.removeFirst(); }
    public boolean empty() { return storage.isEmpty(); }
    public String toString() { return storage.toString(); }
} ///:~
```

[81] Dieses Beispiel ist die einfachste Variante der Definition einer Klasse mit generischem Typparameter. Das Symbol `<T>` nach dem Klassennamen kennzeichnet die Klasse dem Compiler gegenüber als *parametrisierten Typ* und definiert `T` als sogenannten *Typparameter*, der beim Erzeugen eines Objektes dieser Klassen durch einen wirklichen *Parametertyp* ersetzt wird. Die Definition sagt im Grunde aus, daß eine Klasse `Stack` definiert wird, die Objekte vom Typ `T` speichert. Die Stapelspeicherfunktionalität wird mit Hilfe eines Containers vom Typ `LinkedList` implementiert, der bei der Objekterzeugung ebenfalls auf Elemente vom Typ `T` festgelegt wird. Beachten Sie, daß `push()` ein Argument vom Typ `T` erwartet, während `peek()` und `pop()` ein Objekt vom Typ `T` zurückgeben.

Die `peek()`-Methode gibt das oberste Element zurück, ohne es aus dem Speicher zu entfernen, die `pop()`-Methode gibt das oberste Element zurück und löscht es zugleich.

[82] Wenn Sie reine Stapelspeicherfunktionalität möchten, ist die Ableitung von `LinkedList` ungeeignet, weil die abgeleitete Klasse auch alle übrigen Methoden der Basisklasse `LinkedList` erben würde. (In Kapitel 18 lernen Sie, daß die Designer von Java 1.0 bei der Definition der Klasse `java.util.Stack` genau diesen Fehler gemacht haben.)

[83] Das nächste Beispiel zeigt, wie die obige `Stack`-Klasse funktioniert:

```
//: holding/StackTest.java
import net.mindview.util.*;

public class StackTest {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(String s : ''My dog has fleas''.split('' ''))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + '' '');
    }
} /* Output:
      fleas has dog My
   *///:~
```

[84] Wenn Sie diese `Stack`-Klasse in einem Ihrer Programme verwenden möchten, müssen Sie den vollständigen Packagepfad angeben oder den Klassennamen ändern, da es andernfalls zu einer Namenskollision mit der Klasse `java.util.Stack` kommt. Importieren wir beispielsweise im obigen Beispiel das Package `java.util`, so müssen wir unsere `Stack`-Klasse mit Packagenamen verwenden, um Kollisionen vorzubeugen:

```
//: holding/StackCollision.java
import net.mindview.util.*;

public class StackCollision {
    public static void main(String[] args) {
        net.mindview.util.Stack<String> stack =
            new net.mindview.util.Stack<String>();
        for(String s : ''My dog has fleas''.split('' ''))
            stack.push(s);
        while(!stack.empty())
            System.out.print(stack.pop() + '' '');
        System.out.println();
        java.util.Stack<String> stack2 =
            new java.util.Stack<String>();
        for(String s : ''My dog has fleas''.split('' ''))
            stack2.push(s);
        while(!stack2.empty())
            System.out.print(stack2.pop() + '' '');
    }
} /* Output:
      fleas has dog My
      fleas has dog My
   *///:~
```

[85] Beide `Stack`-Klassen haben dieselbe Schnittstelle, aber es gibt kein gemeinsames *`Stack`*-Interface im Package `java.util`, vermutlich weil die ursprüngliche, ärmlich entworfene Klasse `java.util.Stack` den Namen bereits belegt. Die Klasse `LinkedList` liefert eine bessere `Stack`-Klasse als `java.util.Stack`, so daß `net.mindview.util.Stack` vielleicht vorzuziehen ist.

[86] Sie können die Wahl Ihrer bevorzugten `Stack`-Klasse auch durch explizites Importieren steuern:

```
import net.mindview.util.Stack;
```

Nun wird jedes Vorkommen von `Stack` mit der Version im Package `net.mindview.util` identifiziert und Sie müssen den Packagennamen hinzufügen, um `java.util.Stack` zu wählen.

Übungsaufgabe 15: (4) Stapelspeicher werden häufig verwendet, um Ausdrücke von Programmiersprachen auszuwerten. Evaluieren Sie mit Hilfe der Klasse `net.mindview.util.Stack` den folgenden Ausdruck:

```
+U+n+c---+e+r+t---+a-+i-+n+t+y---+ -+r+u--+l+e+s---
```

Ein Pluszeichen bedeutet: „Lege den folgenden Buchstaben auf dem Stapelspeicher ab.“ Ein Minuszeichen bedeutet: „Hole das oberste Zeichen vom Stapelspeicher und gibt es aus.“ ■

12.9 Mengen: Das Interface Set

[87] Ein Container vom Typ *`Set`* weigert sich, mehr als ein Exemplar eines Elementes aufzunehmen. Wenn Sie versuchen, mehr als ein äquivalentes Objekt zu speichern, verhindert der Container das Speichern von Duplikaten. Der häufigste Anwendungsfall von Containern vom Typ *`Set`* ist die Prüfung eines Elementes auf Zugehörigkeit zu einer Menge. Daher ist das Nachschlagen typischerweise die wichtigste Operation eines solchen Containers und Sie wählen in der Regel die für schnelles Nachschlagen optimierte Implementierung `HashSet`.

[88] Die Interfaces *`Set`* und *`Collection`* deklarieren dieselben Methoden, das heißt *`Set`* erweitert *nicht* die Funktionalität von *`Collection`*, wie die Implementierungen des Interfaces *`List`* (`ArrayList` und `LinkedList`), beschreiben aber *unterschiedliches Verhalten*. (Der Ausdruck verschiedenen Verhaltens ist die ideale Verwendung von Ableitung und Polymorphie.) Ein Container vom Typ *`Set`* ermittelt die Zugehörigkeit aufgrund des „Wertes“ eines Objektes. Dieses komplizierte Thema wird in Kapitel 18 behandelt.

[89] Das folgende Beispiel zeigt einen Container vom Typ `HashSet` mit `Integer`-Objekten:

```
//: holding/SetOfInteger.java
import java.util.*;

public class SetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Set<Integer> intset = new HashSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* Output:
      [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 29, 14, 24, 4, 19, 26, 11, 18,
        3, 12, 27, 17, 2, 13, 28, 20, 25, 10, 5, 0]
   *///:~
```

[90] Dem Container werden zehntausend Zufallszahlen zwischen 0 und 29 übergeben. Sie können sich also vorstellen, daß jeder Wert zahlreiche Duplikate hat. Dennoch sehen Sie, daß jeder Wert nur einmal im Ergebnis auftritt.

[91] Beachten Sie, daß die Ausgabe keine erkennbare Reihenfolge hat. Die Klasse `HashSet` verwendet aus Geschwindigkeitsgründen einen *Hashalgorithmus* (siehe Kapitel 18). Die Reihenfolge der

Elemente in einem Container vom Typ `HashSet` unterscheidet sich von der Reihenfolge bei `TreeSet` und `LinkedHashSet`, da jede Implementierung ein anderes Verfahren zur Speicherung der Elemente verwendet. Die Klasse `TreeSet` speichert die Element sortiert in einer Rot-Schwarz-Baumstruktur, während `LinkedHashSet` ebenfalls einen Hashalgorithmus verwendet, die Elemente aber über eine verkettete Liste *scheinbar* in der Einsetzungsreihenfolge speichert.

[92] Wenn Sie ein sortiertes Ergebnis wünschen, können Sie statt `HashSet` den Containertyp `TreeSet` wählen:

```
//: holding/SortedSetOfInteger.java
import java.util.*;

public class SortedSetOfInteger {
    public static void main(String[] args) {
        Random rand = new Random(47);
        SortedSet<Integer> intset = new TreeSet<Integer>();
        for(int i = 0; i < 10000; i++)
            intset.add(rand.nextInt(30));
        System.out.println(intset);
    }
} /* Output:
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18,
        19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29]
   *///:~
```

[93] Eine der häufigsten Operationen, die Sie durchführen werden, ist die Prüfung auf Zugehörigkeit zu einer Menge per `contains()`-Methode, es gibt aber auch Operationen, die Sie an die Mengendiagramme (Venn-Diagramme) aus der Grundschule erinnern werden:

```
//: holding/SetOperations.java
import java.util.*;
import static net.mindview.util.Print.*;

public class SetOperations {
    public static void main(String[] args) {
        Set<String> set1 = new HashSet<String>();
        Collections.addAll(set1, "A B C D E F G H I J K L".split(" "));
        set1.add("M");
        print("H: " + set1.contains("H"));
        print("N: " + set1.contains("N"));
        Set<String> set2 = new HashSet<String>();
        Collections.addAll(set2, "H I J K L".split(" "));
        print("set2 in set1: " + set1.containsAll(set2));
        set1.remove("H");
        print("set1: " + set1);
        print("set2 in set1: " + set1.containsAll(set2));
        set1.removeAll(set2);
        print("set2 removed from set1: " + set1);
        Collections.addAll(set1, "X Y Z".split(" "));
        print("'X Y Z' added to set1: " + set1);
    }
} /* Output:
      H: true
      N: false
      set2 in set1: true
      set1: [D, K, C, B, L, G, I, M, A, F, J, E]
      set2 in set1: false
      set2 removed from set1: [D, C, B, G, M, A, F, E]
      'X Y Z' added to set1: [Z, D, C, B, G, M, A, F, Y, X, E]
```

```
*///:~
```

Die Methoden sind selbsterklärend. In der API-Dokumentation finden Sie noch einige weitere Methoden.

[94] Das Erzeugen einer Liste von eindeutigen Elementen kann recht nützlich sein. Stellen Sie sich zum Beispiel vor, Sie wollten eine Liste aller Wörter im obigen Beispiel *SetOperations.java* haben. Die in Unterabschnitt 19.7.1 beschriebene Hilfsklasse `TextFile` öffnet eine Textdatei und gibt den Dateiinhalt in einem Container vom Typ *`Set`* zurück:

```
//: holding/UniqueWords.java
import java.util.*;
import net.mindview.util.*;

public class UniqueWords {
    public static void main(String[] args) {
        Set<String> words =
            new TreeSet<String>(new TextFile(''SetOperations.java'', ''\\W+''));
        System.out.println(words);
    }
} /* Output:
      [A, B, C, Collections, D, E, F, G, H, HashSet, I, J, K, L, M, N, Output,
        Print, Set, SetOperations, String, X, Y, Z, add, addAll, added, args,
        class, contains, containsAll, false, from, holding, import, in, java,
        main, mindview, net, new, print, public, remove, removeAll, removed,
        set1, set2, split, static, to, true, util, void]
   *///:~
```

[95] Die Hilfsklasse `TextFile` ist von `ArrayList<String>` abgeleitet. Der Konstruktor von `TextFile` öffnet die Textdatei und trennt den Inhalt an den durch einen *regulären Ausdruck* definierten Grenzen auf. Der Ausdruck `\\W+` repräsentiert ein oder mehr „Nicht-Wortzeichen“. (Reguläre Ausdrücke werden in Kapitel 14 eingeführt.) Das Ergebnis wird dem Konstruktor der Klasse `TreeSet` übergeben, der den Inhalt an die Liste anfügt. Die Klasse `TreeSet` sortiert das Ergebnis. Im obigen Fall erfolgt die Sortierung nach *lexikographischer Ordnung*, so daß Groß- und Kleinbuchstaben in verschiedenen Gruppen liegen. Wenn Sie das Ergebnis *alphabetisch* sortieren möchten, übergeben Sie dem `TreeSet`-Konstruktor den Komparator `CASE_INSENSITIVE_ORDER` (ein Komparator ist ein Objekt, das eine Ordnung vermittelt):

```
//: holding/UniqueWordsAlphabetic.java
// Producing an alphabetic listing.
import java.util.*;
import net.mindview.util.*;

public class UniqueWordsAlphabetic {
    public static void main(String[] args) {
        Set<String> words = new TreeSet<String>(String.CASE_INSENSITIVE_ORDER);
        words.addAll(new TextFile(''SetOperations.java'', ''\\W+''));
        System.out.println(words);
    }
} /* Output:
      [A, add, addAll, added, args, B, C, class, Collections, contains,
        containsAll, D, E, F, false, from, G, H, HashSet, holding, I, import,
        in, J, java, K, L, M, main, mindview, N, net, new, Output, Print,
        public, remove, removeAll, removed, Set, set1, set2, SetOperations,
        split, static, String, to, true, util, void, X, Y, Z]
   *///:~
```

[96] Abschnitt 17.7 beschreibt Komparatoren in allen Einzelheiten.

Übungsaufgabe 16: (5) Legen Sie einen Container vom Typ *Set* an, der die Vokale enthält. Zählen Sie die Vokalen in jedem Wort des Beispiels *UniqueWords.java* sowie die Summen der Vokale in der Eingabedatei und geben Sie die Ergebnisse aus. ■

12.10 Assoziative Container: Das Interface Map

[97] Das Konzept, Objekte auf andere Objekte abzubilden, ist ein sehr wirkungsvoller Ansatz zur Lösung von Programmierproblemen. Stellen Sie zum Beispiel ein Programm vor, das die Qualität der von der Klasse `Random` erzeugten Zufallszahlen mißt. Im Idealfall erzeugt `Random` perfekt verteilte Zufallszahlen. Wir müssen zum Testen viele Zufallszahlen generieren und ihre Häufigkeiten zählen. Ein Container vom Typ *Map* hilft, das Problem auf einfache Weise zu lösen: Wir verwenden die Zufallszahl als Schlüssel und die Anzahl ihrer Vorkommen als Wert:

```
//: holding/Statistics.java
// Simple demonstration of HashMap.
import java.util.*;

public class Statistics {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Map<Integer,Integer> m = new HashMap<Integer,Integer>();
        for(int i = 0; i < 10000; i++) {
            // Produce a number between 0 and 20:
            int r = rand.nextInt(20);
            Integer freq = m.get(r);
            m.put(r, freq == null ? 1 : freq + 1);
        }
        System.out.println(m);
    }
} /* Output:
      {15=497, 4=481, 19=464, 8=468, 11=531, 16=533, 18=478, 3=508, 7=471,
        12=521, 17=509, 2=489, 13=506, 9=549, 6=519, 1=502, 14=477, 10=513,
        5=503, 0=481}
   *///:~
```

[98] In der `main()`-Methode wird zunächst der zufällig generierte `int`-Wert per Autoboxing in ein `Integer`-Objekt umgewandelt, das sich als Schlüssel des `HashMap`-Containers verwenden läßt (Container speichern keine Werte primitiven Typs). Die `get()`-Methode gibt `null` zurück, wenn der übergebene Schlüssel nicht bereits im Container angelegt ist (die entsprechende Zufallszahl also zum ersten Mal vorkommt). Andernfalls gibt `get()` eine Referenz auf das zugehörige `Integer`-Objekt (die Häufigkeit) zurück, dessen Wert inkrementiert wird (wiederum vereinfacht Autoboxing den Ausdruck, aber eigentlich finden Umwandlung von und nach `Integer` statt).

[99] Das folgende Beispiel gestattet, Haustiere (`Pet`-Objekte) anhand eines Beschreibungstextes (`String`-Objekt) abzufragen und zeigt außerdem, wie Sie mit den Methoden `containsKey()` und `containsValue()` prüfen können, ob ein Container vom Typ *Map* einen bestimmten Schlüssel beziehungsweise Wert enthält:

```
//: holding/PetMap.java
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetMap {
    public static void main(String[] args) {
```

```
        Map<String,Pet> petMap = new HashMap<String,Pet>();
        petMap.put("My Cat", new Cat("Molly"));
        petMap.put("My Dog", new Dog("Ginger"));
        petMap.put("My Hamster", new Hamster("Bosco"));
        print(petMap);
        Pet dog = petMap.get("My Dog");
        print(dog);
        print(petMap.containsKey("My Dog"));
        print(petMap.containsValue(dog));
    }
} /* Output:
      {My Cat=Cat Molly, My Hamster=Hamster Bosco, My Dog=Dog Ginger}
      Dog Ginger
      true
      true
   *///:~
```

[100] Container vom Typ *Map* können, wie Arrays und Container vom Typ *Collection*, mühelos auf mehrere Dimensionen erweitert werden, indem Sie einfach einen *Map*-Container erzeugen, dessen Werte wiederum *Map*-Container sind (deren Werte ebenfalls Container sein können, inbesondere vom Typ *Map*). Container lassen sich schnell und einfach zu mächtigen Datenstrukturen kombinieren. Das folgende Beispiel verwaltet Leute, die mehrere Haustiere haben in einem Container vom Typ *Map<Person, List<Pet>>*:

```
//: holding/MapOfList.java
package holding;
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class MapOfList {
    public static Map<Person, List<? extends Pet>>
        petPeople = new HashMap<Person, List<? extends Pet>>();
    static {
        petPeople.put(new Person("Dawn"),
            Arrays.asList(new Cymric("Molly"),new Mutt("Spot")));
        petPeople.put(new Person("Kate"),
            Arrays.asList(new Cat("Shackleton"),
                          new Cat("Elsie May"), new Dog("Margrett")));
        petPeople.put(new Person("Marilyn"),
            Arrays.asList(new Pug("Louie aka Louis Snorkelstein Dupree"),
                          new Cat("Stanford aka Stinky el Negro"),
                          new Cat("Pinkola")));
        petPeople.put(new Person("Luke"),
            Arrays.asList(new Rat("Fuzzy"), new Rat("Fizzy")));
        petPeople.put(new Person("Isaac"), Arrays.asList(new Rat("Freckly")));
    }
    public static void main(String[] args) {
        print("People: " + petPeople.keySet());
        print("Pets: " + petPeople.values());
        for(Person person : petPeople.keySet()) {
            print(person + " has:");
            for(Pet pet : petPeople.get(person))
                print(" " + pet);
        }
    }
} /* Output:
      People: [Person Luke, Person Marilyn, Person Isaac, Person Dawn, Person Kate]
```

```
      Pets: [[Rat Fuzzy, Rat Fizzy], [Pug Louie aka Louis Snorkelstein Dupree, Cat
        Stanford aka Stinky el Negro, Cat Pinkola], [Rat Freckly], [Cymric Molly,
Mutt
        Spot], [Cat Shackleton, Cat Elsie May, Dog Margrett]]
      Person Luke has:
      Rat Fuzzy
      Rat Fizzy
      Person Marilyn has:
      Pug Louie aka Louis Snorkelstein Dupree
      Cat Stanford aka Stinky el Negro
      Cat Pinkola
      Person Isaac has:
      Rat Freckly
      Person Dawn has:
      Cymric Molly
      Mutt Spot
      Person Kate has:
      Cat Shackleton
      Cat Elsie May
      Dog Margrett
   *///:~
```

[101] Ein Container vom Typ *Map* kann einen *Set*-Container aus seinen Schlüsseln, einen *Collection*-Container aus seinen Werten oder einen *Set*-Container aus seinen Schlüssel/Wert-Paaren zurückgeben. Die Methode `keySet()` liefert einen *Set*-Container aller Schlüssel des von `petPeople` referenzierten Containers, welcher in der erweiterten `for`-Schleife verwendet wird, um den Containerinhalt paarweise zu durchlaufen.

Übungsaufgabe 17: (1) Speichern Sie Objekte der Klasse `Gerbil` (Wüstenrennmaus) aus Übungsaufgabe 1 (Seite 313) in einem Container vom Typ *Map*. Verwenden Sie den Namen der Maus (ein `String`-Objekt, zum Beispiel „Fuzzy“ oder „Spot“) als Schlüssel und das `Gerbil`-Objekt als Wert. Fordern Sie einen Iterator über die Schlüsselmenge (`keySet()`) an und verwenden Sie ihn, um den Inhalt des Containers paarweise zu verarbeiten. Fragen Sie pro Schlüssel die zugehörige `Gerbil`-Referenz ab, geben Sie den Schlüssel aus und rufen Sie die `hop()`-Methode auf. ■

Übungsaufgabe 18: (3) Füllen Sie einen Container vom Typ `HashMap` mit Schlüssel/Wert-Paaren. Geben Sie den Containerinhalt aus, um die „Sortierung“ der Paare durch den Hashalgorithmus zu zeigen. Entnehmen Sie die gespeicherten Paare, sortieren Sie sie bezüglich ihrer Schlüssel und setzen Sie sie in einen Container vom Typ `LinkedHashMap` ein. Zeigen Sie, daß die Einsetzungsreihenfolge erhalten bleibt. ■

Übungsaufgabe 19: (2) Wiederholen Sie Übungsaufgabe 18 mit den Containertypen `HashSet` und `LinkedHashSet`. ■

Übungsaufgabe 20: (3) Ändern Sie Übungsaufgabe 16 (Seite 332) so, daß das Programm die Zeilennummern der Vorkommen der Vokale speichert. ■

Übungsaufgabe 21: (3) Verwenden Sie einen Container vom Typ *Map<String, Integer>*, um die Häufigkeiten der einzelnen Worte in einer Textdatei zu zählen (orientieren Sie sich am Beispiel *UniqueWords.java*). Sortieren Sie das Ergebnis mit der statischen `Collections`-Methode `sort()` mit `CASE_INSENSITIVE_ORDER` als zweitem Argument (bewirkt alphabetische Sortierung) und geben Sie das Ergebnis aus. ■

Übungsaufgabe 22: (5) Ändern Sie Übungsaufgabe 21 so, daß das Programm eine Klasse mit einem `String`- und einem Zählerfeld verwendet, um die verschiedenen Wörter zu speichern. Verwenden Sie einen *Set*-Container, um die Wörterliste in Form dieser Objekte zu speichern. ■

Übungsaufgabe 23: (4) Schreiben Sie ein Programm, das den Test im Beispiel *Statistics.java* mehrmals durchführt und prüft, ob eine Zahl häufiger als die anderen vorzukommen scheint. ■

Übungsaufgabe 24: (2) Füllen Sie einen Container vom Typ `LinkedHashMap` mit Schlüsseln vom Typ `String` und Objekten Ihrer Wahl. Entnehmen Sie die Paare, sortieren Sie sie bezüglich ihrer Schlüssel und setzen Sie sie erneut in den `LinkedHashMap`-Container ein. ■

Übungsaufgabe 25: (3) Legen Sie einen Container vom Typ *`Map<String, ArrayList<Integer>>`* an. Öffnen Sie mit der Hilfsklasse `TextFile` eine Textdatei und lesen Sie sie Wort für Wort ein (übergeben Sie dem `TextFile`-Konstruktor `\\W+` als zweites Argument. Zählen Sie die Worte beim Einlesen und zeichnen Sie den Zählerstand jeweils in dem mit diesem Wort verknüpften `ArrayList<Integer>`-Container auf (diese Information entspricht der Stelle des Vorkommens in der Datei). ■

Übungsaufgabe 26: (4) Rekonstruieren Sie die Reihenfolge der Worte in der ursprünglichen Datei aus dem Ergebnis von Übungsaufgabe 25. ■

12.11 Warteschlangen: Das Interface Queue

[102] Die Warteschlange ist typischerweise ein „first-in, first-out"-Speicher (FIFO-Speicher). Elemente werden an einem Ende eingesetzt und am anderen Ende entnommen. Die Reihenfolge in der Sie die Elemente einsetzen entspricht der Reihenfolge der Entnahme. Warteschlangen werden häufig verwendet, um Objekte zuverlässig von einem Programmbereich in einen anderen zu transportieren. Darüber hinaus sind Warteschlangen in der Threadprogrammierung besonders wichtig, wie Sie in Kapitel 22 lernen werden, da sie Objekte sicher zwischen zwei Aufgaben übertragen.

[103] Die Klasse `LinkedList` unterstützt Warteschlangenverhalten durch entsprechende Methoden und implementiert das Interface *`Queue`*, Sie können also einen Container vom Typ `LinkedList` als Warteschlange verwenden. Das folgende Beispiel wandelt die Referenz auf einen `LinkedList`-Container in eine *`Queue`*-Referenz um, um die Referenz auf die für Warteschlangen typischen, im Interface *`Queue`* deklarierten Methoden zu beschränken:

```
//: holding/QueueDemo.java
// Upcasting to a Queue from a LinkedList.
import java.util.*;

public class QueueDemo {
    public static void printQ(Queue queue) {
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        Queue<Integer> queue = new LinkedList<Integer>();
        Random rand = new Random(47);
        for(int i = 0; i < 10; i++)
            queue.offer(rand.nextInt(i + 10));
        printQ(queue);
        Queue<Character> qc = new LinkedList<Character>();
        for(char c : "Brontosaurus".toCharArray())
            qc.offer(c);
        printQ(qc);
    }
} /* Output:
```

```
    8 1 1 1 5 14 3 1 0 1
    B r o n t o s a u r u s
*///:~
```

[104] Die warteschlangenspezifische Methode `offer()` setzt an dem dem Kopfende gegenüberliegenden Ende ein Element ein oder gibt `false` zurück, falls kein Element eingesetzt werden kann. Die Methoden `peek()` und `element()` geben das Kopfelement der Warteschlange zurück, *ohne es zu entfernen*, wobei `peek()` bei leerer Warteschlange `null` zurückgibt, während `element()` eine Ausnahme vom Typ `NoSuchElementException` auswirft. Sowohl `poll()` als auch `remove()` entfernen das Kopfelement und geben es zurück, wobei `poll()` bei leerer Warteschlange `null` zurückgibt, während `remove()` eine Ausnahme vom Typ `NoSuchElementException` auswirft.

[105] Der `int`-Rückgabewert von `nextInt()` und der `char`-Wert `c` werden vor dem Einsetzen in die von `queue` beziehungsweise `qc` referenzierte Warteschlange durch Autoboxing automatisch in `Integer`- beziehungsweise `Character`-Objekte umgewandelt. Das Interface *`Queue`* schränkt die vorhandenen Methoden der Klasse `LinkedList` so ein, daß nur die zum Warteschlangenverhalten passenden Methoden verfügbar sind. Sie sind dadurch weniger versucht, `LinkedList`-Methoden aufzurufen, die keine Wartschlangenmethoden sind. (Sie könnten im obigen Fall die Referenzvariable `queue` in den Typ `LinkedList` zurückwandeln, but you are at least discouraged from doing so).

[106] Beachten Sie, daß die warteschlangenspezifischen Methoden komplette und eigenständige Funktionalität bieten, das heißt das Interface *`Queue`* beschreibt eine funktionstüchtige Warteschlange, ohne eine der Methoden des Interfaces *`Collection`* zu verwenden, von dem *`Queue`* abgeleitet ist.

Übungsaufgabe 27: (2) Schreiben Sie eine Klasse namens `Command` mit einem `String`-Feld und einer Methode `operation()`, die den Inhalt des `String`-Feldes ausgibt. Schreiben Sie eine zweite Klasse mit einer Methode, die eine Warteschlange vom Typ *`Queue`* mit `Command`-Objekten füllt und zurückgibt. Übergeben Sie die gefüllte Warteschlange einer Methode in einer dritten Klasse, die die gespeicherten Objekte verbraucht und ihre `operation()`-Methoden aufruft. ■

12.11.1 Prioritätswarteschlangen: Die Klasse PriorityQueue

[107] „First-in, first-out“ (FIFO) ist die typischste Warteschlangenregel. Eine solche Regel definiert das Kriterium, nach dem das nächste Kopfelement bestimmt wird. Die „FIFO-Regel“ besagt, daß das Element an die Reihe kommt, welches am längsten gewartet hat.

[108] Bei einer Prioritätswarteschlange (*priority queue*) kommt dasjenige Element als nächstes an die Reihe, welches den größten Bedarf (die höchste Priorität) hat. Beispielsweise kann am Flughafen ein Reisender aus der Warteschlange bevorzugt werden, wenn seine Maschine schon abflugbereit ist. In einem Benachrichtigungssystem sind manche Benachrichtungen wichtiger als andere und müssen bevorzugt zugestellt werden, unabhängig davon, zu welchem Zeitpunkt sie im System angelegt werden. Die Klasse `PriorityQueue` ist seit der SE 5 vorhanden und bietet eine automatische Implementierung dieses Verhaltens.

[109] Ein einer Prioritätswarteschlange vom Typ `PriorityQueue` per `offer()` übergebenes Element wird in die Reihenfolge der übrigen Elemente einsortiert.[5] Die Standardsortierung beruht auf der *natürlichen Ordnung* der Elemente, kann aber mit Hilfe eines Komparators geändert werden. Die Klasse `PriorityQueue` gewährleistet, daß die Methoden `peek()`, `poll()` und `remove()` stets das Element mit der höchsten Priorität liefern.

[5] Dieses Verhalten ist eigentlich implementierungsabhängig. Die Algorithmen der Prioritätswarteschlangen sortieren typischerweise beim Einsetzen (mit Hilfe eines Heaps), können das Element mit der höchsten Priorität aber auch erst bei der Entnahme bestimmen. Die Wahl des Algorithmus' ist entscheided, wenn sich die Priorität von Objekten ändern kann, während sie in der Warteschlange gespeichert sind.

[110] Eine Prioritätswarteschlange mit Elementen eines eingebauten Typs wie `Integer`, `String` oder `Character` ist ein triviales Beispiel. Die erste Wertemenge im folgenden Beispiel besteht aus denselben Zufallszahlen wie im Beispiel *QueueDemo.java* auf Seite 335, so daß Sie erkennen können, daß die Prioritätswarteschlange die Werte in einer anderen Reihenfolge herausgibt:

```
//: holding/PriorityQueueDemo.java
import java.util.*;

public class PriorityQueueDemo {
    public static void main(String[] args) {
        PriorityQueue<Integer> priorityQueue = new PriorityQueue<Integer>();
        Random rand = new Random(47);
        for(int i = 0; i < 10; i++)
            priorityQueue.offer(rand.nextInt(i + 10));
        QueueDemo.printQ(priorityQueue);

        List<Integer> ints =
            Arrays.asList(25, 22, 20, 18, 14, 9, 3, 1, 1, 2, 3, 9, 14, 18, 21,
                          23, 25);
        priorityQueue = new PriorityQueue<Integer>(ints);
        QueueDemo.printQ(priorityQueue);
        priorityQueue =
            new PriorityQueue<Integer>(ints.size(), Collections.reverseOrder());
        priorityQueue.addAll(ints);
        QueueDemo.printQ(priorityQueue);

        String fact = ''EDUCATION SHOULD ESCHEW OBFUSCATION'';
        List<String> strings = Arrays.asList(fact.split('' ''));
        PriorityQueue<String> stringPQ =
            new PriorityQueue<String>(strings);
        QueueDemo.printQ(stringPQ);
        stringPQ =
            new PriorityQueue<String>(strings.size(), Collections.reverseOrder());
        stringPQ.addAll(strings);
        QueueDemo.printQ(stringPQ);

        Set<Character> charSet = new HashSet<Character>();
        for(char c : fact.toCharArray())
            charSet.add(c); // Autoboxing
        PriorityQueue<Character> characterPQ =
            new PriorityQueue<Character>(charSet);
        QueueDemo.printQ(characterPQ);
    }
} /* Output:
      0 1 1 1 1 1 3 5 8 14
      1 1 2 3 3 9 9 14 14 18 18 20 21 22 23 25 25
      25 25 23 22 21 20 18 18 14 14 9 9 3 3 2 1 1
      A A B C C C D D E E E F H H I I L N N O O O O S S S T T U U U W
      W U U U T T S S S O O O O N N L I I H H F E E E D D C C C B A A
      A B C D E F H I L N O S T U W
   *///:~
```

[111] Duplikate sind erlaubt und der kleinste Wert hat die höchste Priorität (bei `String`-Objekten zählt Leerraum als Wert und hat höhere Priorität als Buchstaben). Beim dritten Aufruf des Konstruktors von `PriorityQueue<Integer>` und dem zweiten Aufruf des Konstruktors von `PriorityQueue<String>` wird der von der statischen `Collections`-Methode `reverseOrder()` (seit der SE 5 vorhanden) zurückgegebene Komparator übergeben, um vorzuführen, wie sich die Ordnung der Element in einer Prioritätswarteschlange per Komparator ändern läßt.

[113] Im letzten Abschnitt wird ein Container vom Typ `HashSet` angelegt, um die `Character`-Duplikate zu entfernen, nur um die Dinge etwas interessanter zu gestalten.

[114] Objekte der Klassen `Integer`, `String` und `Character` arbeiten mit einer Prioritätswarteschlange vom Typ `PriorityQueue` zusammen, weil diese Klassen bereits eine natürliche Ordnung eingebaut haben. Wenn Sie einer Prioritätswarteschlange Objekte einer selbstgeschriebenen Klasse übergeben möchten, müssen Sie zusätzliche Funktionalität entwickeln, um der Klasse eine natürliche Ordnung aufzuprägen oder einen entsprechenden Komparator hinzugeben. Sie finden ein etwas anspruchsvolleres Beispiel dafür in Kapitel 18.

Übungsaufgabe 28: (2) Füllen Sie eine Prioritätswarteschlange vom Typ `PriorityQueue` per `offer()` mit `Double`-Elementen, die Sie mit Hilfe der Klasse `Random` erzeugen. Entnehmen Sie die Element per `poll()` und geben Sie sie aus. ■

Übungsaufgabe 29: (2) Schreiben Sie eine einfache, von `Object` abgeleitete Klasse ohne Felder und Methoden und zeigen Sie, daß Sie einer Prioritätswarteschlange vom Typ `PriorityQueue` nicht mehr als ein Objekt dieser Klasse übergeben können. Die Angelegenheit wird in Kapitel 18 vollständig erklärt. ■

12.12 Konzeptvergleich: Interface Collection oder Iterator als Basis der Containerklassen

[115] Das Interface *`Collection`* ist das Wurzelinterface der Containerbibliothek (mit Ausnahme der Hierarchie unter dem Interface *`Map`*) und beschreibt die Gemeinsamkeiten aller Containerklassen. Man könnte *`Collection`* als „nebensächliches Interface“ bezeichnen, ein Interface, das nur zu existieren scheint, um die Gemeinsamkeiten zwischen den übrigen Containerinterfaces zusammenzufassen. Die abstrakte Klasse `AbstractCollection` liefert eine Standardimplementierung von *`Collection`*, gestattet Ihnen also, neue Klassen von `AbstractCollection` abzuleiten, ohne unnötig Quelltext zu duplizieren.

[116] Ein Argument für Interfaces ist, daß sie generischeren Quelltext ermöglichen. Wenn Sie eine Referenzvariable auf ein Interface statt einer Implementierung beziehen, können Sie Ihren Quelltext auf mehrere Objekttypen anwenden.[6] Eine Methode, die ein Argument vom Typ *`Collection`* erwartet, läßt sich auf jeden Typ anwenden, der *`Collection`* implementiert, and this allows a new class to choose to implement Collection in order to be used with my method. Es ist interessant, an dieser Stelle anzumerken, daß die Standardbibliothek von C++ keine allgemeine Basisklasse für ihre Containerklassen hat, sondern alle Gemeinsamkeiten zwischen den Containerklassen mit Hilfe von Iteratoren erfaßt. Es scheint auch für Java sinnvoll zu sein, dem Ansatz von C++ zu folgen und die Gemeinsamkeiten zwischen den Containerklassen über Iteratoren anstelle des *`Collection`*-Interfaces auszudrücken. Beide Ansätze sind aber miteinander verknüpft, da das Implementieren von *`Collection`* stets auch das Anlegen einer `iterator()`-Methode bedeutet:

```
//: holding/InterfaceVsIterator.java
import typeinfo.pets.*;
import java.util.*;

public class InterfaceVsIterator {
    public static void display(Iterator<Pet> it) {
```

[6] Einige Kollegen befürworten das automatische Anlegen von Interfaces für jede mögliche Kombination von Methoden in einer Klasse, zuweilen sogar für jede einzelne Klasse. Ich bin der Meinung, daß ein Interface mehr Bedeutung haben sollte, als das bloße mechanische Kopieren von Methodenkombinationen. Ich warte daher ab, bis sich der Nutzen eines neuen Interfaces abzeichnet, bevor ich es anlege.

```
        while(it.hasNext()) {
            Pet p = it.next();
            System.out.print(p.id() + ":" + p + " ");
        }
        System.out.println();
    }
    public static void display(Collection<Pet> pets) {
        for(Pet p : pets)
            System.out.print(p.id() + ":" + p + " ");
        System.out.println();
    }
    public static void main(String[] args) {
        List<Pet> petList = Pets.arrayList(8);
        Set<Pet> petSet = new HashSet<Pet>(petList);
        Map<String,Pet> petMap = new LinkedHashMap<String,Pet>();
        String[] names = ("Ralph, Eric, Robin, Lacey, " +
                          "Britney, Sam, Spot, Fluffy").split(", ");
        for(int i = 0; i < names.length; i++)
            petMap.put(names[i], petList.get(i));
        display(petList);
        display(petSet);
        display(petList.iterator());
        display(petSet.iterator());
        System.out.println(petMap);
        System.out.println(petMap.keySet());
        display(petMap.values());
        display(petMap.values().iterator());
    }
} /* Output:
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
      4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
      4:Pug 6:Pug 3:Mutt 1:Manx 5:Cymric 7:Manx 2:Cymric 0:Rat
      {Ralph=Rat, Eric=Manx, Robin=Cymric, Lacey=Mutt, Britney=Pug,
        Sam=Cymric, Spot=Pug, Fluffy=Manx}
      [Ralph, Eric, Robin, Lacey, Britney, Sam, Spot, Fluffy]
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
   *///:~
```

[117] Beide Versionen der `display()`-Methode akzeptieren sowohl iterable Objekte als auch Container vom Typ *`Collection`* und entkoppeln Methode von der Notwendigkeit, die Implementierung des unterliegenden Containers zu kennen.

[118] Im obigen Fall sind beide Ansätze praktisch gleichwertig. Der *`Collection`*-Ansatz hat einen kleinen Vorsprung, da *`Collection`* das Interface *`Iterable`* erweitert: Die Version der `display()`-Methode mit *`Collection`*-Parameter kann die erweiterte `for`-Schleife nutzen, wodurch der Quelltext ein wenig übersichtlicher wird.

[119] Der Iteratoransatz wird zwingend erforderlich, wenn Sie eine hierarchiefremde Klasse schreiben, das heißt eine Klasse die das Interface *`Collection`* nicht implementiert und schwierig oder lästig zu ändern ist, um die Implementierung doch zu bewerkstelligen. Wenn wir beispielsweise eine neue Klasse aus einer Klasse ableiten, die `Pet`-Objekte enthält und das Interface *`Collection`* implementieren wollen, müssen wir alle in *`Collection`* deklarierten Methoden anlegen, auch wenn wir keine davon im Körper der `display()`-Methode aufrufen. Auch wenn Sie eine Implementierung von *`Collection`* mühelos erhalten, indem Sie Ihre Klasse von `AbstractCollection` ableiten, müssen Sie

noch die Methoden `iterator()` und `size()` ausprogrammieren, die in `AbstractCollection` nicht implementiert sind, aber von anderen Methoden aus `AbstractCollection` aufgerufen werden:

```
//: holding/CollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

public class CollectionSequence
    extends AbstractCollection<Pet> {
    private Pet[] pets = Pets.createArray(8);
    public int size() { return pets.length; }
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // Not implemented
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        CollectionSequence c = new CollectionSequence();
        InterfaceVsIterator.display(c);
        InterfaceVsIterator.display(c.iterator());
    }
} /* Output:
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
   *///:~
```

[120] Die `remove()`-Methode ist eine „optionale Operation“ (siehe Fußnote 4, auf Seite 323). Sie lernen in Kapitel 18 mehr darüber. In diesem Beispiel ist es nicht notwendig, die Methode zu implementieren und wenn sie aufgerufen wird, wirft sie eine Ausnahme aus.

[121] Das obige Beispiel *CollectionSequence.java* zeigt, daß das Implementieren des Interfaces *`Collection`* stets auch das Implementierung der `iterator()`-Methode beinhaltet und daß das alleinige Implementieren der `iterator()`-Methode kaum weniger Aufwand bedeutet, als die Ableitung von der abstrakten Klasse `AbstractCollection`. Falls Sie Ihre Klasse bereits von einer anderen Klasse ableiten, scheitet die Ableitung von `AbstractCollection` allerdings aus. In diesem Fall müßten Sie, um *`Collection`* zu implementieren, alle in diesem Interface deklarierten Methoden anlegen. Es ist in diesem Fall einfacher, die Ableitungsbeziehung zu belassen und die Möglichkeit einen Iterator zu erzeugen vorzusehen:

```
//: holding/NonCollectionSequence.java
import typeinfo.pets.*;
import java.util.*;

class PetSequence {
    protected Pet[] pets = Pets.createArray(8);
}

public class NonCollectionSequence extends PetSequence {
    public Iterator<Pet> iterator() {
        return new Iterator<Pet>() {
            private int index = 0;
            public boolean hasNext() {
```

```
                return index < pets.length;
            }
            public Pet next() { return pets[index++]; }
            public void remove() { // Not implemented
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        NonCollectionSequence nc = new NonCollectionSequence();
        InterfaceVsIterator.display(nc.iterator());
    }
} /* Output:
      0:Rat 1:Manx 2:Cymric 3:Mutt 4:Pug 5:Cymric 6:Pug 7:Manx
   *///:~
```

[122] Einen Iterator zu liefern, ist die am wenigsten gekoppelte Möglichkeit, um eine Reihe von Elementen mit einer Methoden zu verbinden, die diese Reihe verarbeitet und setzt die Reihe weniger Einschränkungen aus, als eine Implementierung des Interfaces *`Collection`*.

Übungsaufgabe 30: (5) Ändern Sie das Beispiel *CollectionSequence.java* so, daß die Klasse `CollectionSequence` nicht von `AbstractCollection` abgeleitet wird, sondern das Interface *`Collection`* implementiert. ■

12.13 Der Zusammenhang zwischen Iteratoren und der erweiterten for-Schleife

[123] Wir haben die erweiterte `for`-Schleife bis jetzt hauptsächlich auf Arrays angewendet, obwohl sie auch bei Containern vom Typ *`Collection`* funktioniert. Sie haben bereits einige Beispiele mit Containern vom Typ `ArrayList` gesehen, aber hier ist ein allgemeingültiger Beweis:

```
//: holding/ForEachCollections.java
// All collections work with foreach.
import java.util.*;

public class ForEachCollections {
    public static void main(String[] args) {
        Collection<String> cs = new LinkedList<String>();
        Collections.addAll(cs, ”Take the long way home”.split(” ”));
        for(String s : cs)
            System.out.print(”’” + s + ”’ ”);
    }
} /* Output:
      ’Take’ ’the’ ’long’ ’way’ ’home’
   *///:~
```

Da `sc` einen Container vom Typ *`Collection`* referenziert, zeigt dieses Beispiel, daß die Eigenschaft, in die erweitere `for`-Schleife eingesetzt werden zu können, für alle Containerklassen kennzeichnend ist.

[124] Diese Eigenschaft existiert, seit der SE 5 das neue Interface *`Iterable`* eingeführt hat, welches eine Methode namens `iterator()` deklariert, die eine Referenz auf ein Objekt vom Typ *`Iterator`* zurückgibt. Die erweiterte `for`-Schleife stützt sich auf das Interface *`Iterable`*, um sich durch eine Reihe von Elementen zu bewegen. Sie können jede Klasse, die *`Iterable`* implementiert („iterabel“ ist), in die erweiterte `for`-Schleife einsetzen:

```
//: holding/IterableClass.java
// Anything Iterable works with foreach.
import java.util.*;

public class IterableClass implements Iterable<String> {
    protected String[] words =
        (''And that is how we know the Earth to be banana-shaped.'').split(''  '');
    public Iterator<String> iterator() {
        return new Iterator<String>() {
            private int index = 0;
            public boolean hasNext() {
                return index < words.length;
            }
            public String next() { return words[index++]; }
            public void remove() { // Not implemented
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        for(String s : new IterableClass())
            System.out.print(s + '' '');
    }
} /* Output:
      And that is how we know the Earth to be banana-shaped.
   *///:~
```

[125] Die `iterator()`-Methode gibt eine Referenz auf ein Objekt der anonymen inneren Klasse zurück, die das Interface *`Iterable<String>`* implementiert und ein Wort nach dem anderen aus dem Array zurückgibt. In der `main()`-Methode sehen Sie, daß sich ein Objekt der Klasse `IterableClass` tatsächlich in die erweiterte `for`-Schleife einsetzen läßt.

[126] In der SE 5 wurden viele Klassen um das Interface *`Iterable`* ergänzt, hauptsächlich die Containerklassen unter dem Interface *`Collection`* (nicht aber die Containerklassen unter *`Map`*). Das folgende Beispiel zeigt alle Umgebungsvariablen Ihres Betriebssystems an:

```
//: holding/EnvironmentVariables.java
import java.util.*;

public class EnvironmentVariables {
    public static void main(String[] args) {
        for(Map.Entry entry: System.getenv().entrySet()) {
            System.out.println(entry.getKey() + '': '' + entry.getValue());
        }
    }
} /* (Execute to see output) *///:~
```

[127] Die statische `System`-Methode `getenv()`[7] gibt einen Container vom Typ *`Map`* zurück, `entrySet()` einen *`Set`*-Container von *`Map.Entry`*-Objekten, der in die erweiterte `for`-Schleife eingesetzt werden kann, da die Klasse *`Set`* das Interface *`Iterable`* implementiert.

[128] Jedes Array und jede Klasse, die das Interface *`Iterable`* implementiert, läßt sich in die erweiterte `for`-Schleife einsetzen. Daraus folgt allerdings *nicht*, das Arrays *`Iterable`* implementieren oder eine automatische Umwandlung in einen iterablen Typ erfolgt:

[7] Diese Möglichkeit war vor der [SE 5 nicht vorhanden, da man die Ansicht vertrat, eine zu enge Bindung an das Betriebssystem zu implementieren und somit den Anspruch von Java „write once, run anywhere“ zu verletzen. Die Tatsache, daß diese Möglichkeit nun noch besteht, erweckt den Anschein, daß die Designer von Java pragmatischer werden.

```
//: holding/ArrayIsNotIterable.java
import java.util.*;

public class ArrayIsNotIterable {
    static <T> void test(Iterable<T> ib) {
        for(T t : ib)
            System.out.print(t + " ");
    }
    public static void main(String[] args) {
        test(Arrays.asList(1, 2, 3));
        String[] strings = { "A", "B", "C" };
        // An array works in foreach, but it's not Iterable:
        //! test(strings);
        // You must explicitly convert it to an Iterable:
        test(Arrays.asList(strings));
    }
} /* Output:
      1 2 3 A B C
   *///:~
```

[129] Der Versuch ein Array als Argument vom Typ *Iterable* zu übergeben scheitert. Es gibt keine automatische Umwandlung in einen iterierbaren Typ. Sie müssen die Umwandlung von Hand erledigen.

Übungsaufgabe 31: (3) Ändern Sie das Beispiel *polymorphism/shape/RandomShapeGenerator-.java* (Seite **??**f), so daß die Klasse `RandomShapeGenerator` das Interface *Iterable* implementiert. Sie müssen einen zusätzlichen Konstruktor anlegen, der die Anzahl der vom Iterator zu liefernden Elemente erwartet, bevor der Iterator anhält. Verifizieren Sie, daß Ihr Programm funktioniert. ■

12.13.1 Die Adaptermethode

[130] Angenommen, Sie haben eine Klasse die das Interface *Iterable* implementiert und Sie möchten diese Klasse um eine oder mehrere Anwendungsmöglichkeiten in der erweiterten `for`-Schleife ergänzen. Stellen Sie sich beispielsweise vor, daß Sie wählen können möchten, ob eine Liste von Wörtern vorwärts oder rückwärts durchlaufen wird. Wenn Sie einfach eine neue Klasse ableiten und die `iterator()`-Methode überschreiben, ersetzen Sie die existierende Methode und haben keine Wahlmöglichkeit mehr.

[131] Dieser Unterabschnitt stellt einen Lösungsansatz vor, den ich als „Adaptermethode“ bezeichne. Der „Adapter“-Teil stammt vom gleichnamigen Entwurfsmuster (*Adapter*), da Sie eine bestimmte Schnittstelle liefern müssen, um die erweiterte `for`-Schleife zu bedienen. Liegt eine Schnittstelle vor, wobei aber eine andere benötigt wird, so können Sie das Problem mit Hilfe eines Adapters lösen. Wir wollen die Fähigkeit, zusätzlich zum vorwärtsgerichteten Standarditerator einen rückwärtsgerichteten Iterator erzeugen zu können *ergänzen*, können die `iterator()`-Methode also nicht überschreiben. Stattdessen legen wir eine Methode an, die ein iterierbares Objekt zurückgibt, welches wir anschließend in die erweiterte `for`-Schleife einsetzen können. Dieser Ansatz gestattet, die erweiterte `for`-Schleife auf mehrere Arten zu nutzen:

```
//: holding/AdapterMethodIdiom.java
// The "Adapter Method" idiom allows you to use foreach
// with additional kinds of Iterables.
import java.util.*;

class ReversibleArrayList<T> extends ArrayList<T> {
    public ReversibleArrayList(Collection<T> c) { super(c); }
    public Iterable<T> reversed() {
```

```
        return new Iterable<T>() {
            public Iterator<T> iterator() {
                return new Iterator<T>() {
                    int current = size() - 1;
                    public boolean hasNext() { return current > -1; }
                    public T next() { return get(current--); }
                    public void remove() { // Not implemented
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
}

public class AdapterMethodIdiom {
    public static void main(String[] args) {
        ReversibleArrayList<String> ral =
            new ReversibleArrayList<String>(
                Arrays.asList("To be or not to be".split(" ")));
        // Grabs the ordinary iterator via iterator():
        for(String s : ral)
            System.out.print(s + " ");
        System.out.println();
        // Hand it the Iterable of your choice
        for(String s : ral.reversed())
            System.out.print(s + " ");
    }
} /* Output:
      To be or not to be
      be to not or be To
   *///:~
```

[132] Wenn Sie den von `ral` referenzierten Container in die erweiterte `for`-Schleife einsetzen, erhalten Sie das Standardverhalten, also den vorwärtsgerichteten Iterator. Wenn Sie die `reversed()`-Methode des Containers aufrufen, erhalten Sie ein anderes Verhalten.

[133] Mit diesem Ansatz läßt sich das Beispiel *IterableClass.java* um zwei Adaptermethoden erweitern:

```
//: holding/MultiIterableClass.java
// Adding several Adapter Methods.
import java.util.*;

public class MultiIterableClass extends IterableClass {
    public Iterable<String> reversed() {
        return new Iterable<String>() {
            public Iterator<String> iterator() {
                return new Iterator<String>() {
                    int current = words.length - 1;
                    public boolean hasNext() { return current > -1; }
                    public String next() { return words[current--]; }
                    public void remove() { // Not implemented
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
```

```
    public Iterable<String> randomized() {
        return new Iterable<String>() {
            public Iterator<String> iterator() {
                List<String> shuffled =
                    new ArrayList<String>(Arrays.asList(words));
                Collections.shuffle(shuffled, new Random(47));
                return shuffled.iterator();
            }
        };
    }
    public static void main(String[] args) {
        MultiIterableClass mic = new MultiIterableClass();
        for(String s : mic.reversed())
            System.out.print(s + " ");
        System.out.println();
        for(String s : mic.randomized())
            System.out.print(s + " ");
        System.out.println();
        for(String s : mic)
            System.out.print(s + " ");
    }
} /* Output:
      banana-shaped. be to Earth the know we how is that And
      is banana-shaped. Earth that how the be And we know to
      And that is how we know the Earth to be banana-shaped.
   *///:~
```

[134] Beachten Sie, daß die zweite Methode (`randomized()`) keinen eigenen Iterator definiert, sondern einfach den Iterator des durcheinander gemischten *`List`*-Containers zurückgibt.

[135] Die Ausgabe zeigt, daß die statische `Collections`-Methode `shuffle()` nur die Referenzen in dem von `shuffled` referenzierten `ArrayList`-Container mischt, nicht aber die Elemente des ursprünglichen Arrays. Dies trifft nur zu, weil die `randomized()`-Methode das Ergebnis von `Arrays.asList()` in einen `ArrayList`-Container setzt. Würde die von `Arrays.asList()` zurückgegebene Liste direkt sortiert, so würde das unterliegende Array ebenfalls verändert:

```
//: holding/ModifyingArraysAsList.java
import java.util.*;

public class ModifyingArraysAsList {
    public static void main(String[] args) {
        Random rand = new Random(47);
        Integer[] ia = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
        List<Integer> list1 = new ArrayList<Integer>(Arrays.asList(ia));
        System.out.println("Before shuffling: " + list1);
        Collections.shuffle(list1, rand);
        System.out.println("After shuffling: " + list1);
        System.out.println("array: " + Arrays.toString(ia));

        List<Integer> list2 = Arrays.asList(ia);
        System.out.println("Before shuffling: " + list2);
        Collections.shuffle(list2, rand);
        System.out.println("After shuffling: " + list2);
        System.out.println("array: " + Arrays.toString(ia));
    }
} /* Output:
      Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
      After shuffling: [4, 6, 3, 1, 8, 7, 2, 5, 10, 9]
```

```
    array: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    Before shuffling: [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
    After shuffling: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
    array: [9, 1, 6, 3, 7, 2, 5, 10, 4, 8]
*///:~
```

[136] Im ersten Fall wird die Ausgabe von `Arrays.asList()` dem Konstruktor der Klasse `ArrayList` übergeben, welcher einen Container dieses Typs erzeugt, der die Elemente in dem von `ia` referenzierten Array referenziert. Das Mischen dieser Referenzen hat keine Auswirkungen auf das Array. Wenn Sie das Ergebnis von `Arrays.asList()` direkt verwenden, verändert das Mischen die Reihenfolge der Elemente in `ia`. Es ist wichtig, zu verstehen, daß `Arrays.asList()` einen Container vom Typ *`List`* erzeugt, welcher das unterliegende Array als physikalische Implementierung verwendet. Wenn Sie diesen Container verändern und nicht wollen, daß sich das ursprüngliche Array ebenfalls ändert, müssen Sie eine Kopie in einem anderen Container anlegen.

Übungsaufgabe 32: (2) Legen Sie im Beispiel *NonCollectionSequence.java* zwei Methoden `reversed()` und `randomized()` an (orientieren Sie sich am Beispiel *MultilterableClass.java* auf Seite 344f) und ergänzen Sie auch die Klasse `NonCollectionSequence` so daß sie das Interface *`Iterable`* implementiert. Zeigen Sie, daß alle Ansätze mit der erweiterten `for`-Schleife zusammenarbeiten. ■

12.14 Zusammenfassung

[137] Java bietet verschiedene Möglichkeiten zum Speichern von Objekten:

- Ein Array verknüpft Objekte mit einem ganzzahligen Index. Es enthält Objekte bekannten Typs, so daß beim Nachschlagen keine Typumwandlung erforderlich ist. Arrays können mehrdimensional sein und Elemente primitiven Typs enthalten. Die Länge eines Arrays kann nach dem Erzeugen des Arrayobjektes nicht mehr geändert werden.

- Ein Container vom Typ *`Collection`* enthält einzelne Elemente, ein Container vom Typ *`Map`* enthält Schlüssel/Wert-Paare. Bei einem generischen Container geben Sie den Typ der gespeicherten Objekte an, so daß Sie keine typfremden Elemente einsetzen können und den Typ eines aus dem Container entnommen Elementes nicht umwandeln müssen. Beide Containertypen passen ihre Kapazität automatisch an, wenn Sie weitere Elemente eintragen. Ein Container kann zwar keine Elemente primitiven Typs enthalten, aber der Autoboxingmechanismus kümmert sich um die Umwandlung zwischen primitivem Typ und dem Wrappertyp der gespeicherten Elemente.

- Eine Liste (ein Container vom Typ *`List`*) verknüpft Objekte mit ganzzahligen Indices, wie ein Array. Somit sind Arrays und Listen geordnete Container.

- Wenn Sie häufig Zugriff auf beliebige Elemente brauchen, wählen Sie einen Container vom Typ `ArrayList`. Wenn Sie häufig Elemente in die Mitte der Liste einsetzen oder von dort entfernen müssen, wählen Sie einen Container vom Typ `LinkedList`.

- Die Klasse `LinkedList` implementiert auch Stack- und Warteschlangenverhalten.

- Ein Container vom Typ *`Map`* gestattet, Objekte nicht mit ganzzahligen Werten, sondern mit *Objekten* zu verknüpfen. Der Containertyp `HashMap` ist für hohe Zugriffsgeschwindigkeit gedacht, während `TreeMap` die Schlüssel in sortierter Reihenfolge speichert und daher weniger schnell ist als `HashMap`. Der Containertyp `LinkedHashMap` erhält die Einsetzungsreihenfolge der Elemente und gestattet durch einen Hashalgorithmus schnellen Zugriff.

- Ein Container vom Typ *Set* akzeptiert höchstens ein Exemplar eines Elements. Der Containertyp `HashSet` gestattet schnellstmögliches Nachschlagen, während `TreeSet` die Elemente in sortierter Reihenfolge speichert. Der Containertyp `LinkedHashSet` erhält die Einsetzungsreihenfolge der Elemente.
- Es besteht keine Notwendigkeit, die veralteten Klassen `Vector`, `Hashtable` und `Stack` in einem neu entwickelten Quelltext noch zu verwenden.

[138] Abbildung [Seite 439 im Buch] zeigt eine vereinfachte Übersicht über die Containerbibliothek von Java (ohne abstrakte Klassen und veraltete Komponenten). Die Abbildung umfaßt nur die Interfaces und Klassen, die mit denen Sie in der Regel arbeiten.

[139] Wie Sie sehen, gibt es eigentlich nur vier grundlegende Containertypen, nämlich *Map*, *List*, *Set* und *Queue*, sowie jeweils nur zwei oder drei Implementierungen (die *Queue*-Implementierungen im Package `java.util.concurrent` sind in der Abbildung nicht verzeichnet). Die am häufigsten vorkommenden Containertypen haben einen dicken schwarzen Rahmen.

[140] Die gestrichelten Rahmen stellen Interfaces dar, die durchgezogenen Rahmen gewöhnliche (konkrete) Klassen. Gestrichelte Linien mit hohlen Pfeilspitzen bedeuten, daß eine bestimmte Klasse ein Interface implementiert. Durchgezogene Linien mit ausgefüllter Pfeilspitze bedeuten, daß eine Klasse Objekte der Klasse erzeugen kann, auf die die Pfeilspitze zeigt. Beispielsweise kann jeder Container vom Typ *Collection* einen Iterator (*Iterator*) erzeugen und jeder Container vom Typ *List* einen Listeniterator (*ListIterator*) sowie einen gewöhnlichen Iterator, da *List* das Interface *Collection* erweitert.

[141] Das folgende Beispiel zeigt die Unterschiede in den öffentlichen Schnittstellen der verschiedenen Containerklassen. Das Programm, welches die folgende Ausgabe erzeugt stammt aus Kapitel 16 (Seite 504) und wird hier nur aufgerufen, um die Ausgabe zu verwerten. Die Ausgabe zeigt auch die von der/dem jeweiligen Klasse beziehungsweise Interface implementierten beziehungsweise erweiterten Interfaces an:

```
//: holding/ContainerMethods.java
import net.mindview.util.*;

public class ContainerMethods {
    public static void main(String[] args) {
        ContainerMethodDifferences.main(args);
    }
} /* Output: (Sample)
      Collection: [add, addAll, clear, contains, containsAll, equals, hashCode,
        isEmpty, iterator, remove, removeAll, retainAll, size, toArray]
      Interfaces in Collection: [Iterable]
      Set extends Collection, adds: []
      Interfaces in Set: [Collection]
      HashSet extends Set, adds: []
      Interfaces in HashSet: [Set, Cloneable, Serializable]
      LinkedHashSet extends HashSet, adds: []
      Interfaces in LinkedHashSet: [Set, Cloneable, Serializable]
      TreeSet extends Set, adds: [pollLast, navigableHeadSet, descendingIterator,
        lower, headSet, ceiling, pollFirst, subSet, navigableTailSet, comparator,
        first, floor, last, navigableSubSet, higher, tailSet]
      Interfaces in TreeSet: [NavigableSet, Cloneable, Serializable]
      List extends Collection, adds: [listIterator, indexOf, get, subList, set,
        lastIndexOf]
      Interfaces in List: [Collection]
      ArrayList extends List, adds: [ensureCapacity, trimToSize]
      Interfaces in ArrayList: [List, RandomAccess, Cloneable, Serializable]
```

```
    LinkedList extends List, adds: [pollLast, offer, descendingIterator, addFirst,
      peekLast, removeFirst, peekFirst, removeLast, getLast, pollFirst, pop, poll,
      addLast, removeFirstOccurrence, getFirst, element, peek, offerLast, push,
      offerFirst, removeLastOccurrence]
    Interfaces in LinkedList: [List, Deque, Cloneable, Serializable]
    Queue extends Collection, adds: [offer, element, peek, poll]
    Interfaces in Queue: [Collection]
    PriorityQueue extends Queue, adds: [comparator]
    Interfaces in PriorityQueue: [Serializable]
    Map: [clear, containsKey, containsValue, entrySet, equals, get, hashCode,
      isEmpty, keySet, put, putAll, remove, size, values]
    HashMap extends Map, adds: []
    Interfaces in HashMap: [Map, Cloneable, Serializable]
    LinkedHashMap extends HashMap, adds: []
    Interfaces in LinkedHashMap: [Map]
    SortedMap extends Map, adds: [subMap, comparator, firstKey, lastKey, headMap,
      tailMap]
    Interfaces in SortedMap: [Map]
    TreeMap extends Map, adds: [descendingEntrySet, subMap, pollLastEntry,
      lastKey, floorEntry, lastEntry, lowerKey, navigableHeadMap,
      navigableTailMap, descendingKeySet, tailMap, ceilingEntry, higherKey,
      pollFirstEntry, comparator, firstKey, floorKey, higherEntry, firstEntry,
      navigableSubMap, headMap, lowerEntry, ceilingKey]
    Interfaces in TreeMap: [NavigableMap, Cloneable, Serializable]
*///:~
```

[142] Mit Ausnahme von `TreeSet` haben alle *`Set`*-Implementierungen exakt dieselbe Schnittstelle wie das Interface *`Collection`*. Die Interfaces *`List`* und *`Collection`* unterscheiden sich deutlich voneinander, obwohl *`List`* Methoden verlangt, die bereits in *`Collection`* deklariert sind. Andererseits sind die in *`Queue`* deklarierten Methoden eigenständig, das heißt keine der Methoden aus *`Collection`* ist notwendig, um eine funktionstüchtige Implementierung des *`Queue`*-Interfaces zu schreiben. Der einzige Zusammenhang zwischen den Containertypen *`Map`* und *`Collection`* besteht darin, daß ein *`Map`*-Container mit Hilfe einer Methoden `entrySet()` und `values()` einen Container vom Typ *`Collection`* erzeugen kann.

[143] Beachten Sie das von `ArrayList`, nicht aber von `LinkedList` implementierte Markierungsinterface *`java.util.RandomAccess`*. Es dient als Information für Algorithmen, die ihr Verhalten abhängig vom Listentyp dynamisch ändern.

[144] Diese Organisation ist hinsichtlich der objektorientierten Hierarchie ein wenig seltsam. Wenn Sie sich tiefer mit den Containerklassen im Package `java.util` auseinandersetzen (besonders in Kapitel 18), werden Sie sehen, daß neben einer etwas sonderbaren Ableitungsstruktur noch weitere merkwürdige Dinge gibt. Containerbibliotheken waren schon immer ein schwieriges Design-Problem. Das Lösen dieses Problems bedeutet, viele Anforderungen zu berücksichtigen, die häufig gegeneinander gerichtet sind. Sie sollten sich also auf den einen oder anderen Kompromiß gefaßt machen.

[145] Dennoch sind die Containerklassen von Java fundamentale Hilfsmittel, die Ihnen die alltägliche Arbeit erleichtern können und Ihre Programme mächtiger und effektiver gestalten. Es mag eine Weile dauern, bis Sie sich an den einen oder anderen Gesichtspunkt der Containerbibliothek gewöhnt haben, aber ich glaube, daß Sie die Klassen dieser Bibliothek schnell annehmen und einsetzen werden.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 13

Fehlerbehandlung mit Ausnahmen

Inhaltsübersicht

[0] Ein grundlegender Leitsatz von Java lautet „badly formed code will not be run“, übersetzt etwa: „schlechter Quelltext wird nicht ausgeführt“.

[1] Der ideale Zeitpunkt zum Abfangen eines Fehlers ist während der Übersetzungszeit, noch bevor Sie überhaupt versuchen können, das Programm aufzurufen. Allerdings können nicht alle Fehler zur Übersetzungszeit entdeckt werden. Diese verbleibenden Probleme müssen zur Laufzeit über einen Formalismus verarbeitet werden, der dem Verursacher des Fehlers ermöglicht, eine entsprechende Benachrichtigung an einen Empfänger zu übergeben, der weiß, wie das Problem angemessen behandelt wird.

[2] „Error Recovery“, übersetzt etwa: „Wiederherstellung eines funktionsfähigen Programmzustandes“, gehört zu den wirksamsten Verfahren, um die Robustheit Ihres Programms zu verbessern. Diese Regenerationsfähigkeit des Programmzustandes nach dem Auftreten von Fehlern sollte ein fundamentales Anliegen bei jedem Ihrer Programme sein und ist bei Java besonders wichtig, da eines der primären Ziele darin besteht, Programmkomponenten zu entwickeln, die andere Programmierer anwenden können. Die Entwicklung eines robusten Systems setzt die Robustheit aller Komponenten des Systems voraus. Das auf Ausnahmen beruhende konsistente Fehlerbenachrichtungsmodell von Java gestattet Komponenten, den aufrufenden Programmteil zuverlässig über Schwierigkeiten zu informieren.

[3] Die Ziele der Ausnahmebehandlung bei Java sind das vereinfachte Anlegen großer zuverlässiger Programme mit weniger Quelltext als gegenwärtig möglich sowie dies mit mehr Vertrauen dahingehend zu tun, daß Ihre Anwendung keine unbehandelten Fehler beinhaltet. Der Umgang mit Ausnahmen ist nicht schwer zu erlernen und Ausnahmen gehören zu den Spracheigenschaften, die sich unmittelbar und vorteilhaft auf Ihre Projekte auswirken.

[4] Da die Ausnahmebehandlung bei Java das einzige offizielle Verfahren zur Anzeige von Fehlern ist und seine Anwendung vom Compiler erzwungen wird, gibt es in diesem Buch nur wenige Beispiele, in denen keine Ausnahmebehandlung vorkommt. In diesem Kapitel werden die Anweisungen vorgestellt, die Sie anlegen müssen, um Ausnahmen korrekt zu behandeln. Außerdem zeigt dieses Kapitel, wie Sie eigene Ausnahmeklassen schreiben können, falls eine Ihrer Methoden in Schwierigkeiten gerät.

13.1 Fehlerbehandlungskonzepte

[5] C und andere ältere Sprachen kannten häufig mehrere Verfahren zur Fehlerbehandlung, die außerdem in der Regel auf dem Einhalten von Konventionen beruhten, aber kein Teil der Programmiersprache waren. Typischerweise wurde ein spezieller Wert zurückgegeben oder ein Flag gesetzt und der Programmierer hatte die Aufgabe, den Rückgabewert oder das Flag auszuwerten, um den Fehlerzustand festzustellen. Im Laufe der Jahre zeigte sich aber, daß die Programmierer, die Bibliotheken anwendeten, dazu neigten, sich für unfehlbar zu halten („Fehler passieren nur anderen, aber nicht mir.“) und daher auf die Auswertung des Fehlerzustandes verzichteten (teilweise waren die Fehlerzustände aber auch zu einfältig, um geprüft zu werden[1]). Waren Sie dagegen so sorgfältig, jeden Funktionsaufruf auf eventuelle Fehler hin zu überprüfen, so verwandelte sich der Quelltext unter Umständen in einen unleserlichen Alptraum. Da die Programmierer die Fehlerbehandlung in diesen Sprache nicht ernst zu nehmen brauchten, waren sie resistent dagegen, sich einzugestehen, daß diese Art der „Fehlerbehandlung“ eine der Hauptbeschränkungen bei der Entwicklung großer, robuster und pflegbarer Programme darstellte.

[1] Der C-Programmierer betrachte die Rückgabewerte der `printf()`-Funktion als Beispiel hierfür.

[6] Die Lösung besteht darin, der Fehlerbehandlung die Zwanglosigkeit zu nehmen und statt dessen Formalität zu erzwingen. Die Fehlerbehandlung hat eine lange Geschichte, da die Implementierungen bis zu den Betriebssystemen in den 1960er Jahren zurückreichen, sogar bis zur `on error goto`-Anweisung von BASIC. Die Fehlerbehandlung von C++ beruht auf Ada, die Fehlerhandlung von Java wiederum primär auf C++ (obwohl sie optisch eher an Object Pascal erinnert).

[7] Der Begriff „Exception“ (Ausnahme) ist im Sinne von „to take exception to sth.“ gemeint, übersetzt etwa: „Anstoß an etwas nehmen“. An der Stelle an der ein Fehler auftritt, wissen Sie eventuell nicht, wie Sie damit verfahren sollen. Sie wissen andererseits, daß Sie das Programm nicht einfach weiter laufen lassen können. Sie müssen das Programm unterbrechen und irgendjemand muß an irgendeiner Stelle definieren, was geschehen soll. Sie übergeben das Problem einem übergeordneten Kontext, in dem jemand in der Lage ist, eine angemessene Entscheidung zu fällen.

[8] Ein anderer wesentlicher Vorteil von Ausnahmen besteht darin, daß sie tendentiell die Komplexität der Fehlerbehandlungsanweisungen verringern. Ohne Ausnahmen müssen Sie an mehreren Stellen in Ihrem Programm prüfen, ob ein bestimmter Fehler aufgetreten ist und den Fehler behandeln. Mit Ausnahmen brauchen Sie den Fehlerzustand nicht länger an der Stelle des Methodenaufrufs auszuwerten, da der Ausnahmenmechanismus garantiert, daß eine hervorgerufene („ausgeworfene“) Ausnahme abgefangen wird. Das Problem wird an nur einer Stelle bearbeitet, nämlich dem sogenannten *Ausnahmebehandler*. Dadurch wird der Quelltext kürzer und die Anweisungen für den normalen Programmablauf sind von den Anweisungen getrennt, die im Fehlerfall aufgerufen werden. Im allgemeinen wird das Lesen, Schreiben sowie die Fehlersuche im Quelltext mit Ausnahmen viel klarer, als bei den älteren Fehlerbehandlungsverfahren.

13.2 Grundlagen der Ausnahmebehandlung

[9] Eine °**Ausnahmesituation** ist ein Problem, das die Fortsetzung der Verarbeitung der aktuellen Methode oder der Anweisungen im aktuellen Geltungsbereich verhindert. Es ist wichtig, Ausnahmesituationen von „gewöhnlichen Problemen“ zu unterscheiden, bei denen der aktuelle Kontext genügend Informationen enthält, um mit der Lage zurecht zu kommen. In einer Ausnahmesituation kann die Verarbeitung dagegen nicht fortgesetzt werden, weil *der aktuelle Kontext* die zur Problemlösung notwendigen Informationen *nicht enthält*. Sie können nicht mehr tun, als den aktuellen Kontext zu verlassen und das Problem an einen übergeordneten Kontext weiterzugeben. Genau dies geschieht beim °**Auswerfen einer Ausnahme**.

[10] Die Division zweier Zahlen liefert ein einfaches Beispiel: Besteht die Möglichkeit der Division durch Null, so ist es sinnvoll, den Teiler vorher zu überprüfen. Doch was bedeutet es, wenn der Teiler Null wird? Vielleicht können Sie aus dem Kontext des Problems, das Sie in dieser bestimmten Methode zu lösen versuchen, ableiten wie Sie einen verschwindenden Teiler interpretieren müssen. Ist die Null dagegen ein unerwarteter Wert, so können Sie die Situation nicht deuten und müssen eine Ausnahme auswerfen, statt den Ausführungspfad fortzusetzen.

[11] Beim Auswerfen einer Ausnahme geschehen mehrere Dinge: Zuerst wird per `new`-Operator ein Ausnahmeobjekt im dynamischen Speicher erzeugt, analog zur Erzeugung eines beliebigen anderen Java-Objektes. Anschließend wird der aktuelle Ausführungspfad (der nicht fortgesetzt werden kann) unterbrochen und die Referenz auf das zuvor erzeugte Ausnahmeobjekt aus dem aktuellen Kontext ausgeworfen. Nun übernimmt der Ausnahmebehandlungsmechanismus die Kontrolle und sucht nach einer passenden Stelle, um das Programm fortzusetzen. Diese passende Stelle ist der *Ausnahmebehandler*, dessen Aufgabe darin besteht, nach dem Auftreten eines bestimmten Problems wieder einen funktionsfähigen Zustand herzustellen, so daß die Verarbeitung des Programmes entweder einen anderen Weg nimmt oder einfach fortgesetzt werden kann.

[12] Wir betrachten eine Referenzvariable `t` als einfaches Beispiel für das Auswerfen einer Ausnahme. Es ist nicht auszuschließen, daß `t` nicht initialisiert ist. Bevor Sie über `t` eine Methode aufrufen, ist es also sinnvoll, die Referenzvariable zu prüfen. Sie übergeben die Information über das Auftreten des Fehlers „`t` nicht initialisiert“ an einen übergeordneten Kontext, indem Sie ein Objekt erzeugen, welches diese Information repräsentiert und die Referenz auf dieses Objekt anschließend aus dem aktuellen Kontext „auswerfen“:

```
if (t == null)
    °throw new NullPointerException();
```

[13] Das Auswerfen dieser Ausnahme gestattet Ihnen, im aktuellen Kontext von der Verantwortung zurückzutreten, sich über diese Angelegenheit Gedanken zu machen. Das Problem wird einfach an einer anderen Stelle behandelt. Abschnitt 13.3 zeigt *wo genau* die Ausnahmebehandlung stattfindet.

[14] Ausnahmen gestatten Ihnen, alles was Sie tun als °Transaktion zu betrachten, die von Ausnahmen überwacht wird: „... the fundamental premise of transactions is that we needed exception handling in distributed computations. Transactions are the computer equivalent of contract law. If anything goes wrong, we'll just blow away the whole computation.“[2] Übersetzt etwa: „Die Notwendigkeit der Ausnahmebehandlung bei verteilten Anwendungen ist die fundamentale Voraussetzung für Transaktionen. Im Kontext der Informatik sind Transaktionen das Äquivalent zum Vertragsrecht. Tritt ein Problem auf, so wird die gesamte Berechnung einfach verworfen.“ Sie können sich Ausnahmen auch wie eine eingebaute „Undo“-Funktionalität vorstellen, da Sie mit etwas Sorgfalt verschiedene stabile Punkte („Wiederherstellungspunkte“) in Ihrem Programm definieren können. Scheitert ein Programmteil, so kehrt die Programmausführung zu einem bekannten stabilen Punkt des Programms zurück.

[15] Einer der wichtigsten Aspekte von Ausnahmen besteht darin, daß sie dem Programm nicht erlauben, dem gewöhnlichen Ausführungspfad zu folgen, wenn ein Problem auftritt. Dies war bei Sprachen wie C und C++ eine echte Schwierigkeit. Vor allem bei C gab es keine Möglichkeit, um die Ausführung eines Programmes bei Problemen zu unterbrechen, so daß Probleme über lange Zeiträume hinweg ignoriert werden und zu völlig unpassenden Zuständen führen konnten. Ausnahmen gestatten Ihnen dagegen, (falls es keine andere Möglichkeit gibt) eine Unterbrechung des Programmes zu zwingen und Ihnen mitzuteilen was passiert ist oder (im Idealfall) das Problem zu behandeln und das Programm in einen funktionsfähigen Zustand zurückzuversetzen.

13.2.1 Ausnahmen mit String-Argument

[16] Ausnahmeobjekte, kurz „Ausnahmen“, werden wie alle Java-Objekte mit Hilfe des `new`-Operators im dynamischen Speicher erzeugt, wobei Arbeitsspeicher allokiert und ein Konstruktor aufgerufen wird. Alle Standardausnahmen haben zwei Konstruktoren, nämlich einen Standardkonstruktor und einen Konstruktor, der ein Argument vom Typ `String` erwartet, so daß Sie dem Objekt eine aussagekräftige Information mitgeben können:

```
throw new NullPointerException(''t = null'');
```

Die mitgegebene Information kann später mit Hilfe verschiedener Methoden ausgewertet werden.

[17] Das Schlüsselwort `throw` hat einige interessante Auswirkungen. Nach dem Erzeugen des Ausnahmeobjektes per `new`, übergeben Sie die entsprechende Objektreferenz an `throw`. Die jeweilige Methode „gibt“ diese Referenz in gewisser Weise „zurück“, obwohl die Methode eigentlich nicht für

[2] °Jim Gray, Gewinner des Turing-Awards für die Beiträge seines Teams zu Transaktionen, in einem Interview auf *www.acmqueue.org*.

diesen Rückgabetyp konstruiert ist. Sie können sich die Ausnahmebehandlung als eine Art Rückgabemechanismus vorstellen, wobei Sie aber in Schwierigkeiten geraten, wenn Sie diese Analogie zu weit ausdehnen. Sie können durch Auswerfen einer Ausnahme auch einen gewöhnlichen Geltungsbereich verlassen. Stets wird ein Ausnahmeobjekt „zurückgegeben“ und die Programmsausführung verläßt den Geltungsbereich beziehungsweise die Methode.

[18] Damit endet die Ähnlichkeit mit der gewöhnlichen Wertrückgabe aus einer Methode, da das Programm nach dem Auswerfen einer Ausnahme an einen völlig anderen Ort „zurückkehrt“, als nach Beendigung eines normalen Methodenaufrufs. (Das Programm springt zu einem passenden Ausnahmebehandler, der auf dem Aufrufstapel viele Ebenen von der Stelle entfernt sein kann, an der die Ausnahme ausgeworfen wurde.)

[19] Darüber hinaus können Sie ein Objekt einer beliebigen von `Throwable` abgeleiteten Klasse auswerfen, der Wurzelklasse der Ausnahmeklassenhierarchie. Typischerweise gehört zu jeder Fehlersorte eine andere Ausnahmeklasse. Die Information über die Fehlerursache wird sowohl im Ausnahmeobjekt selbst, als auch implizit im Namen der Ausnahmeklasse transportiert, so daß in einem übergeordneten Kontext über die Behandlung der Ausnahme entschieden werden kann. (Häufig ist die Ausnahmeklasse die einzige verfügbare Information und das Ausnahmeobjekt enthält keine aussagekräftigen Daten.)

13.3 Abfangen einer Ausnahme

[20] Sie müssen zunächst das Konzept des *geschützten Bereiches* (*guarded region*) verstehen, bevor Sie das Abfangen einer Ausnahme nachvollziehen können. Ein geschützter Bereich ist ein Abschnitt des Quelltextes, bei dessen Verarbeitung Ausnahmen ausgeworfen werden können und steht unmittelbar vor den Anweisungen zur Behandlung dieser Ausnahmen.

13.3.1 Geschütze Bereiche: Die try-Klausel

[21] Wenn Sie im Geltungsbereich einer Methode (im Methodenkörper) eine Ausnahme auswerfen (oder eine in diesem Geltungsbereich aufgerufene Methode eine Ausnahme auswirft), so verläßt die Programmausführung den Geltungsbereich dieser Methode während die Ausnahme ausgeworfen wird. Wenn Sie verhindern möchten, daß eine `throw`-Anweisung zum Verlassen des Geltungsbereichs einer Methode führt, können Sie im Methodenkörper einen speziellen „Block“ definieren, um eventuell ausgeworfene Ausnahmen abzufangen. Dieser geschützte Bereich wird auch als „`try`-Klausel“ bezeichnet, da Sie „versuchen“ (*to try*) eine oder mehrere Methoden in diesem Bereich aufzurufen und ist ein gewöhnlicher, mit dem führenden Schlüsselwort `try` bezeichneter Geltungsbereich:

```
try {
    // Code that might generate exception
}
```

[22] Sorgfältige Fehlerprüfung in einer Programmiersprache ohne Ausnahmebehandlung bedeutet, daß Sie bei jedem Methodenaufruf entsprechende Anweisungen anlegen müssen, auch wenn Sie dieselbe Methode mehrmals aufrufen. Mit Ausnahmebehandlung setzen Sie alles in eine `try`-Klausel und fangen die Ausnahmen an einer Stelle ab. Der Quelltext ist dadurch einfacher zu schreiben und zu lesen, da das Ziel des Quelltextes nicht durch Fehlerprüfungen verschleiert wird.

13.3.2 Ausnahmebehandler: Die catch-Klausel

[23] Eine ausgeworfene Ausnahme muß selbstverständlich ein Ziel haben. Dieses Ziel ist der Ausnahmebehandler und zu jeder Ausnahme, die Sie abfangen möchten muß ein solcher Ausnahmebehandler definiert sein. Ausnahmebehandler schließen sich unmittelbar an die `try`-Klausel an und werden mit dem Schlüsselwort `catch`-bezeichnet („`catch`-Klausel“):

```
try {
    // Code that might generate exceptions
} catch(Type1 id1) {
    // Handle exceptions of Type 1
} catch(Type2 id2) {
    // Handle exceptions of Type 2
} catch(Type3 id3) {
    // Handle exceptions of Type 3
}
// etc...
```

[24] Eine `catch`-Klausel ähnelt einer Methode, die genau ein Argument erwartet. Der Bezeichner (`id1`, `id2` und so weiter) kann innerhalb der `catch`-Klausel wie das Argument einer Methode verwendet werden. Gelegentlich wird der Bezeichner nicht verwendet, da der Ausnahmetyp bereits genügend Informationen zur Ausnahmebehandlung enthält, muß aber dennoch stets vorhanden sein.

[25–26] Die Ausnahmebehandler müssen direkt an die `try`-Klausel anschließen. Wird eine Ausnahme ausgeworfen, so sucht der Ausnahmebehandlungsmechanismus nach dem ersten Behandler, dessen Argument zum Typ der Ausnahme paßt. Mit dem „Eintritt“ in den Geltungsbereich einer passenden `catch`-Klausel gilt die Ausnahme als behandelt. Die Suche nach einem passenden Ausnahmebehandler endet mit dem Verlassen der `catch`-Klausel. Im Gegensatz zur `switch`-Anweisung, deren verbleibende `case`-Zweige verarbeitet werden, wenn Sie eine `break`-Anweisungen setzen, wird nur die passende `catch`-Klausel ausgeführt. Beachten Sie, daß verschiedene Methoden in einer `try`-Klausel dieselbe Ausnahme auswerfen können, Sie aber nur einen Behandler brauchen.

13.3.2.1 Entscheidung: Abbruch oder Wiederaufnahme der Programmausführung?

[27] Die Theorie der Ausnahmebehandlung kennt zwei Modelle. Java unterstützt den *Programmabbruch*[3] (*termination*). Bei diesem Ansatz setzen Sie einen derart schwerwiegenden Fehler voraus, daß keine Möglichkeit besteht, an die Stelle zurückzukehren, an der die Ausnahme ausgeworfen wurde. Der Verursacher der Ausnahme hat entschieden, daß die Situation nicht mehr gerettet werden kann und *will nicht* zurückkehren.

[28] Das andere Modell ist die *Wiederaufnahme der Verarbeitung* des Programms und sieht vor, daß der Ereignisbehandler die Situation korrigiert, woraufhin der fehlgeschlagene Methodenaufruf in Erwartung eines nun erfolgreichen Verlaufs wiederholt wird. Sie wählen die Wiederaufnahme der Verarbeitung, wenn Sie hoffen, daß die Verarbeitung des Programmes nach Behandlung der Ausnahme fortgesetzt werden kann.

[29] Wenn Sie wiederaufnahmeartiges Verhalten in Java implementieren wollen, dürfen Sie beim Auftreten eines Fehlers keine Ausnahme auswerfen. Sie können statt dessen eine Methode aufrufen, die das Problem korrigiert. Alternativ können Sie Ihre `try`-Klausel in eine `while`-Schleife einsetzen, die solange immer wieder in die `try`-Klausel eintritt, bis ein befriedigendes Ergebnis vorliegt.

[3] Wie die meisten Sprachen, darunter auch C++, C#, Python und D.

[30] Die Geschichte der Softwareentwicklung hat gezeigt, daß Programmierer auf Betriebssystemen mit Unterstützung wiederaufnahmefähiger Ausnahmebehandlung, letztendlich abbruchartige Lösung verwenden und auf die Wiederaufnahme der Verarbeitung des Programms verzichten. Obwohl auf den ersten Blick attraktiv, ist die wiederaufnahmefähige Ausnahmebehandlung in der Praxis nicht besonders nützlich. Unter den Gründen überwiegt wahrscheinlich die entstehende Kopplung: Ein wiederaufnahmefähiger Ausnahmebehandler müßte „wissen", an welcher Stelle die Ausnahme ausgeworfen wurde und nicht-generische Anweisungen bezüglich dieser Stelle enthalten. Dadurch wird der Quelltext schwierig zu schreiben und zu pflegen, vor allem bei großen Softwaresystemen, bei denen eine Ausnahme von vielen Stellen ausgeworfen werden kann.

13.4 Ableiten eigener Ausnahmeklassen

[31] Sie sind nicht auf die vorhandenen Ausnahmeklassen beschränkt. Die Hierarchie der Java-Ausnahmeklassen kann nicht jeden Fehler vorhersehen, den Sie eventuell erfassen möchten, so daß Sie eigene Ausnahmen definieren können, um spezielle Probleme zu beschreiben, die in Ihrer Bibliothek auftreten können.

[32] Wenn Sie eine eigene Ausnahmeklasse definieren möchten, müssen Sie Ihre Klasse von einer der vorhandenen Ausnahmeklassen ableiten, vorzugsweise von einer solchen, mit ähnlicher Bedeutung (obwohl dies häufig nicht möglich ist). Die einfachste Möglichkeit, eine neue Ausnahmeklasse zu definieren, überläßt es dem Compiler, den Standardkonstruktor zu erzeugen und verlangt praktisch keinen Quelltext:

```
//: exceptions/InheritingExceptions.java
// Creating your own exceptions.

class SimpleException extends Exception {}

public class InheritingExceptions {
    public void f() throws SimpleException {
        System.out.println("Throw SimpleException from f()");
        throw new SimpleException();
    }
    public static void main(String[] args) {
        InheritingExceptions sed = new InheritingExceptions();
        try {
            sed.f();
        } catch(SimpleException e) {
            System.out.println("Caught it!");
        }
    }
} /* Output:
      Throw SimpleException from f()
      Caught it!
   *///:~
```

[33] Der Compiler erzeugt einen Standardkonstruktor, der automatisch (und unsichtbar) den Standardkonstruktor der Basisklasse aufruft. Natürlich bekommen Sie auf diese Weise keinen Konstruktor mit `String`-Argument, aber diese Variante kommt in der Praxis ohnehin nur selten vor. Wie Sie noch lernen werden, ist der Klassenname die wichtigste Eigenschaft einer Ausnahmeklasse. In den meisten Fällen ist eine Ausnahmeklasse wie die obige Klasse `SimpleException` ausreichend.

[34–35] Das Beispiel *InheritingExceptions.java* gibt sein Ergebnis über die Konsole (den Standardausgabekanal) aus. Unter Umständen bevorzugen Sie die Ausgabe von Fehlermeldungen über den

Standardfehlerkanal, indem Sie das `PrintStream`-Objekt `System.err` zur Ausgabe verwenden. `System.err` eignet sich in der Regel besser für Fehlermeldungen als `System.out`, da die Ausgabe über `System.out` umgeleitet werden kann. Wenn Sie Ihre Fehlermeldungen an `System.err` senden, werden sie nicht zusammen mit `System.out` umgeleitet und wahrscheinlicher vom Benutzer bemerkt. Sie können natürlich auch eine eigene Ausnahmeklasse mit einem `String`-Argument anlegen:

```
//: exceptions/FullConstructors.java
class MyException extends Exception {
    public MyException() {}
    public MyException(String msg) { super(msg); }
}

public class FullConstructors {
    public static void f() throws MyException {
        System.out.println("Throwing MyException from f()");
        throw new MyException();
    }
    public static void g() throws MyException {
        System.out.println("Throwing MyException from g()");
        throw new MyException("Originated in g()");
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException e) {
            e.printStackTrace(System.out);
        }
    }
} /* Output:
      Throwing MyException from f()
      MyException
      at FullConstructors.f(FullConstructors.java:11)
      at FullConstructors.main(FullConstructors.java:19)
      Throwing MyException from g()
      MyException: Originated in g()
      at FullConstructors.g(FullConstructors.java:15)
      at FullConstructors.main(FullConstructors.java:24)
   *///:~
```

[36] Der Quelltext wird nur geringfügig erweitert: Zwei Konstruktoren definieren, wie `MyException`-Objekte erzeugt werden. Der zweite Konstruktor ruft mit Hilfe des Schlüsselwortes `super` explizit den Basisklassenkonstruktor mit `String`-Argument auf.

[37] In beiden Ausnahmebehandlern wird die `Throwable`-Methode `printStackTrace()` aufgerufen (`Throwable` ist die Basisklasse von `Exception`). Wie Sie an der Ausgabe nachvollziehen können, liefert `printStackTrace()` Informationen über die Reihenfolge der Methodenaufrufe bis zum Auswerfen der Ausnahme. In diesem Beispiel wird die Ausgabe an `System.out` gesendet. Die argumentlose Standardversion `e.printStackTrace()` sendet die Ausgabe dagegen an den Standardfehlerkanal.

Übungsaufgabe 1: (2) Schreiben Sie eine Klasse, deren `main()`-Methode in einer `try`-Klausel ein Objekt der Klasse `Exception` auswirft. Übergeben Sie dem `Exception`-Konstruktor ein `String`-Argument. Fangen Sie die ausgeworfene Ausnahme mit einer `catch`-Klausel ab und geben Sie das

`String`-Argument aus. Legen Sie eine `finally`-Klausel an und geben Sie dort eine Meldung aus, um nachzuweisen, daß die `finally`-Klausel erreicht wurde. ■

Übungsaufgabe 2: (1) Definieren Sie eine Referenzvariable und initialisieren Sie sie mit `null`. Versuchen Sie, über diese Referenzvariable eine Methode aufzurufen. Setzen Sie den Methodenaufruf nun in eine `try`/`catch`-Kombination, um die Ausnahme abzufangen. ■

Übungsaufgabe 3: (1) Schreiben Sie ein Programm, welches eine Ausnahme vom Typ `ArrayIndexOutOfBoundsException` auswirft und abfängt. ■

Übungsaufgabe 4: (2) Definieren Sie durch Ableitung per `extends` eine eigene Ausnahmeklasse. Legen Sie für diese Klasse einen Konstruktor an, der ein `String`-Argument erwartet und in einem `String`-Feld im Ausnahmeobjekt speichert. Legen Sie eine Methode an, die den Inhalt des gespeicherten `String`-Objektes ausgibt. Legen Sie eine `try`/`catch`-Kombination an, um die Ausnahme auszuwerfen und abzufangen. ■

Übungsaufgabe 5: (3) Implementieren Sie ein eigenes wiederaufnahmeartiges Verhalten, indem Sie eine `while`-Schleife solange wiederholen, bis eine Ausnahme nicht mehr ausgeworfen wird. ■

13.4.1 Protokollieren von Ausnahmen

[38] Das Package `java.util.logging` gestattet Ihnen, die Ausgaben eines Programms zu protokollieren. Sie finden im Anhang von *http://www.mindview.net/Books/BetterJava* eine detaillierte Darstellung zum Thema „Protokollieren“, die Grundlagen sind aber einfach genug, um an dieser Stelle Verwendung zu finden:

```
//: exceptions/LoggingExceptions.java
// An exception that reports through a Logger.
import java.util.logging.*;
import java.io.*;

class LoggingException extends Exception {
    private static Logger logger =
        Logger.getLogger("LoggingException");
    public LoggingException() {
        StringWriter trace = new StringWriter();
        printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
}

public class LoggingExceptions {
    public static void main(String[] args) {
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Caught " + e);
        }
        try {
            throw new LoggingException();
        } catch(LoggingException e) {
            System.err.println("Caught " + e);
        }
    }
} /* Output: (85% match)
      Aug 30, 2005 4:02:31 PM LoggingException <init>
```

```
      SEVERE: LoggingException
      at LoggingExceptions.main(LoggingExceptions.java:19)

      Caught LoggingException
      Aug 30, 2005 4:02:31 PM LoggingException <init>
      SEVERE: LoggingException
      at LoggingExceptions.main(LoggingExceptions.java:24)

      Caught LoggingException
   *///:~
```

[39] Die statische `Logger`-Methode `getLogger()` liefert die Referenz auf ein mit ihrem `String`-Argument (üblicherweise der qualifizierte Name der von den Fehlermeldungen betroffenen Klasse) verknüpftes `Logger`-Objekt, welches seine Ausgabe an `System.out` sendet. Die einfachste Möglichkeit, dem `Logger`-Objekt Protokollmeldungen zu übergeben, besteht darin, die mit dem gewünschten Gewicht (*logging level*) assoziierte Methode aufzurufen, hier `severe()`. Wir wollen nun den Inhalt des Aufrufstapels beim Auswerfen der Ausnahme als Protokollmeldung verwenden. Die überladene `printStackTrace()`-Methode gibt allerdings kein `String`-Objekt zurück. Um ein solches `String`-Objekt zu bekommen, rufen wir diejenige Version von `printStackTrace()` auf, die ein `java.io.PrintWriter`-Objekt als Argument erwartet (die Einzelheiten werden in Kapitel 19 erklärt). Übergeben wir dem Konstruktor der Klasse `PrintWriter` ein `StringWriter`-Objekt, so können wir dessen `toString()`-Methode aufrufen, um den Inhalt des Aufrufstapels in Form eines `String`-Objektes bekommen.

[40] Auch wenn der Ansatz der Klasse `LoggingException` sehr komfortabel ist, da die Ausnahmeklasse die gesamte Protokollinfrastruktur enthält und somit automatisch sowie ohne Eingriff des Clientprogrammieres funktioniert, müssen Sie in der Regel Ausnahmen anderer Programmierer abfangen und protokollieren, die Protokollmeldung also im Ausnahmebehandler zusammensetzen:

```
//: exceptions/LoggingExceptions2.java
// Logging caught exceptions.
import java.util.logging.*;
import java.io.*;

public class LoggingExceptions2 {
    private static Logger logger =
        Logger.getLogger("LoggingExceptions2");
    static void logException(Exception e) {
        StringWriter trace = new StringWriter();
        e.printStackTrace(new PrintWriter(trace));
        logger.severe(trace.toString());
    }
    public static void main(String[] args) {
        try {
            throw new NullPointerException();
        } catch(NullPointerException e) {
            logException(e);
        }
    }
} /* Output: (90% match)
      Aug 30, 2005 4:07:54 PM LoggingExceptions2 logException
      SEVERE: java.lang.NullPointerException
      at LoggingExceptions2.main(LoggingExceptions2.java:16)
   *///:~
```

[41] Das Entwickeln einer eigenen Ausnahmeklasse hat noch weitere Facetten. Sie können zum Beispiel zusätzliche Konstruktoren und Komponenten anlegen:

```
//: exceptions/ExtraFeatures.java
// Further embellishment of exception classes.
import static net.mindview.util.Print.*;

class MyException2 extends Exception {
    private int x;
    public MyException2() {}
    public MyException2(String msg) { super(msg); }
    public MyException2(String msg, int x) {
        super(msg);
        this.x = x;
    }
    public int val() { return x; }
    public String getMessage() {
        return "Detail Message: "+ x + " "+ super.getMessage();
    }
}

public class ExtraFeatures {
    public static void f() throws MyException2 {
        print("Throwing MyException2 from f()");
        throw new MyException2();
    }
    public static void g() throws MyException2 {
        print("Throwing MyException2 from g()");
        throw new MyException2("Originated in g()");
    }
    public static void h() throws MyException2 {
        print("Throwing MyException2 from h()");
        throw new MyException2("Originated in h()", 47);
    }
    public static void main(String[] args) {
        try {
            f();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            g();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(MyException2 e) {
            e.printStackTrace(System.out);
            System.out.println("e.val() = " + e.val());
        }
    }
} /* Output:
      Throwing MyException2 from f()
      MyException2: Detail Message: 0 null
      at ExtraFeatures.f(ExtraFeatures.java:22)
      at ExtraFeatures.main(ExtraFeatures.java:34)
      Throwing MyException2 from g()
      MyException2: Detail Message: 0 Originated in g()
      at ExtraFeatures.g(ExtraFeatures.java:26)
      at ExtraFeatures.main(ExtraFeatures.java:39)
```

```
    Throwing MyException2 from h()
    MyException2: Detail Message: 47 Originated in h()
    at ExtraFeatures.h(ExtraFeatures.java:30)
    at ExtraFeatures.main(ExtraFeatures.java:44)
    e.val() = 47
*///:~
```

[42] Die Ausnahmeklasse `MyException2` verfügt über ein `int`-Feld `x`, zusammen mit einer Abfragemethode für dieses Feld sowie einem weiteren Konstruktor, der `x` bewertet. Die `Throwable`-Methode `getMessage()` wurde überschrieben und gibt eine interessantere und detailliertere Meldung zurück. Die `getMessage()`-Methode ist eine Art `toString()`-Methode für Ausnahmeklassen.

[43] Da Ausnahmen gewöhnliche Objekte sind, läßt sich die Ausgestaltung der Eigenschaften und Fähigkeiten Ihrer Ausnahmeklassen beliebig fortgesetzen. Bedenken Sie hierbei aber, daß alle diese Mühe vergeblich ist, wenn sich der Clientprogrammierer, der Ihr Package anwendet, darauf beschränkt, zu prüfen, ob die Ausnahme ausgeworfen wird und nichts weiter tut. (Die meisten Ausnahmen der Java-Bibliothek werden auf diese Weise benutzt.)

Übungsaufgabe 6: (1) Definieren Sie zwei Ausnahmeklassen, die sich jeweils selbst protokollieren. Zeigen Sie, daß Ihre Klassen funktionieren. ■

Übungsaufgabe 7: (1) Ändern Sie Übungsaufgabe 3, so daß die Ergebnisse in einer `catch`-Klausel protokolliert werden. ■

13.5 Deklaration des Ausnahmeverhaltens: Die throws-Klausel

[44] Java fordert Sie auf, die Clientprogrammierer, die Ihre Methode aufrufen, über die Ausnahmen zu informieren, welche die Methode eventuell auswirft. Das ist eine kultivierte Vorgehensweise, da der Aufrufer somit genau weiß, welche Anweisungen notwendig sind, um alle potentiellen Ausnahmen abzufangen. Ist der Quelltext verfügbar, so könnte der Clientprogrammierer nach `throw`-Anweisungen suchen, aber nicht jede Bibliothek wird mit Quelltext angeboten. Damit sich in dieser Hinsicht keine Schwierigkeiten einstellen können, legt Java eine spezielle Syntax fest (und erzwingt deren Anwendung), mit der Sie dem Clientprogrammierer höflich mitteilen können, welche Ausnahmen eine Methoden auswerfen kann, so daß der Clientprogrammierer diese Ausnahmen behandeln kann. Diese *Deklaration des Ausnahmeverhaltens* (*exception specification*) ist ein Teil der Methodendefinition und wird nach der Argumentliste angegeben.

[45] Die Deklaration des Ausnahmeverhaltens wird mit dem Schlüsselwort `throws` eingeleitet, dem eine kommaseparierte Liste der potentiellen Ausnahmentypen folgt, zum Beispiel:

```
void f() throws TooBig, TooSmall, DivZero { // ...
```

Andererseits bedeutet die Definition

```
void f() { // ...
```

daß diese Methode keine Ausnahmen auswirft. (Ausnahme von dieser Regel: Ausnahmen, deren Klassen von `RuntimeException` abgeleitet sind, dürfen überall ohne Deklaration ausgeworfen werden, siehe Unterabschnitt 13.7.1.)

[46] Schwindeln ist bei der Deklaration des Ausnahmeverhaltens nicht erlaubt. Enthält Ihre Methode eine Anweisung, die eine Ausnahme bewirken kann, behandelt diese Ausnahme aber nicht, so bemerkt der Compiler diese Situation und fordert Sie auf, diese Ausnahme entweder zu behandeln oder per `throws`-Klausel zu deklarieren, daß Ihre Methode eine Ausnahme dieses Typs auswerfen

kann. Das Erzwingen der Deklaration des Ausnahmeverhaltens gestattet Java, zur Übersetzungszeit ein gewisses Korrektheitsniveau hinsichtlich der Fehlerbehandlung zu gewährleisten.

[47] Eine kleine Schummelei ist doch möglich: Sie können eine Ausnahme deklarieren, die nicht ausgeworfen wird. Der Compiler nimmt Sie beim Wort und zwingt die Clientprogrammierer, so mit Ihrer Methode umzugehen, als ob sie diese Ausnahme tatsächlich auswerfen könnte. Dies hat die nützliche Auswirkung, daß Sie eine Ausnahme als Platzhalter deklarieren und die Ausnahme später tatsächlich auswerfen können, ohne daß existierende Quelltexte geändert werden müssen. Die Möglichkeit, eine Ausnahme als Platzhalter deklarieren zu können, ist auch für die Definition von abstrakten Basisklassen und Interfaces wichtig, deren abgeleitete beziehungsweise implementierende Klassen eventuell Ausnahmen auswerfen müssen. Ausnahmen, die zur Übersetzungszeit geprüft und deren Deklaration oder Behandlung zur Übersetzungszeit erzwungen wird, heißen *geprüfte Ausnahmen* (*checked exceptions*).

Übungsaufgabe 8: (1) Schreiben Sie eine Klasse mit einer Methode, die eine Ausnahme des in Übungsaufgabe 4 (Seite 357) definierten Typs auswirft. Versuchen Sie, die Klasse ohne Deklaration des Ausnahmeverhaltens ihrer Methode zu übersetzen und beobachten Sie, welche Meldung der Compiler ausgibt. Ergänzen Sie nun die erforderliche `throws`-Klausel. Probieren Sie Ihre Methode mit `throws`-Klausel in einer `try`/`catch`-Kombination aus. ■

13.6 Abfangen beliebiger Ausnahmen

[48] Sie können einen Ausnahmebehandler definieren, der eine Ausnahme beliebigen Typs abfängt, indem Sie in der `catch`-Klausel den Typ `Exception` wählen, die Wurzelklasse der Ausnahmeklassenhierarchie (es gibt zwar noch eine andere Wurzelklasse für Ausnahme*situationen*, nämlich `java.lang.Error`, aber in der Regel brauchen Sie nur `Exception`):

```
catch (Exception e) {
    System.out.println("Caught an exception");
}
```

Dieser Ausnahmebehandler fängt jede Ausnahme ab und sollte am Ende Ihrer Liste von Ausnahmebehandlern stehen, um eventuell folgenden `catch`-Klauseln nicht zuvor zu kommen.

[49–51] Da `Exception` die Basisklasse aller für den Programmierer wichtigen Ausnahmeklassen ist, erhalten Sie nur wenig spezifische Informationen über die Ausnahmesituation, können aber die Methoden von `Throwable` aufrufen, der Basisklasse von `Exception`: Die Methoden `getMessage()` und `getLocalizedMessage()` geben jeweils die Referenz auf ein `String`-Objekt zurück, welches die Detailbeschreibung der Ausnahme beziehungsweise eine an die Local-Einstellungen angepaßte Detailbeschreibung enthält. Die Methode `toString()` gibt die Referenz auf ein `String`-Objekt zurück, welches eine Kurzbeschreibung des `Throwable`-Objektes enthält, darunter die Detailbeschreibung, falls vorhanden.

[52] Die drei Versionen der überladenden `printStackTrace()`-Methode, `printStackTrace()`, `printStackTrace(PrintStream)` und `printStackTrace(PrintWriter)` geben keinen Wert zurück (`void`) und liefern die Ausgabe der `toString()`-Methode des `Throwable`-Objektes sowie den Inhalt des Aufrufstapels. Der Aufrufstapel dokumentiert die Reihenfolge der Methodenaufrufe bis zu der Stelle, an der die Ausnahme ausgeworfen wurde. Die erste Version übergibt ihre Ausgabe dem Standardfehlerkanal, die zweite und dritte Version dem Ausgabestrom Ihrer Wahl (zeichen- beziehungsweise byteorientiert; siehe Kapitel 19).

[53] Die Methode `fillInStackTrace()` zeichnet in „ihrem“ `Throwable`-Objekt Informationen über den aktuellen Zustand der Stackframes auf und ist beim erneuten Auswerfen eines Fehler oder einer

Ausnahme nützlich (siehe Unterabschnitt 13.6.2).

[54] Darüber hinaus haben Sie die Methoden der `Object` zur Verfügung, der Basisklasse von `Throwable` (und Wurzelklasse aller Java-Klassen): Ein für Ausnahmen praktischer Vertreter dieser Methoden ist `getClass()`, welche die Referenz auf ein Objekt zurückgibt, das die Klasse dieses Objektes repräsentiert. Die `getName()`- beziehungsweise `getSimpleName()`-Methode dieses sogenannten *Klassenobjektes* (siehe Abschnitt 15.2) liefert den Klassennamen mit beziehungsweise ohne Packagezugehörigkeit. Das folgende Beispiel zeigt die Anwendung der grundlegenden Methoden der Klasse `Exception`:

```
//: exceptions/ExceptionMethods.java
// Demonstrating the Exception Methods.
import static net.mindview.util.Print.*;

public class ExceptionMethods {
    public static void main(String[] args) {
        try {
            throw new Exception("My Exception");
        } catch(Exception e) {
            print("Caught Exception");
            print("getMessage():" + e.getMessage());
            print("getLocalizedMessage():" +
                  e.getLocalizedMessage());
            print("toString():" + e);
            print("printStackTrace():");
            e.printStackTrace(System.out);
        }
    }
} /* Output:
      Caught Exception
      getMessage():My Exception
      getgetLocalizedMessage():My Exception
      toString():java.lang.Exception: My Exception
      printStackTrace():
      java.lang.Exception: My Exception
      at ExceptionMethods.main(ExceptionMethods.java:8)
   *///:~
```

Die Methoden liefern nacheinander immer mehr Information. Jede der Methoden `getLocalizedMessage()`, `toString()` und `printStackTrace()` ruft ihren Vorgänger auf (siehe API-Dokumentation der Klasse `Throwable`).

Übungsaufgabe 9: (2) Definieren Sie drei neue Ausnahmeklassen. Schreiben Sie eine Klasse mit einer Methode, die Ausnahmen aller drei Typen auswirft. Rufen Sie diese Methode aus der `main()`-Methode auf und verwenden Sie dabei nur eine einzige `catch`-Klausel, um alle drei Typen von Ausnahmen abzufangen. ■

13.6.1 Aufbau, Inhalt und Auswertung des Aufrufstapels

[57] Die von der `printStackTrace()`-Methode gelieferte Information kann auch über die Methode `getStackTrace()` direkt abgefragt werden. Diese Methode gibt die Referenz auf ein Array von `StackTraceElement`-Objekten zurück, die jeweils einen Stackframe repräsentieren. Das Arrayelement mit dem Index Null ist das obere Ende des Stapels und stellt den letzten Methodenaufruf in der Reihenfolge dar, bevor das `Throwable`-Objekt erzeugt und ausgeworfen wurde. Das letzte Arrayelement am unteren Ende des Stapels repräsentiert den ersten Methodenaufruf der Reihenfolge

dar. Das folgende einfache Beispiel zeigt den Inhalt des Aufrufstapels:

```
//: exceptions/WhoCalled.java
// Programmatic access to stack trace information.

public class WhoCalled {
    static void f() {
        // Generate an exception to fill in the stack trace
        try {
            throw new Exception();
        } catch (Exception e) {
            for(StackTraceElement ste : e.getStackTrace())
                System.out.println(ste.getMethodName());
        }
    }
    static void g() { f(); }
    static void h() { g(); }
    public static void main(String[] args) {
        f();
        System.out.println("--------------------------------");
        g();
        System.out.println("--------------------------------");
        h();
    }
} /* Output:
      f
      main
      --------------------------------
      f
      g
      main
      --------------------------------
      f
      g
      h
      main
   *///:~
```

Wir begnügen uns mit der Ausgabe des Methodennamens, aber Sie können auch das gesamte `StackTraceElement`-Objekt ausgeben, das zusätzliche Informationen beinhaltet.

13.6.2 Erneutes Auswerfen einer Ausnahme

[58] Es kann vorkommen, daß Sie eine gerade abgefangene Ausnahme erneut auswerfen möchten, insbesondere wenn sich Ihre `catch`-Klausel auf den Typ `Exception` bezieht, also jede Ausnahme erfaßt. Da Sie die Referenz auf die gerade abgefangene Ausnahme bereits zur Verfügung haben, können Sie sie per `throw`-Anweisung einfach noch einmal auswerfen:

```
catch(Exception e) {
    System.out.println("An exception was thrown");
    throw e;
}
```

[59] Eine erneut ausgeworfene Ausnahme geht an die Ausnahmebehandler des nächsthöheren übergeordneten Kontextes über. Die `catch`-Klauseln der ursprünglichen `try`-Klausel werden ignoriert. Außerdem bleiben alle Eigenschaften und Fähigkeiten des Ausnahmeobjektes erhalten, so daß der

Ausnahmebehandler im übergeordneten Kontext, der Ausnahmen dieses spezifischen Typs abfängt, sämtliche Informationen über das Ausnahmeobjekt zur Verfügung hat.

[60–61] Beim erneuten Auswerfen einer Ausnahme, liefert die `printStackTrace()`-Methode keine Informationen über die Stelle des erneuten Auswurfs, sondern über den *Ursprung* der Ausnahme. Wenn Sie den aktuellen Zustand des Aufrufstapels erfassen möchten, können Sie die Methode `fillInStackTrace()` aufrufen, welche eine `Throwable`-Referenz auf das ursprüngliche Ausnahmeobjekt zurückgibt, das nun den aktuellen Zustand des Aufrufstapels enthält: Ein Beispiel:

```
//: exceptions/Rethrowing.java
// Demonstrating fillInStackTrace()

public class Rethrowing {
    public static void f() throws Exception {
        System.out.println("originating the exception in f()");
        throw new Exception("thrown from f()");
    }
    public static void g() throws Exception {
        try {
            f();
        } catch(Exception e) {
            System.out.println("Inside g(),e.printStackTrace()");
            e.printStackTrace(System.out);
            throw e;
        }
    }
    public static void h() throws Exception {
        try {
            f();
        } catch(Exception e) {
            System.out.println("Inside h(),e.printStackTrace()");
            e.printStackTrace(System.out);
            throw (Exception) e.fillInStackTrace();
        }
    }
    public static void main(String[] args) {
        try {
            g();
        } catch(Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
        try {
            h();
        } catch(Exception e) {
            System.out.println("main: printStackTrace()");
            e.printStackTrace(System.out);
        }
    }
} /* Output:
      originating the exception in f()
      Inside g(),e.printStackTrace()
      java.lang.Exception: thrown from f()
      at Rethrowing.f(Rethrowing.java:7)
      at Rethrowing.g(Rethrowing.java:11)
      at Rethrowing.main(Rethrowing.java:29)
      main: printStackTrace()
      java.lang.Exception: thrown from f()
```

```
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.g(Rethrowing.java:11)
        at Rethrowing.main(Rethrowing.java:29)

        originating the exception in f()
        Inside h(),e.printStackTrace()
        java.lang.Exception: thrown from f()
        at Rethrowing.f(Rethrowing.java:7)
        at Rethrowing.h(Rethrowing.java:20)
        at Rethrowing.main(Rethrowing.java:35)
        main: printStackTrace()
        java.lang.Exception: thrown from f()
        at Rethrowing.h(Rethrowing.java:24)
        at Rethrowing.main(Rethrowing.java:35)
    *///:~
```

Die Zeile in der die Methode `fillInStackTrace()` aufgerufen wird, ist der neue Ausgangspunkt der Ausnahme.

[62–63] Sie können auch eine andere, als die abgefangene Ausnahme auswerfen. Dabei erhalten Sie einen ähnlichen Effekt, als beim Aufrufen der `fillInStackTrace()`-Methode: Die Information über den Ursprung der ersteren Ausnahme geht verloren und Sie bekommen nur die Informationen, welche die neu ausgeworfene Ausnahme betreffen:

```
//: exceptions/RethrowNew.java
// Rethrow a different object from the one that was caught.

class OneException extends Exception {
    public OneException(String s) { super(s); }
}

class TwoException extends Exception {
    public TwoException(String s) { super(s); }
}

public class RethrowNew {
    public static void f() throws OneException {
        System.out.println("originating the exception in f()");
        throw new OneException("thrown from f()");
    }
    public static void main(String[] args) {
        try {
            try {
                f();
            } catch(OneException e) {
                System.out.println("Caught in inner try, e.printStackTrace()");
                e.printStackTrace(System.out);
                throw new TwoException("from inner try");
            }
        } catch(TwoException e) {
            System.out.println("Caught in outer try, e.printStackTrace()");
            e.printStackTrace(System.out);
        }
    }
} /* Output:
      originating the exception in f()
      Caught in inner try, e.printStackTrace()
      OneException: thrown from f()
      at RethrowNew.f(RethrowNew.java:15)
      at RethrowNew.main(RethrowNew.java:20)
```

```
        Caught in outer try, e.printStackTrace()
        TwoException: from inner try
        at RethrowNew.main(RethrowNew.java:25)
    *///:~
```

Die letzte Ausnahme „weiß“ nur, daß sie aus der inneren `try`-Klausel in der `main()`-Methode stammt, nicht aber von `f()`.

[64] Sie brauchen sich nicht um das Aufräumen der vorigen Ausnahme oder sonstiger Ausnahmen zu kümmern. Alle Ausnahmen sind Objekte im dynamischen Speicher und werden per `new`-Operator erzeugt, so daß die automatische Speicherbereinigung das Aufräumen übernimmt.

13.6.3 Verkettung (Schachtelung) von Ausnahmen

[65] Es kommt häufig vor, daß Sie eine Ausnahme abfangen und eine andere auswerfen wollen, wobei aber die Informationen über die ursprüngliche Ausnahme erhalten bleiben soll. Die Lösung ist die sogenannte *Verkettung von Ausnahmen* (*exception chaining*), oder treffender *Schachtelung von Ausnahmen*. Vor Version 1.4 des Java Development Kits mußte die Konservierung der ursprünglichen Ausnahme selbst programmiert werden. Aber nun kann jedes Objekt, einer der drei unmittelbar von `Throwable` abgeleiteten Klassen, per Konstruktor eine Referenz auf eine *Ursache* (*cause*) speichern. Als Ursache ist das ursprüngliche Ausnahmeobjekt vorgesehen. Durch die Übergabe und Speicherung der Ursache bleibt der Inhalt des Aufrufstapels bis zu ihrem Ursprung erhalten, obwohl Sie ein neues Ausnahmeobjekt erzeugen und auswerfen.

[66] Beachten Sie, daß nur die drei fundamentalen `Throwable`-Unterklassen `Error` (von der Laufzeitumgebung verwendet, um Systemfehler zu melden), `Exception` und `RuntimeException` Konstruktoren mit dem Argument `cause` definieren. Bei den übrigen Ausnahmetypen erwirken Sie die Verkettung dagegen nicht per Konstruktor, sondern mit Hilfe der `Throwable`-Methode `initCause()`.

[67] Das folgende Beispiel erzeugt zur Laufzeit dynamisch neue Felder:

```
//: exceptions/DynamicFields.java
// A Class that dynamically adds fields to itself.
// Demonstrates exception chaining.
import static net.mindview.util.Print.*;

class DynamicFieldsException extends Exception {}

public class DynamicFields {
    private Object[][] fields;
    public DynamicFields(int initialSize) {
        fields = new Object[initialSize][2];
        for(int i = 0; i < initialSize; i++)
            fields[i] = new Object[] { null, null };
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(Object[] obj : fields) {
            result.append(obj[0]);
            result.append(": ");
            result.append(obj[1]);
            result.append("\n");
        }
        return result.toString();
    }
    private int hasField(String id) {
        for(int i = 0; i < fields.length; i++)
```

```
        if(id.equals(fields[i][0]))
            return i;
    return -1;
}
private int getFieldNumber(String id) throws NoSuchFieldException {
    int fieldNum = hasField(id);
    if(fieldNum == -1)
        throw new NoSuchFieldException();
    return fieldNum;
}
private int makeField(String id) {
    for(int i = 0; i < fields.length; i++)
        if(fields[i][0] == null) {
            fields[i][0] = id;
            return i;
        }
    // No empty fields. Add one:
    Object[][] tmp = new Object[fields.length + 1][2];
    for(int i = 0; i < fields.length; i++)
        tmp[i] = fields[i];
    for(int i = fields.length; i < tmp.length; i++)
        tmp[i] = new Object[] { null, null };
    fields = tmp;
    // Recursive call with expanded fields:
    return makeField(id);
}
public Object getField(String id) throws NoSuchFieldException {
    return fields[getFieldNumber(id)][1];
}
public Object setField(String id, Object value)
    throws DynamicFieldsException {
    if(value == null) {
        // Most exceptions don't have a ''cause'' constructor.
        // In these cases you must use initCause(),
        // available in all Throwable subclasses.
        DynamicFieldsException dfe = new DynamicFieldsException();
        dfe.initCause(new NullPointerException());
        throw dfe;
    }
    int fieldNumber = hasField(id);
    if(fieldNumber == -1)
        fieldNumber = makeField(id);
    Object result = null;
    try {
        result = getField(id); // Get old value
    } catch(NoSuchFieldException e) {
        // Use constructor that takes ''cause'':
        throw new RuntimeException(e);
    }
    fields[fieldNumber][1] = value;
    return result;
}
public static void main(String[] args) {
    DynamicFields df = new DynamicFields(3);
    print(df);
    try {
        df.setField(''d'', ''A value for d'');
```

```
            df.setField(''number'', 47);
            df.setField(''number2'', 48);
            print(df);
            df.setField(''d'', ''A new value for d'');
            df.setField(''number3'', 11);
            print(''df: '' + df);
            print(''df.getField(\''d\'') : '' + df.getField(''d''));
            Object field = df.setField(''d'', null); // Exception
        } catch(NoSuchFieldException e) {
            e.printStackTrace(System.out);
        } catch(DynamicFieldsException e) {
            e.printStackTrace(System.out);
        }
    }
} /* Output:
      null: null
      null: null
      null: null

      d: A value for d
      number: 47
      number2: 48

      df: d: A new value for d
      number: 47
      number2: 48
      number3: 11

      df.getField(''d'') : A new value for d
      DynamicFieldsException
      at DynamicFields.setField(DynamicFields.java:64)
      at DynamicFields.main(DynamicFields.java:94)
      Caused by: java.lang.NullPointerException
      at DynamicFields.setField(DynamicFields.java:66)
      ... 1 more
   *///:~
```

[68] Jedes Objekt der Klasse `DynamicFields` enthält ein Array von `Object`/`Object`-Paaren. Die erste Komponente eines solchen Paares ist der Feldname (ein `String`-Objekt) und die zweite Komponente der Feldwert (ein Objekt beliebigen Typs, mit Ausnahme der primitiven Typen). Beim Erzeugen eines `DynamicFields`-Objektes treffen Sie eine sinnvolle Annahme über die Anzahl der benötigten Felder. Die `setField()`-Methode findet entweder das vorhandene Feld oder legt ein neues Feld an und speichert den übergebenen Wert. Reicht der Speicherplatz im Array nicht mehr aus, so erzeugt die `setField()`-Methode ein neues, um ein Feld längeres Array und kopiert die Elemente aus dem alten in das neue Array. Die `setField()`-Methode erzeugt, beim Versuch den Wert `null` zu speichern, ein Ausnahmeobjekt vom Typ `DynamicFieldsException`, setzt per `initCause()` eine Ausnahme vom Typ `NullPointerException` als Ursache ein und wirft die erstere Ausnahme aus.

[69] Die `setField()`-Methode fragt darüber hinaus per `getField()` den vorigen Wert des Feldes ab, wobei eine Ausnahme vom Typ `NoSuchFieldException` ausgeworfen werden kann. Der Clientprogrammierer ist beim Aufrufen der `getField()`-Methode selbst für die Behandlung dieser Ausnahme zuständig. In der `setField()`-Methode würde eine Ausnahme dieses Typs dagegen einen Programmierfehler anzeigen, weshalb sie, per Konstruktor mit `cause`-Argument, in einer Ausnahme vom Typ `RuntimeException` verpackt wird.

[70] Beachten Sie, daß die `toString()`-Methode ein Objekt der Klasse `StringBuilder` verwendet,

um ihren Rückgabewert zusammen zu setzen. Sie werden die Klasse `StringBuilder` in Kapitel 14 kennenlernen. In der Regel verwenden Sie ein `StringBuilder`-Objekt in `toString()`-Methoden, die Schleifen enthalten (wie im obigen Beispiel).

Übungsaufgabe 10: (2) Schreiben Sie eine Klasse mit zwei Methoden `f()` und `g()`. Methode `g()` wirft eine Ausnahme eines Typs aus, den Sie noch definieren müssen. Methode `f()` ruft Methode `g()` auf, fängt deren Ausnahme ab und wirft in der `catch`-Klausel eine neue Ausnahme aus (eines zweiten Typs, den Sie ebenfalls noch definieren müssen). Testen Sie Ihr Programm in der `main()`-Methode. ■

Übungsaufgabe 11: (1) Wiederholen Sie Übungsaufgabe 10, wobei Sie in der `catch`-Klausel die Ausnahme von Methode `g()` in einer Ausnahme vom Typ `RuntimeException` verpacken. ■

13.7 Die Standardausnahmen von Java

[71] Die Klasse `Throwable` beschreibt alle Klassen, deren Objekte als Ausnahmen ausgeworfen werden können. Es gibt zwei fundamentale, von `Throwable` abgeleitete Klassen: Die Klasse `Error` repräsentiert Fehler zur Übersetzungszeit sowie Systemfehler, die Sie nicht abfangen können (mit Ausnahme sehr spezieller Fälle). Die Klasse `Exception` ist die Wurzelklasse aller Ausnahmeklassen. Alle Methoden der Standardbibliothek von Java, jede Ihrer eigenen Methoden und die Laufzeitumgebung selbst können Ausnahmen vom Typ `Exception` auswerfen. Das Interesse des Java-Programmierers gilt somit in der Regel dem Ausnahmetyp `Exception`.

[72] Die API-Dokumentation ist die beste Möglichkeit, um sich einen Überblick über die verschiedenen Ausnahmetypen zu verschaffen. Es lohnt sich, die Dokumentation einmal durchzusehen, um einen Eindruck von den unterschiedlichen Typen zu bekommen, wobei Sie schnell erkennen werden, daß sich zwei Ausnahmen hauptsächlich durch ihre Namen unterscheiden. Die Anzahl der verfügbaren Java-Ausnahmen nimmt noch immer zu. Es ist nutzlos, sie in einem Buch aufzulisten. Jede neue Bibliothek, die Sie von einem Drittanbieter bekommen, definiert wahrscheinlich weitere eigene Ausnahmen. Es kommt darauf an, das Konzept zu verstehen und zu wissen, wie Sie mit Ausnahmen umgehen müssen.

[73] Die Grundidee besteht darin, daß der Name einer Ausnahme das aufgetretene Problem beschreibt und selbsterklärend gewählt werden soll. Nicht alle Ausnahmen liegen im Package `java.lang`. Es gibt Ausnahmen in den Packages `java.util`, `java.net` und `java.io`, wie Sie an den qualifizierten Klassennamen oder an den Basisklassen der Ausnahmen ablesen können. Die Ausnahmen des Ein-/Ausgabesystems sind beispielsweise alle von der Klasse `java.io.IOException` abgeleitet.

13.7.1 Ausnahmen vom Typ RuntimeException

[74] Das erste Beispiel in diesem Kapitel war (Seite 352):

```
if (t == null)
    throw new NullPointerException();
```

Der Gedanke, jede einer Methode übergebene Objektreferenz auf `null` prüfen zu müssen, da Sie nicht wissen können, ob der Aufrufer eine gültige Referenz übergeben hat, ist unangenehm. Glücklicherweise entfällt diese Prüfung beziehungsweise gehört zu den von der Laufzeitumgebung durchgeführten Standardprüfungen. Wird auf der `null`-Referenz eine Methode aufgerufen, so wirft die Laufzeitumgebung automatisch eine Ausnahme vom Typ `NullPointerException` aus. Die obige

`catch`-Klausel ist somit stets unnötig, obwohl Sie eventuell durch andere Prüfungen gewährleisten möchten, daß Ihr Programm eine Ausnahme vom Typ `NullPointerException` auswirft.

[75] Diese Kategorie umfaßt viele Ausnahmetypen. Sie werden von der Laufzeitumgebung automatisch ausgeworfen und müssen nicht in der Deklaration des Ausnahmeverhaltens erfaßt werden. Komfortablerweise sind alle Ausnahmen dieser Kategorie unter einer einzigen Basisklasse gruppiert: `RuntimeException`. Die Klasse `RuntimeException` ist ein perfektes Beispiel für Ableitung: Sie begründet eine Familie von Typen mit gemeinsamen Eigenschaften und gemeinsamem Verhalten. Sie brauchen niemals eine `throws`-Klausel, um bekannt zu geben, daß eine Methode eine Ausnahme vom Typ `RuntimeException` oder einem hiervon abgeleiteten Typ auswerfen kann, da die Ausnahmen unter `RuntimeException` sogenannte *ungeprüfte Ausnahmen* sind. Nachdem diese Ausnahmen Programmierfehler anzeigen, werden sie üblicherweise nicht abgefangen, da Sie die Fehler umgehend korrigieren. Wären Sie gezwungen, ungeprüfte Ausnahmen abzufangen, so könnte Ihr Quelltext zu unübersichtlich werden. Geprüfte Ausnahmen werden zwar nicht abgefangen, aber Sie können in Ihren eigenen Packages eventuell die eine oder andere Ausnahme dieses Typs auswerfen.

[76] Was geschieht, wenn Sie eine ungeprüfte Ausnahme nicht abfangen? Da der Compiler die Deklaration dieser Ausnahmen nicht verlangt, leuchtet ein, daß eine ungeprüfte Ausnahme den gesamten Aufrufstapel bis hin zur `main()`-Methode „durchläuft“, ohne abgefangen zu werden. Das folgende Beispiel zeigt was geschieht:

```
//: exceptions/NeverCaught.java
// Ignoring RuntimeExceptions.
// {ThrowsException}

public class NeverCaught {
    static void f() {
        throw new RuntimeException("From f()");
    }
    static void g() {
        f();
    }
    public static void main(String[] args) {
        g();
    }
} ///:~
```

[77] Sie können schon erkennen, daß eine Ausnahme vom Typ `RuntimeException` oder einem davon abgeleiteten Typ ein Sonderfall ist, weil der Compiler keine Deklaration des Ausnahmeverhaltens für diese ungeprüften Ausnahmen fordert. Die Ausgabe des obigen Beispiel wird an den Standardfehlerkanal gesendet:

```
Exception in thread "main" java.lang.RuntimeException: From f()
        at NeverCaught.f(NeverCaught.java:7)
        at NeverCaught.g(NeverCaught.java:10)
        at NeverCaught.main(NeverCaught.java:13)
```

Die Antwort lautet also: „Durchläuft“ eine ungeprüfte Ausnahme den Aufrufstapel bis hin zur `main()`-Methode, ohne abgefangen zu werden, so wird vor dem Programmende die `printStackTrace()`-Methode dieser Ausnahme aufgerufen.

[78] Denken Sie daran, daß Sie nur Ausnahmen vom Typ `RuntimeException` und Unterklassen ignorieren dürfen. Der Compiler wacht sorgfältig über die Behandlung aller ungeprüften Ausnahmen. Der Grund für diese Unterscheidung besteht darin, daß ungeprüfte Ausnahmen auf Programmierfehler hindeuten, das heißt:

- Fehler, gegen die Sie nichts tun können, zum Beispiel kann die Übergabe einer `null`-Referenz

außerhalb Ihres Einflusses liegen.

- Fehler, auf die Sie beim Schreiben Ihres Quelltextes achten müssen, zum Beispiel wird eine Ausnahme vom Typ `ArrayIndexOutOfBoundsException` ausgeworfen, wenn Sie die Länge eines Arrays nicht korrekt beachtet haben. Eine Ausnahme nach dem vorigen Punkt ist häufig die Ursache einer Ausnahme nach diesem Punkt.

[79] Sie sehen, welch beträchtlichen Nutzen Ausnahmen in diesen Fällen bedeuten, da sie Ihnen bei der Fehlersuche helfen.

[80] Es ist interessant, daß sich die Ausnahmebehandlung von Java nicht als Einzweckwerkzeug klassifizieren läßt. Einerseits wurde sie entworfen, um die ärgerlichen Laufzeitfehler behandeln zu können, deren Ursachen außerhalb der Kontrollmöglichkeiten Ihres Quelltextes liegen. Andererseits ist die Ausnahmebehandlung auch wesentlich für das Entdecken und Korrigieren bestimmter Programmierfehler, die der Compiler nicht feststellen kann.

Übungsaufgabe 12: (3) Ändern Sie das Beispiel *innerclasses/Sequence.java* (Abschnitt 11.1, Seite **??**), so daß es beim Hinzufügen zu vieler Elemente eine passende Ausnahme auswirft. ■

13.8 Die finally-Klausel

[81] Es gibt häufig einige Anweisungen, die Sie unabhängig davon ausführt haben wollen, ob in einer `try`-Klausel eine Ausnahme ausgeworfen wird oder nicht. Im allgemeinen beziehen sich diese Anweisungen nicht auf die Freigabe des Arbeitsspeichers, da sich die automatische Speicherbereinigung um diese Aufgabe kümmert. Sie erreichen diesen Effekt mit Hilfe einer `finally`-Klausel[4] nach dem letzten Ausnahmebehandler. Die vollständige Struktur der Ausnahmebehandlungsklauseln ist somit:

```
try {
    // The guarded region: Dangerous activities
    // that might throw A, B, or C
} catch (A a1) {
    // Handler for situation A
} catch (B b1) {
    // Handler for situation B
} catch (C c1) {
    // Handler for situation C
} finally {
    // Activities that happens every time
}
```

[82] Das folgende Programm zeigt, daß die `finally`-Klausel immer ausgeführt wird.

```
//: exceptions/FinallyWorks.java
// The finally clause is always executed.

class ThreeException extends Exception {}

public class FinallyWorks {
    static int count = 0;
    public static void main(String[] args) {
        while(true) {
            try {
                // Post-increment is zero first time:
```

[4] Die Ausnahmebehandlung bei C++ kennt keine `finally`-Klausel, sondern verläßt sich darauf, daß die Desktruktoren das Aufräumen übernehmen.

```
                if(count++ == 0)
                    throw new ThreeException();
                System.out.println("No exception");
            } catch(ThreeException e) {
                System.out.println("ThreeException");
            } finally {
                System.out.println("In finally clause");
                if(count == 2) break; // out of "while"
            }
        }
    }
} /* Output:
      ThreeException
      In finally clause
      No exception
      In finally clause
   *///:~
```

Sie sehen an der Ausgabe, daß die `finally`-Klausel immer ausgeführt wird, ob eine Ausnahme ausgeworfen wird oder nicht.

[83] Dieses Programm liefert auch einen Tip, wie Sie mit der Tatsache umgehen können, daß Sie die Verarbeitung eines bei Java-Ausnahmen nach dem Auswerfen einer Ausnahme nicht an der Stelle fortgesetzen können, an der die Ausnahme ausgeworfen wurde (siehe Unterunterabschnitt 13.3.2.1). Indem Sie Ihre `try`-Klausel in eine Schleife setzen, können Sie eine Bedingung festlegen, die erfüllt sein muß, bevor das Programm fortgesetzt wird. Sie können auch einen statischen Zähler oder einen anderen Mechanismus verwenden, um eine Anzahl von Schleifendurchläufen festzulegen, bevor das Programm aufgibt. Auf diese Weise können Sie Ihre Programm robuster gestalten.

13.8.1 Aufgabe der finally-Klausel

[84] Bei einer Sprache ohne automatische Speicherbereinigung und ohne automatische Destruktoraufrufe[5] ist die `finally`-Klausel wichtig, um die Freigabe des beanspruchten Arbeitsspeichers unabhängig vom Geschehen in der `try`-Klausel garantieren zu können. Da Java eine automatische Speicherbereinigung hat, ist die Freigabe von Speicherplatz fast nie ein Problem. Außerdem hat Java keine Destruktoren. Worin also besteht die Aufgabe der `finally`-Klausel bei Java?

[85] Die `finally`-Klausel ist notwendig, wenn Sie etwas anderes als den Arbeitsspeicher in seinen ursprünglichen Zustand zurückversetzen müssen, etwa das Schließen einer geöffneten Datei oder Netzwerkverbindung oder die Wiederherstellung des Bildschirminhaltes oder des Zustandes eines Schalters außerhalb des Programms, wie im folgenden Beispiel:

```
//: exceptions/Switch.java
import static net.mindview.util.Print.*;

public class Switch {
    private boolean state = false;
    public boolean read() { return state; }
    public void on() { state = true; print(this); }
    public void off() { state = false; print(this); }
    public String toString() { return state ? "on" : "off"; }
} ///:~
```

[5] Ein Destruktor ist eine Funktion, die immer dann aufgerufen wird, wenn ein Objekt nicht mehr benötigt wird. Sie wissen stets genau, zu welchem Zeitpunkt und an welcher Stelle der Destruktor aufgerufen wird. C++ ruft Destruktoren automatisch auf. C# (Java viel ähnlicher als C++) kann Destruktoren automatisch aufrufen.

```
//: exceptions/OnOffException1.java
public class OnOffException1 extends Exception {} ///:~

//: exceptions/OnOffException2.java
public class OnOffException2 extends Exception {} ///:~

//: exceptions/OnOffSwitch.java
// Why use finally?

public class OnOffSwitch {
    private static Switch sw = new Switch();
    public static void f()
        throws OnOffException1,OnOffException2 {}
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            f();
            sw.off();
        } catch(OnOffException1 e) {
            System.out.println("OnOffException1");
            sw.off();
        } catch(OnOffException2 e) {
            System.out.println("OnOffException2");
            sw.off();
        }
    }
} /* Output:
      on
      off
   *////:~
```

[86–87] Es kommt in diesem Beispiel darauf an, zu garantieren, daß der Schalter beim Verlassen der `main()`-Methode auf „aus“ steht. Daher enthalten die `try`-Klausel und die beiden `catch`-Klauseln je einmal die Anweisung `sw.off()`. Möglicherweise wird aber eine Ausnahme ausgeworfen, die von den beiden `catch`-Klauseln nicht erfaßt wird, so daß die Anweisung `sw.off()` nicht zur Verarbeitung kommt. Eine `finally`-Klausel gestattet Ihnen aber, die Aufräumanweisung an nur einer einzigen Stelle zu hinterlegen:

```
//: exceptions/WithFinally.java
// Finally Guarantees cleanup.

public class WithFinally {
    static Switch sw = new Switch();
    public static void main(String[] args) {
        try {
            sw.on();
            // Code that can throw exceptions...
            OnOffSwitch.f();
        } catch(OnOffException1 e) {
            System.out.println("OnOffException1");
        } catch(OnOffException2 e) {
            System.out.println("OnOffException2");
        } finally {
            sw.off();
        }
    }
} /* Output:
      on
```

```
        off
    *///:~
```

Die Anweisung `sw.off()` tritt nur noch an einer Stelle auf und wird garantiert verarbeitet, gleichgültig was in der `try`-Klausel geschieht.

[88–89] Selbst wenn die ausgeworfene Ausnahme von keiner `catch`-Klausel abgefangen wird, wird die `finally`-Klausel verarbeitet, bevor der Ausnahmebehandlungsmechanismus im nächsthöheren übergeordneten Kontext nach einer passenden `catch`-Klausel sucht:

```
//: exceptions/AlwaysFinally.java
// Finally is always executed.
import static net.mindview.util.Print.*;

class FourException extends Exception {}

public class AlwaysFinally {
    public static void main(String[] args) {
        print("Entering first try block");
        try {
            print("Entering second try block");
            try {
                throw new FourException();
            } finally {
                print("finally in 2nd try block");
            }
        } catch(FourException e) {
            System.out.println("Caught FourException in 1st try block");
        } finally {
            System.out.println("finally in 1st try block");
        }
    }
} /* Output:
      Entering first try block
      Entering second try block
      finally in 2nd try block
      Caught FourException in 1st try block
      finally in 1st try block
   *///:~
```

Die `finally`-Klausel wird auch bei Anwesenheit von `break`- und `continue`-Anweisungen verarbeitet. Beachten Sie, daß die `finally`-Klausel, kombiniert mit benannten `break`- und `continue`-Anweisungen den Bedarf einer `goto`-Anweisung bei Java eliminiert.

Übungsaufgabe 13: (2) Ändern Sie Übungsaufgabe 9, indem Sie eine `finally`-Klausel anlegen. Verifizieren Sie, daß die `finally`-Klausel auch beim Auswerfen einer Ausnahme vom Typ `NullPointerException` ausgeführt wird. ■

Übungsaufgabe 14: (2) Zeigen Sie, daß die Steuerung des Schalters im Beispiel *OnOffSwitch.java* (Seite 373) scheitern kann, indem Sie in der `try`-Klausel eine Ausnahme vom Typ `RuntimeException` auswerfen. ■

Übungsaufgabe 15: (2) Zeigen Sie, daß die Steuerung des Schalters im Beispiel *WithFinally.java* (Seite 373) nicht scheitert, indem Sie in der `try`-Klausel eine Ausnahme vom Typ `RuntimeException` auswerfen. ■

13.8.2 Verarbeitung der finally-Klausel vor der return-Anweisung

[90] Da die `finally`-Klausel stets verarbeitet wird, kann eine Methode mehrere Rückkehrpunkte (`return`-Anweisungen) haben und dennoch werden die Aufräumanweisungen garantiert ausgeführt:

```
//: exceptions/MultipleReturns.java
import static net.mindview.util.Print.*;

public class MultipleReturns {
    public static void f(int i) {
        print("Initialization that requires cleanup");
        try {
            print("Point 1");
            if(i == 1) return;
            print("Point 2");
            if(i == 2) return;
            print("Point 3");
            if(i == 3) return;
            print("End");
            return;
        } finally {
            print("Performing cleanup");
        }
    }
    public static void main(String[] args) {
        for(int i = 1; i <= 4; i++)
            f(i);
    }
} /* Output:
      Initialization that requires cleanup
      Point 1
      Performing cleanup
      Initialization that requires cleanup
      Point 1
      Point 2
      Performing cleanup
      Initialization that requires cleanup
      Point 1
      Point 2
      Point 3
      Performing cleanup
      Initialization that requires cleanup
      Point 1
      Point 2
      Point 3
      End
      Performing cleanup
   *///:~
```

Sie sehen an der Ausgabe, daß die Verarbeitung der `finally`-Klausel nicht davon abhängt, an welcher Stelle das Programm aus der Methode zurückkehrt.

Übungsaufgabe 16: (2) Ändern Sie das Beispiel *reusing/CADSystem.java* (Unterabschnitt 8.4.1, Seite 202f), um zu zeigen, daß die Aufräumarbeiten in der `finally`-Klausel auch bei einer `return`-Anweisung in der `try`-Klausel einer `try`/`finally`-Kombination anstandslos ausgeführt werden. ■

Übungsaufgabe 17: (3) Ändern Sie das Beispiel *polymorphism/Frog.java* (Unterabschnitt 9.3.2, Seite 236), um die Verarbeitung der Aufräumanweisungen mittels einer `try`/`finally`-Kombination

zu garantieren und zeigen Sie, daß die Garantie auch bei Rückkehr aus der Mitte der `try`-Klausel der `try`/`finally`-Kombination besteht. ■

13.8.3 Falle: Die verloren gegangene Ausnahme

[91] Die Implementierung des Ausnahmenmechanismus in Java hat leider eine Schwachstelle. Obwohl Ausnahmen Hinweise auf Probleme in Ihrem Programm sind und keinesfalls ignoriert werden sollten, ist es möglich, daß eine Ausnahme einfach „verloren geht". Diese Situation tritt bei einer besonderen Konfiguration mit der `finally`-Klausel ein:

```
//: exceptions/LostMessage.java
// How an exception can be lost.

class VeryImportantException extends Exception {
    public String toString() {
        return ''A very important exception!'';
    }
}

class HoHumException extends Exception {
    public String toString() {
        return ''A trivial exception'';
    }
}

public class LostMessage {
    void f() throws VeryImportantException {
        throw new VeryImportantException();
    }
    void dispose() throws HoHumException {
        throw new HoHumException();
    }
    public static void main(String[] args) {
        try {
            LostMessage lm = new LostMessage();
            try {
                lm.f();
            } finally {
                lm.dispose();
            }
        } catch(Exception e) {
            System.out.println(e);
        }
    }
} /* Output:
      A trivial exception
   *////:~
```

[92] Die Ausgabe liefert keinen Hinweis auf die Ausnahme vom Typ `VeryImportantException`, die in der `finally`-Klausel durch eine Ausnahme vom Typ `HoHumException` „ersetzt wird". Dies ist ein ernstes Problem, da eine Ausnahme offenbar spurlos verschwinden kann, insbesondere in einer subtileren und schwieriger zu untersuchenden Situation als in diesem Beispiel. C++ behandelt die Situtation, daß eine zweite Ausnahme ausgeworfen wird, bevor die erste Ausnahme behandelt wurde, als schwerwiegenden Programmierfehler. Vielleicht wird eine zukünftige Java-Version dieses Problem beheben (andererseits wird typischerweise jede Methode, die eine Ausnahme auswerfen kann, in eine `try`/`catch`-Kombination eingesetzt, wie `f()` im obigen Beispiel).

[93] Eine `return`-Anweisung in einer `finally`-Klausel ist eine noch einfachere Möglichkeit, um eine Ausnahme zu verlieren:

```
//: exceptions/ExceptionSilencer.java
public class ExceptionSilencer {
    public static void main(String[] args) {
        try {
            throw new Exception();
        } finally {
            // Using 'return' inside the finally block
            // will silence any thrown exception.
            return;
        }
    }
} ///:~
```

Das Programm liefert keine Ausgabe, obwohl eine *geprüfte* Ausnahme ausgeworfen wurde.

Übungsaufgabe 18: (3) Legen Sie im Beispiel *LostMessage.java* (Seite 376) eine zweite Ebene an, in der eine Ausnahme verloren geht, so daß die Ausnahme vom Typ `HoHumException` durch eine dritte Ausnahme „ersetzt wird“. ■

Übungsaufgabe 19: (2) Reparieren Sie das Beispiel *LostMessage.java* (Seite 376), indem Sie den Methodenaufruf in der `finally`-Klausel in eine `try`/`catch`-Kombination einsetzen. ■

13.9 Einschränkung des Ausnahmeverhaltens in abgeleiteten und implementierenden Klassen

[94] Eine überschriebene Methode darf nur die in ihrer Basisklassenversion deklarierten Ausnahmen auswerfen. Diese Einschränkung ist sinnvoll, da somit ein Programmteil, der mit einem Objekt der Basisklasse arbeitet, automatisch auch mit jedem Objekt einer abgeleiteten Klasse funktioniert. Dieses fundamentale Konzept der objektorientierten Programmierung gilt auch für Ausnahmen.

[95] Das folgende Beispiel demonstriert die zur Übersetzungszeit bestehenden Einschränkungen für Ausnahmen:

```
//: exceptions/StormyInning.java
// Overridden methods may throw only the exceptions
// specified in their base-class versions, or exceptions
// derived from the base-class exceptions.

class BaseballException extends Exception {}
class Foul extends BaseballException {}
class Strike extends BaseballException {}

abstract class Inning {
    public Inning() throws BaseballException {}
    public void event() throws BaseballException {
        // Doesn't actually have to throw anything
    }
    public abstract void atBat() throws Strike, Foul;
    public void walk() {} // Throws no checked exceptions
}

class StormException extends Exception {}
class RainedOut extends StormException {}
class PopFoul extends Foul {}
```

```
interface Storm {
    public void event() throws RainedOut;
    public void rainHard() throws RainedOut;
}

public class StormyInning extends Inning implements Storm {
    // OK to add new exceptions for constructors, but you
    // must deal with the base constructor exceptions:
    public StormyInning() throws RainedOut, BaseballException {}
    public StormyInning(String s) throws Foul, BaseballException {}
    // Regular methods must conform to base class:
    //! void walk() throws PopFoul {} //Compile error
    // Interface CANNOT add exceptions to existing
    // methods from the base class:
    //! public void event() throws RainedOut {}
    // If the method doesn't already exist in the
    // base class, the exception is OK:
    public void rainHard() throws RainedOut {}
    // You can choose to not throw any exceptions,
    // even if the base version does:
    public void event() {}
    // Overridden methods can throw inherited exceptions:
    public void atBat() throws PopFoul {}
    public static void main(String[] args) {
        try {
            StormyInning si = new StormyInning();
            si.atBat();
        } catch(PopFoul e) {
            System.out.println("Pop foul");
        } catch(RainedOut e) {
            System.out.println("Rained out");
        } catch(BaseballException e) {
            System.out.println("Generic baseball exception");
        }
        // Strike not thrown in derived version.
        try {
            // What happens if you upcast?
            Inning i = new StormyInning();
            i.atBat();
            // You must catch the exceptions from the
            // base-class version of the method:
        } catch(Strike e) {
            System.out.println("Strike");
        } catch(Foul e) {
            System.out.println("Foul");
        } catch(RainedOut e) {
            System.out.println("Rained out");
        } catch(BaseballException e) {
            System.out.println("Generic baseball exception");
        }
    }
} ///:~
```

[96] Der Konstruktor und die `event()`-Methode der Klasse `Inning` deklarieren jeweils eine Ausnahme vom Typ `BaseballException`, die aber nirgends ausgeworfen wird. Das ist zulässig, da Sie den Clientprogrammierer auf diese Weise zwingen können, Ausnahmen abzufangen, die unter Umständen von einer überschriebenen Version der `event()`-Methode ausgeworfen werden. Dasselbe gilt für

abstrakte Methoden, siehe `atBat()`.

[97] Das Interface *`Storm`* ist dadurch interessant, daß eine der deklarierten Methoden auch in `Inning` definiert ist, die andere aber nicht. Beide in *`Storm`* deklarierten Methoden deklarieren eine Ausnahme vom Typ `RainedOut`. Die Kombination der Ableitung der Klasse `StormyInning` von `Inning` bei gleichzeitiger Implementierung des Interfaces *`Storm`* zeigt, daß die dortige Deklaration der `event()`-Methode die Deklaration des Ausnahmeverhaltens der gleichnamigen Methode in der Klasse `Inning` nicht abwandeln kann. Das ist wiederum sinnvoll, da Sie andernfalls niemals wissen könnten, ob Sie die richtige Ausnahme abfangen, wenn Sie sich auf die Basisklasse beziehen. Ist eine im Interface deklarierte Methode, wie `rainHard()`, nicht in der Basisklasse definiert, so darf sie beliebige Ausnahmen deklarieren.

[98] Die Beschränkungen des Ausnahmeverhaltens in abgeleiteten und implementierenden Klassen gelten nicht für Konstruktoren. Wie Sie in der Klasse `StormyInning` sehen, kann der Konstruktor beliebige Ausnahmen auswerfen, ungeachtet des Ausnahmeverhaltens des Konstruktors der Basisklasse. Da ein Basisklassenkonstruktor aber stets auf die eine oder andere Weise aufgerufen wird (im obigen Beispiel wird automatisch der Standardkonstruktor aufgerufen), muß der Konstruktor in der abgeleiteten Klasse stets alle Ausnahmen des Basisklassenkonstruktors deklarieren.

[99] Ein Konstruktor einer abgeleiteten Klasse kann keine Ausnahmen abfangen, die „sein“ Basisklassenkonstruktor ausgeworfen hat.

[100] Die `walk()`-Methode der Klasse `StormyInning` läßt sich nicht übersetzen, da sie eine Ausnahme auswirft, ihre Basisklassenversion in `Inning` dagegen nicht. Wäre dieses Ausnahmeverhalten erlaubt, so könnten Sie einen Programmteil schreiben, der die `Inning`-Version der `walk()`-Methode ohne Ausnahmebehandlung aufruft und scheitert, wenn Sie ein Objekt einer von `Inning` abgeleiteten Klasse einsetzen, das Ausnahmen auswirft. Durch den Zwang eines konformen Ausnahmeverhaltens zwischen Methoden der Basisklasse und überschriebenen Methoden in abgeleiten Klassen bleibt die Ersetzbarkeit von Objekten erhalten.

[101] Die überschriebene `event()`-Methode zeigt, daß die Version einer Methode in einer abgeleiteten Klasse auf die Deklaration von Ausnahmen verzichten kann, die in der Basisklassenversion deklariert sind. Das ist in Ordnung, da kein Programmteil funktionsuntüchtig wird, der das Ausnahmeverhalten der Basisklassenversion berücksichtigt. Eine ähnliche Erklärung gilt für die Methode `atBat()`, deren Version in der Klasse `StormyInning` eine Ausnahme vom Typ `PopFoul` deklariert, welcher vom Ausnahmetyp `Foul` der Basisklassenversion von `atBat()` abgeleitet ist. Ruft ein Programmteil die `Inning`-Version von `atBat()` auf, so muß die potentielle `Foul`-Ausnahme abgefangen werden. Da `PopFoul` von `Foul` abgeleitet ist, erfaßt dieser Ausnahmebehandler auch `PopFoul`.

[102] Der letzte interessante Punkt befindet sich in der `main()`-Methode. Wenn Sie das `StormyInning`-Objekt über eine Referenzvariable vom Typ `StormyInning` „bedienen“, fordert der Compiler lediglich, die für diese Klasse spezifischen Ausnahmen abzufangen. Verwenden Sie aber eine Referenzvariable des Basistyps `Inning`, so verlangt der Compiler (korrekterweise), daß Sie die für den Basistyp spezifischen Ausnahmen abfangen. Alle diese Einschränkungen führen zu einer erheblich robusteren Ausnahmebehandlung.[6]

[103] Die Deklaration des Ausnahmeverhaltens einer Methode gehört nicht zu ihrer Signatur, die nur aus Methodenname und Argumenttypen besteht. Folglich können Sie Methoden nicht bezüglich ihres Ausnahmeverhaltens überladen. Die Deklaration des Ausnahmeverhaltens einer Methode in der Basisklasse bedeutet keineswegs, daß auch die Version in einer abgeleiteten Klasse Ausnahmeverhalten deklarieren muß. Dies ist ein deutlicher Unterschied von den Ableitungsregeln, nach denen eine

[6] ISO C++ wurde um ähnliche Einschränkungen ergänzt, welche verlangen, daß die Ausnahmen von Methoden in abgeleiteten Klassen identisch mit oder von den Ausnahmen der Basisklassenversion abgeleitet sein müssen. Dies ist ein Fall, in dem C++ tatsächlich deklariertes Ausnahmeverhalten zur Übersetzungszeit prüfen kann.

Methode in der Basisklasse auch in der abgeleiteten Klasse existieren muß. Mit anderen Worten: Die Deklaration des Ausnahmeverhaltens einer Methode kann durch Ableitung und Überschreiben „enger“ werden, aber nicht „weiter“. Dies ist genau das Gegenteil der Regel für die Schnittstellen von Klassen bei Ableitung.

Übungsaufgabe 20: (3) Ändern Sie das Beispiel *StormyInning.java*, indem Sie einen Ausnahmetyp `UmpireArgument` und Methoden definieren, die eine Ausnahme dieses Typs auswerfen. Testen Sie die modifizierte Hierarchie. ■

13.10 Konstruktoren mit Ausnahmeverhalten

[104] Es ist wichtig, sich stets zu fragen, ob im Falle einer Ausnahme alles ordentlich aufgeräumt wird. In der Regel sind Sie auf der sicheren Seite, aber bei Konstruktoren können Schwierigkeiten auftreten. Der Konstruktor versetzt das Objekt in einen sicheren Startzustand, kann aber eine Operation ausführen (zum Beispiel eine Datei öffnen), die nicht aufgeräumt wird, bevor das Objekt seine Aufgabe erfüllt hat und eine spezielle Methode zum Aufräumen aufgerufen wird. Wirft der Konstruktor nun eine Ausnahme aus, so werden die Anweisungen zum Aufräumen eventuell nicht verarbeitet. Sie müssen daher bei der Arbeit an Ihren Konstruktoren besonders sorgfältig vorgehen.

[105] Vielleicht denken Sie an eine Lösung per `finally`-Klausel, aber so einfach ist es nicht, da die Anweisungen in der `finally`-Klausel *immer* ausgeführt werden. Scheitert der Konstruktor auf halbem Weg, so ist unter Umständen ein Teil des Objektes noch nicht vorhanden, der aber in der `finally`-Klausel aufgeräumt wird.

[106] Das folgende Beispiel zeigt eine Klasse namens `InputFile`, die eine Datei öffnet und zeilenweise einliest. Die Klasse verwendet dabei die Klassen `FileReader` und `BufferedReader` aus dem `java.io`-Package der Standardbibliothek von Java (siehe Kapitel 19). Die Funktionsweise beider Klassen ist einfach genug, um ihre grundlegende Anwendung ohne Schwierigkeiten verstehen zu können:

```
//: exceptions/InputFile.java
// Paying attention to exceptions in constructors.
import java.io.*;

public class InputFile {
    private BufferedReader in;
    public InputFile(String fname) throws Exception {
        try {
            in = new BufferedReader(new FileReader(fname));
            // Other code that might throw exceptions
        } catch(FileNotFoundException e) {
            System.out.println("Could not open " + fname);
            // Wasn't open, so don't close it
            throw e;
        } catch(Exception e) {
            // All other exceptions must close it
            try {
                in.close();
            } catch(IOException e2) {
                System.out.println("in.close() unsuccessful");
            }
            throw e; // Rethrow
        } finally {
            // Don't close it here!!!
        }
    }
```

```
    public String getLine() {
        String s;
        try {
            s = in.readLine();
        } catch(IOException e) {
            throw new RuntimeException("readLine() failed");
        }
        return s;
    }
    public void dispose() {
        try {
            in.close();
            System.out.println("dispose() successful");
        } catch(IOException e2) {
            throw new RuntimeException("in.close() failed");
        }
    }
} ///:~
```

[107] Der Konstruktor der Klasse `InputFile` erwartet den Namen der zu öffnenden Datei als `String`-Argument. Der Konstruktor erzeugt in der `try`-Klausel ein `FileReader`-Objekt, wobei der Dateiname verwendet wird. Ein `FileReader`-Objekt ist erst programmiererfreundlich, nachdem Sie es in einem `BufferedReader`-Objekt verpackt haben. Einer der Vorteile der Klasse `InputFile` ist, daß Sie diese beiden Operationen kombiniert.

[108] Der Konstruktor der Klasse `FileReader` wirft eine Ausnahme vom Typ `FileNotFoundException` aus, wenn die Datei nicht auffindbar ist. In diesem Fall brauchen Sie die Datei nicht zu schließen, da sie nicht geöffnet wurde. Jede *andere* `catch`-Klausel muß die Datei dagegen schließen, da sie zum Zeitpunkt des Eintritts in den Ausnahmebehandler geöffnet war. (Die Anlegenheit wird komplizierter, wenn mehr als eine Methode eine Ausnahme vom Typ `FileNotFoundException` auswerfen kann. In diesem Fall teilen Sie die Methoden in der Regel auf mehrere `try`-Klauseln auf.) Die `close()`-Methode kann selbst eine Ausnahme auswerfen und steht in einer eigenen `try`/`catch`-Kombination, obwohl sie im Geltungsbereich einer `catch`-Klausel aufgerufen wird (für den Compiler lediglich ein Paar geschweifter Klammern). Nach einigen lokalen Operationen wird die Ausnahme erneut ausgeworfen. Dieses Verhalten ist angemessen, da der Konstruktor gescheitert ist und die aufrufende Methode nicht davon ausgehen darf, daß das Objekt ordnungsgemäß erzeugt wurde und gültig ist.

[109] In diesem Beispiel ist die `finally`-Klausel definitiv *nicht* die richtige Stelle, um die Datei zu schließen, da die Datei *jedesmal* beim Verlassen des Konstruktors geschlossen werden würde. Wir möchten aber, daß die Datei über den gesamten Lebenszyklus des Objektes hinweg geöffnet bleibt.

[110] Die `getLine()`-Methode gibt ein `String`-Objekt zurück, das den Inhalt der nächsten Zeile der Datei enthält und ruft die `readLine()`-Methode auf, die eine Ausnahme vom Typ `IOException` auswerfen kann. Da diese Ausnahme abgefangen wird, wirft `getLine()` keine Ausnahme aus. Eine Design-Entscheidung bei Ausnahmen betrifft die Frage, ob die Ausnahme auf der Ebene ihrer Entstehung vollständig behandelt werden, nur teilweise behandelt und erneut (oder statt dessen eine andere Ausnahme) ausgeworfen oder einfach in unveränderter Form erneut ausgeworfen werden soll. Das Neuauswerfen kann, falls passend, die Programmierarbeit natürlich vereinfachen.

[111] Der Clientprogrammierer muß die `dispose()`-Methode aufrufen, wenn das `InputFile`-Objekt nicht mehr gebraucht wird. Diese Methode gibt die Resourcen des Systems (beispielsweise ein Dateihandle) frei, die von den `BufferedReader`- und/oder `FileReader`-Objekten beansprucht wurden. Dieser Schritt ist nicht sinnvoll, bevor Sie das `InputFile`-Objekt nicht mehr brauchen. Falls Sie sich überlegen, solche Funktionalität in einer `finalize()`-Methode zu implementieren, dann den-

ken Sie daran, daß es keine Garantie für das Aufrufen dieser Methode gibt (und selbst wenn Sie sich des Aufrufs sicher sein können, wissen Sie nicht zu welchem Zeitpunkt der Aufruf erfolgt; siehe Abschnitt 6.5). Ein Nachteil von Java ist, daß es, mit Ausnahme der Speicherbereinigung, keine automatische Aufräumfunktionalität gibt. Sie müssen die Clientprogrammierer darüber in Kenntnis setzen, daß sie selbst zuständig sind.

[112] Die Schachtelung von `try`-Klauseln ist die sicherste Möglichkeit, eine Klasse zu verwenden, deren Konstruktor Ausnahmen auswerfen kann und die Aufräumtätigkeiten benötigt:

```
//: exceptions/Cleanup.java
// Guaranteeing proper cleanup of a resource.

public class Cleanup {
    public static void main(String[] args) {
        try {
            InputFile in = new InputFile("Cleanup.java");
            try {
                String s;
                int i = 1;
                while((s = in.getLine()) != null)
                    ; // Perform line-by-line processing here...
            } catch(Exception e) {
                System.out.println("Caught Exception in main");
                e.printStackTrace(System.out);
            } finally {
                in.dispose();
            }
        } catch(Exception e) {
            System.out.println("InputFile construction failed");
        }
    }
} /* Output:
      dispose() successful
   *////:~
```

[113] Sehen Sie sich die Logik genau an: Die Erzeugung des `InputFile`-Objektes steht in einer eigenen `try`-Klausel. Scheitert der Aufruf des Konstruktors, so wird die äußere `catch`-Klausel verarbeitet und die `dispose()`-Methode wird nicht aufgerufen. Verläuft die Objekterzeugung aber erfolgreich, so möchten Sie sicherstellen, daß das Objekt aufgeräumt wird. Daher legen Sie unmittelbar nach der Objekterzeugung eine neue `try`-Klausel an. Die `finally`-Klausel mit dem Aufruf der `dispose()`-Methode gehört zur *inneren* `try`-Klausel. Somit wird die `finally`-Klausel beim Scheitern des Konstruktors nicht aufgerufen, andererseits aber immer, wenn das Objekt erfolgreich erzeugt wurde.

[114] Dieser „Aufräumansatz“ sollte auch dann angewendet werden, wenn der Konstruktor keine Ausnahme auswirft. Die Grundregel lautet: „Die `try`/`finally`-Kombination beginnt unmittelbar nach dem Erzeugen des Objektes beim dem Aufräumtätigkeiten erforderlich sind“:

```
//: exceptions/CleanupIdiom.java
// Each disposable object must be followed by a try-finally

class NeedsCleanup { // Construction can't fail
    private static long counter = 1;
    private final long id = counter++;
    public void dispose() {
        System.out.println("NeedsCleanup " + id + " disposed");
    }
}
```

```
class ConstructionException extends Exception {}

class NeedsCleanup2 extends NeedsCleanup {
    // Construction can fail:
    public NeedsCleanup2() throws ConstructionException {}
}

public class CleanupIdiom {
    public static void main(String[] args) {
        // Section 1:
        NeedsCleanup nc1 = new NeedsCleanup();
        try {
            // ...
        } finally {
            nc1.dispose();
        }

        // Section 2:
        // If construction cannot fail you can group objects:
        NeedsCleanup nc2 = new NeedsCleanup();
        NeedsCleanup nc3 = new NeedsCleanup();
        try {
            // ...
        } finally {
            nc3.dispose(); // Reverse order of construction
            nc2.dispose();
        }

        // Section 3:
        // If construction can fail you must guard each one:
        try {
            NeedsCleanup2 nc4 = new NeedsCleanup2();
            try {
                NeedsCleanup2 nc5 = new NeedsCleanup2();
                try {
                    // ...
                } finally {
                    nc5.dispose();
                }
            } catch(ConstructionException e) { // nc5 constructor
                System.out.println(e);
            } finally {
                nc4.dispose();
            }
        } catch(ConstructionException e) { // nc4 constructor
            System.out.println(e);
        }
    }
} /* Output:
      NeedsCleanup 1 disposed
      NeedsCleanup 3 disposed
      NeedsCleanup 2 disposed
      NeedsCleanup 5 disposed
      NeedsCleanup 4 disposed
   *////:~
```

[115] Der erste Abschnitt in der `main()`-Methode ist leicht zu verstehen: Der Erzeugung des Objektes folgt eine `try`/`finally`-Kombination. Kann die Erzeugung eines Objektes nicht scheitern, so ist keine `catch`-Klausel erforderlich. Der zweite Abschnitt zeigt, daß Objekte, deren Konstruktoren

keine Ausnahmen auswerfen, sowohl bei ihrer Erzeugung als auch beim Aufräumen zusammengefaßt werden können.

[116] Der dritte Abschnitt zeigt den Umgang mit Objekten deren Konstruktoren Ausnahmen auswerfen können und die aufgeräumt werden müssen. Die Angelegenheit wird unübersichtlich, wenn Sie die Situation korrekt handhaben wollen, da jeder Konstruktoraufruf in einer eigenen `try`/`catch`-Kombination stehen muß und jeder Objekterzeugung eine `try`/`finally`-Kombination folgen muß, um das Aufräumen zu garantieren.

[117] Die unübersichtliche Ausnahmebehandlung ist in diesem Fall ein gewichtiges Argument für Konstruktoren ohne Ausnahmeverhalten, aber dies ist nicht immer möglich.

[118] Beachten Sie, daß weitere `try`-Klauseln notwendig werden, wenn die `dispose()`-Methode Ausnahmen auswerfen kann. Sie müssen über alle Möglichkeiten gründlich nachdenken und Ihr Programm vor jeder einzelnen schützen.

Übungsaufgabe 21: (2) Zeigen Sie, daß ein Konstruktor in einer abgeleiteten Klasse keine Ausnahmen „seines“ Basisklassenkonstruktors abfangen kann. ■

Übungsaufgabe 22: (2) Schreiben Sie eine Klasse `FailingConstructor` mit einem Konstruktor, der während der Objekterzeugung scheitert und eine Ausnahme auswirft. Legen Sie in der `main()`-Methode die erforderlichen Anweisungen an, um das Programm gegen diesen Fehler zu schützen. ■

Übungsaufgabe 23: (4) Ergänzen Sie Übungsaufgabe 22 um eine Klasse mit einer `dispose()`-Methode. Ändern Sie die Klasse `FailingConstructor`, so daß der Konstruktor ein Objekt dieser ergänzten Klasse als Komponente erzeugt (Komposition), anschließend eine Ausnahme auswirft und ein zweites Objekt der ergänzten Klasse als Komponente erzeugt. Legen Sie die erforderlichen Anweisungen an, um das Programm gegen diesen Fehler zu schützen und zeigen Sie in der `main()`-Methode, daß Sie alle möglichen Ausfallsituationen erfaßt haben. ■

Übungsaufgabe 24: (3) Legen Sie in der Klasse `FailingConstructor` eine `dispose()`-Methode un die erforderlichen Anweisungen an, um ein Objekt dieser Klasse korrekt zu verwenden. ■

13.11 Passung zwischen Ausnahme und catch-Klausel

[119] Wird eine Ausnahme ausgeworfen, so durchsucht der Ausnahmebehandlungsmechanismus die „nächsten“ Ausnahmebehandler in der Reihenfolge ihrer Definition. Findet der Mechanismus eine passende `catch`-Klausel, so gilt die Ausnahme als behandelt und die Suche wird eingestellt.

[120] Der Abgleich zwischen Ausnahme und Ausnahmebehandler setzt keine perfekte Übereinstimmung voraus. Eine Ausnahme eines abgeleiteten Typs paßt zu einer `catch`-Klausel für den Basistyp, wie das folgende Beispiel zeigt:

```
//: exceptions/Human.java
// Catching exception hierarchies.

class Annoyance extends Exception {}
class Sneeze extends Annoyance {}

public class Human {
    public static void main(String[] args) {
        // Catch the exact type:
        try {
            throw new Sneeze();
        } catch(Sneeze s) {
            System.out.println("Caught Sneeze");
```

```
            } catch(Annoyance a) {
                System.out.println("Caught Annoyance");
            }
            // Catch the base type:
            try {
                throw new Sneeze();
            } catch(Annoyance a) {
                System.out.println("Caught Annoyance");
            }
        }
    } /* Output:
          Caught Sneeze
          Caught Annoyance
       *///:~
```

[121] Die `Sneeze`-Ausnahme wird von der ersten passenden `catch`-Klausel abgefangen, im ersten Teil dieses Beispiels natürlich der ersten `catch`-Klausel. Wenn Sie die erste `catch`-Klausel entfernen, so daß nur noch die zweite Klausel für den Ausnahmetyp `Annoyance` stehen bleibt, funktioniert das Programm noch immer, da der Basistyp von `Sneeze` abgefangen wird. Mit anderen Worten: Die Klause `catch(Annoyance)` fängt Ausnahmen vom Typ `Annoyance` und jedem von `Annoyance` abgeleiteten Typ ab. Das ist ein praktisches Verhalten, denn wenn Sie sich entscheiden, daß eine Methode weitere Ausnahmen abgeleiteten Typs auswerfen kann, brauchen die Clientprogrammierer ihre Quelltext nicht ändern, solange sie Ausnahmen des Basistyps abfangen.

[122] Wenn Sie versuchen, eine `catch`-Klausel eines abgeleiteten Types hinter der `catch`-Klausel des Basistyps zu „verstecken“:

```
try {
    throw new Sneeze();
} catch (Annoyance a) {
    // ...
} catch (Sneeze s) {
    // ...
}
```

gibt der Compiler eine Fehlermeldung aus: `exception Sneeze has already been caught`. Der Compiler erkennt, daß die `catch`-Klausel für den Typ `Sneeze` unter keinen Umständen erreicht werden kann.

Übungsaufgabe 25: (2) Legen Sie eine drei Ebene tiefe Hierarchie von Ausnahmeklassen an. Schreiben Sie nun eine Basisklasse `A` mit einer Methode, die eine Ausnahme vom Basistyp Ihrer Hierarchie auswirft. Leiten Sie eine Klasse `B` von `A` ab und überschreiben Sie die Methode, so daß Sie eine Ausnahme aus der zweiten Ebene Ihrer Hierarchie auswirft. Wiederholen Sie diesen Schritt, indem Sie eine Klasse `C` aus `B` ableiten. Erzeugen Sie in der `main()`-Methode ein Objekt der Klasse `C` und wandeln Sie die Objektreferenz in `A` um. Rufen Sie die Methode auf. ■

13.12 Alternative Fehlerbehandlungsmöglichkeiten

[123] Ein Ausnahmebehandlungsmechanismus ist eine Falltür, die Ihnen gestattet, die normale Verarbeitungsreihenfolge von Anweisungen abzubrechen, wenn eine „Ausnahmesituation“ eintritt, durch die der reguläre Programmablauf nicht mehr möglich oder erwünscht ist. Eine Ausnahme repräsentiert eine Situation, die die gegenwärtig ausgeführte Methode nicht behandeln kann. Ausnahmebehandlungsmechanismen wurden entwickelt, da die Behandlung jedes möglicherweise auftretenden

Fehlers bei jedem Funktionsaufruf zu mühsehlig war und sich die Programmierer einfach nicht darum gekümmert hatten. Die Fehler wurden ignoriert. An dieser Stelle sollte vermerkt werden, daß eine für die Programmierer komfortable Möglichkeit zur Fehlerbehandlung, von Anfang an einer der Hauptgründe für die Entwicklung des Ausnahmebehandlungsmechanismus waren.

[124] Eine wichtige Richtlinie für die Ausnahmebehandlung lautet: „Fangen Sie keine Ausnahme ab, wenn Sie nicht wissen wie Sie sie behandeln sollen“. Tatsächlich besteht eines der wesentlichen Ziele der Ausnahmebehandlung darin, die Fehlerbehandlungsanweisungen von der Stelle des Fehlerauftritts zu entfernen. Sie können sich dadurch in einem Abschnitt von Anweisungen auf die eigentliche Funktionalität und in einem anderen auf die Behandlung von Fehlern konzentrieren. Der eigentliche Quelltext ist nun nicht mehr mit Anweisungen zur Fehlerbehandlung überhäuft und läßt sich viel besser verstehen und pflegen. Die Ausnahmebehandlung reduziert außerdem tendentiell den Umfang der Fehlerbehandlungsanweisungen, da ein Ausnahmebehandler auf mehrere Fehlerquellen angewendet werden kann.

[125] Die geprüften Ausnahmen verkomplizieren die Situation ein wenig, da Sie gezwungen sind, eine `try`/`catch`-Kombinationen anzulegen, obwohl Sie noch nicht bereit sind, den Fehler zu behandeln. Das führt zu dem Problem des „Verschluckens“ einer möglicherweise wichtigen Ausnahme:

```
try {
    // ... to do something useful
} catch(ObligatoryException e) {} // Gulp!
```

[126] Die Programmierer (mich inbegriffen, in der ersten Auflage dieses Buches) können die Ausnahme nun einfach „verschlucken“. Dies geschieht häufig unbeabsichtigt, aber der Compiler ist anschließend zufrieden und die Ausnahme geht verloren, wenn Sie nicht zu der `catch`-Klausel zurückkehren und den Fehler korrigieren. Die Ausnahme wird zwar ausgeworfen, verschwindet aber spurlos, wenn sie „verschluckt“ wird. Da der Compiler Sie zwingt, die Ausnahme vor Ort zu behandeln, scheint das „Verschlucken“ die einfachste Lösung zu sein, ist aber das schlechteste Verfahren, das Sie wählen können.

[127] Erschrocken angesichts der Erkenntnis, daß mir dieser Fehler unterlaufen war, „korrigierte“ ich die betroffenen `catch`-Klauseln in der zweiten Auflage durch Ausgabe des Aufrufstapels im Ausnahmebehandler (Sie sehen diese Korrektur, dort wo sie angemessen ist, noch immer in vielen Beispielen in diesem Kapitel). Diese Vorgehensweise ist zwar nützlich, um das Vorkommen von Ausnahmen zu verfolgen, dokumentiert aber auch, daß Sie im Grunde genommen nicht wissen, wie Sie die Ausnahme an der Stelle ihres Auftretens im Quelltext behandeln sollen. In diesem Abschnitt diskutieren wir einige Aspekte, Schwierigkeiten und Behandlungsmöglichkeiten von geprüften Ausnahmen.

[128] Das Thema wirkt einfach, ist aber in Wirklichkeit nicht nur kompliziert, sondern auch von Unbeständigkeit geprägt. Es gibt auf beiden Seiten des Grabens standhaft verwurzelte Kollegen, die felsenfest davon überzeugt sind, daß die richtigen Ansichten (ihre) offensichtlich und unstrittig sind. Ich vermute den Grund für den einen oder anderen dieser Standpunkte im individuellen Nutzen des Übergangs von einer ärmlich typisierten Sprache wie C vor der ANSI-Standardisierung zu einer stark und statisch typisierten Sprache (Typprüfung zur Übersetzungzeit) wie C++ oder Java. Wenn Sie (wie ich) diesen Übergang vollzogen haben, sind die Vorteile derart dramatisch, daß die statische Typprüfung scheinbar die beste Lösung fast aller Probleme darstellt. Ich möchte an dieser Stelle ein wenig über meine eigene Entwicklung berichten, durch deren Verlauf ich den *absoluten* Wert der statischen Typprüfung mittlerweise in Frage stelle. Die statische Typprüfung ist natürlich meistens nützlich, aber es gibt eine undeutliche Abgrenzung zu den Fällen hin, in denen die statische Typprüfung im Weg steht und zum Hindernis wird (eins meiner Lieblingszitate lautet: „Alle Modelle sind falsch, aber einige sind nützlich.“)

13.12.1 Geschichte der Fehlerbehandlung

[129] Das Konzept der Ausnahmebehandlung stammt aus Programmiersprachen wie PL/1 und Mesa, tauchte später in CLU, Smalltalk, Modula-3, Ada Eiffel, C++, Python, Java sowie in Ruby und C# auf, die Java folgten. Das Design bei Java ähnelt dem Design bei C++ mit Ausnahme der Stellen, an denen die Java-Designer ein Eindruck hatten, der Ansatz von C++ könne Schwierigkeiten verursachen.

[130] Die Ausnahmebehandlung wurde, befürwortet von Bjarne Stroustrup, dem ursprünglichen Autor der Sprache, erst relativ spät im Standardisierungsprozeß von C++ berücksichtigt, um den Programmierern ein Framework zur Verfügung zu stellen, daß mehr Aussicht auf Anwendung zur Fehlerbehandlung und Wiederherstellung eines funktionsfähigen Programmzustandes hatte. Das Ausnahmenmodell von C++ stammte primär von CLU. Es gab zu dieser Zeit aber auch anderer Sprachen, die Ausnahmebehandlung unterstützten, etwa Ada, Smalltalk (beide kannten Ausnahmen, aber keine Deklaration des Ausnahmeverhaltens) und Modula-3 (sowohl Ausnahmen als auch Deklaration des Ausnahmeverhaltens).

[131] Liskov und Snyder dokumentierten in ihrer wegweisenden Veröffentlichung[7] zu diesem Thema, ein wesentlicher Fehler von Sprachen wie C, welche Fehler in an transient fashion berichten, bestehe darin, daß

> „... every invocation must be followed by a conditional test to determine what the outcome was. This requirement leads to programs that are difficult to read, and probably inefficient as well, thus discouraging programmers from signaling and handling exceptions.“
>
> Übersetzt etwa: „... jedem (Funktions)aufruf die Prüfung des Rückgabewertes folgen muß, um das Ergebnis festzustellen. Diese Voraussetzung führt zu schwer lesbaren und wahrscheinlich auch ineffizienten Programmen und schrecken die Programmierer davon ab, Fehler zu melden und zu behandeln.“

[132] Eines der ursprünglichen Ziele der Ausnahmebehandlung war daher, diese Voraussetzung zu vermeiden. Die geprüften Ausnahmen von Java führen aber im allgemeinen zu genau dieser Sorte Quelltext. Liskov und Snyder fahren fort:

> „... requiring that the text of a handler be attached to the invocation that raises the exception would lead to unreadable programs in which expressions were broken up with handlers.“
>
> Übersetzt etwa: „... zu fordern, daß die Anweisungen eines Ausnahmebehandlers an die Stelle angehängt werden, an der die Ausnahme auftritt, würde zu unlesbaren Programmen führen, deren Anweisungen durch Ausnahmebehandler unterbrochen sind.“

[133] Stroustrup gab an, man sei beim Design der Ausnahmen von C++ dem Ansatz von CLU gefolgt, um das Ausmaß der Anweisungen zur Fehlerbehandlung zu verringern. Vermutlich hatte er beobachtet, daß die Programmierer in C typischerweise keine Fehlerbehandlung anlegten, weil Umfang und Platzierung dieser Anweisungen abschreckend und störend wirkten. Stattdessen waren es die Programmierer gewohnt, Fehler im Quelltext nicht zu beachten und Probleme mit Hilfe des Debuggers zu lösen. Diese C-Programmierer mußten, um Ausnahmen anzuwenden, erst überzeugt werden, „zusätzliche“ Anweisungen anzulegen, die sie üblicherweise nicht gebrauchten. Der Umfang an „zusätzlichen“ Anweisungen durfte also nicht lästig sein, um die Programmierer hinsichtlich

[7] Barbara Liskov and Alan Snyder: *Exception Handling in CLU*, In: *IEEE Transactions on Software Engineering*, Vol. 5, Issue 6, November 1979, pp. 546-558.

Diese Publikation ist nicht im Internet verfügbar, sondern nur im Druck. Kontaktieren Sie eine Bibliothek, um ein Exemplar zu erhalten.

der Fehlerbehandlung auf eine bessere Fährte zu setzen. Ich halte es für wichtig, dieses Ziel im Gedächtnis zu haben, wenn Sie die Auswirkungen der geprüften Ausnahmen bei Java betrachten.

[134] C++ hat noch eine weitere Idee von CLU übernommen: Die Deklaration des Ausnahmeverhaltens, um programmatisch in der Methodensignatur zu dokumentieren, das eine Methode Ausnahmen auswerfen kann. Die Deklaration des Ausnahmeverhaltens hat eigentlich zwei Aufgaben. Sie sagt erstens aus, daß die Ausnahme aus der Implementierung der Methode selbst stammt und behandelt werden muß. Die zweite Aussage lautet, daß die Methode eine Ausnahme ignoriert, die beim Verarbeiten der Anweisungen im Methodenkörper ausgeworfen werden kann und behandelt werden muß. Wir haben uns beim Betrachten der Mechanik und Syntax der Ausnahmen auf die Behandlung der Ausnahme konzentriert. Ich möchte an dieser Stelle betonen, daß Ausnahmen häufig ignoriert werden und dies eine Aussage der Deklaration des Ausnahmeverhaltens ist.

[135] Die Deklaration des Ausnahmeverhaltens ist bei C++ kein Teil der Signatur einer Methode. Die Typprüfung zur Übersetzungszeit beschränkt sich darauf, zu gewährleisten, daß die Deklaration des Ausnahmeverhaltens konsistent angewendet wird. Wirft beispielsweise eine Methode Ausnahmen aus, so müssen auch die überladenen beziehungsweise abgeleiteten Versionen dieser Methode diese Ausnahmen deklarieren. Im Gegensatz zu Java findet zur Übersetzungszeit keine Prüfung statt, ob die Methode die Ausnahme tatsächlich auswirft, oder ob die Deklaration des Ausnahmeverhaltens vollständig ist (das heißt ob alle potentiell ausgeworfenen Ausnahmen erfaßt wurden). Diese Prüfung findet zur Laufzeit statt. Wirft eine Methode eine nicht deklarierte Ausnahme aus, so ruft das C++ Programm die Funktion `unexpected()` aus der Standardbibliothek auf.

[136] Interessanterweise enthält die Standardbibliothek von C++ aufgrund der Verwendung von Templates keine einzige Deklaration von Ausnahmeverhalten. Java beschränkt die Verwendung generischer Typen in der Deklaration des Ausnahmeverhaltens.

13.12.2 Ausblick

[137] Erstens soll darauf hingewiesen werden, daß das Konzept der geprüften Ausnahme für Java erfunden wurde (mit Sicherheit angeregt durch die Deklaration des Ausnahmeverhaltes bei C++ und die Tatsache, daß C++-Programmierer sich typischerweise nicht damit abgeben). Es handelt sich allerdings um ein Experiment, das in einer der folgenden Programmiersprachen wiederholt wurde.

[138] Zweitens scheinen geprüfte Ausnahmen bei einführenden Beispielen und kleinen Programm offenbar eine gute Spracheigenschaft zu sein. Es gibt Andeutung dahingehend, daß die subtilen Schwierigkeiten aufzutreten beginnen, wenn Programme zu wachsen anfangen. Selbstständlich stellt sich der Umfang eines Programms nicht schlagartig, sondern schleichend ein. Programmiersprachen die sich nicht für große Softwareprojekte eignen, werden für kleinere Projekte verwendet. Diese Projekten wachsen und irgendwann erkennen wir, da die Dinge von „überschaubar“ nach „kompliziert“ umgeschlagen sind. Ich möchte den Gedanken anregen, daß dieser Effekt bei übertriebener Typprüfung eintritt, speziell bei geprüften Ausnahmen.

[139] Die Größenordnung des Programms scheint ein wesentlicher Faktor zu sein. Das ist problematisch, da die meisten Diskussionen nur kleine Programme zu Diskussionszwecken heranziehen. Einer der Designer von C# hat beobachtet, daß

> „Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result-decreased productivity and little or no increase in code quality.“

Übersetzt etwa: „Die Betrachtung kleiner Programme führt zur Schlußfolgerung, daß die Forderung nach der Deklaration des Ausnahmeverhaltens sowohl die Produktivität des Entwicklers als auch die Qualität des Quelltextes verbessern kann. Die Erfahrung mit großen Softwareprojekte deutet dagegen an, daß die Produktivität nachläßt und die Qualität des Quelltextes nur geringfügig oder überhaupt nicht verbessert wird.“

[140] In Bezug auf nicht abgefangene Ausnahmen vertreten die CLU-Schöpfer die Auffassung:

„We felt it was unrealistic to require the programmer to provide handlers in situations where no meaningful action can be taken.“[8]

Übersetzt etwa: „Wir hielten es für wirklichkeitsfremd, den Programmierer das Anlegen von Ausnahmebehandlern in Situationen abzuverlangen, in denen sich keine sinnvolle Funktionalität implementieren läßt. “

[141–142] Stroustrup erklärt, warum die Definition einer Methode ohne Deklaration des Ausnahmeverhaltens bedeutet, daß die Methode *beliebige* Ausnahmen statt *keiner* Ausnahmen auswerfen kann:

„However, that would require exception specifications for essentially every function, would be a significant cause for recompilation, and would inhibit cooperation with software written in other languages. This would encourage programmers to subvert the exception-handling mechanisms and to write spurious code to suppress exceptions. It would provide a false sense of security to people who failed to notice the exception.“[9]

In der Übersetzung der 3., aktualisierten und erweiterten Auflage von Nicolai Josuttis und Achim Lörke (Seite 401): „Dies würde allerdings Ausnahme-Spezifikationen[10] für quasi jede Funktion erfordern, wäre eine erhebliche Ursache für Neuübersetzungen und würde die Zusammenarbeit mit in anderen Sprachen geschriebener Software unmöglich machen. Dies würde Programmierer ermutigen, den Ausnahmebehandlungsmechanismus zu unterlaufen und «Mogel»-Code zur Unterdrückung von Ausnahmen zu schreiben. Es würde eine falsche Vorstellung von Sicherheit bei Leuten hervorrufen, die das Unterlaufen des Mechanismus nicht bemerk haben.“

Wir sehen eben dieses Verhalten (das Unterlaufen von Ausnahmen) bei den geprüften Ausnahmen in Java.

[143] Martin Fowler, der Autor von *UML Distilled*, *Refactoring* und *Analysis Patterns*, hat mit einmal geschrieben:

„... on the whole I think that exceptions are good, but Java checked exceptions are more trouble than they are worth.“

Übersetzt etwa: „... im Großen und Ganzen halte ich Ausnahmen für gut, aber die geprüften Ausnahmen von Java verursachen mehr Schwierigkeiten als sie wert sind.“

[144] Meiner Ansicht nach besteht der wesentliche Schritt bei Java in der Vereinheitlichung des Fehlermeldungsmodells, so daß nun alle Fehler mit Hilfe von Ausnahmen gemeldet werden. Aufgrund der Rückwärtskompatibilität mit dem alten C-Modell, in dem Fehler einfach ignoriert werden konnten, gilt diese Einheitlichkeit nicht für C++. Steht aber ein konsistentes Fehlermeldungsmodell mit Ausnahmen zur Verfügung, so können Ausnahmen auf Wunsch verwendet oder bis zum höchsten

[8] Barbara Liskov and Alan Snyder: *Exception Handling in CLU*, In: *IEEE Transactions on Software Engineering*, Vol. 5, Issue 6, November 1979, pp. 546-558.

[9] Bjarne Stroustrup: *The C++ Programming Language*, 3rd Edition, Addison-Wesley (1997), p. 376.

[10] Anmerkung des Übersetzers: Nicolai Josuttis und Achim Lörke nennen die Deklaration des Ausnahmeverhaltens in der deutschen Übersetzung „Ausnahme-Spezifikation“.

übergeordneten Kontext (der Konsole beziehungsweise dem aufrufenden Programm) weitergegeben werden. Als das Modell von C++ für Java modifiziert und Ausnahmen die einzige Möglichkeit zum Melden eines Fehlers wurden, ~~the extra enforcement of checked exceptions may have become less necessary~~.

[145] Ich war früher sehr davon überzeugt, daß geprüfte Ausnahmen und statische Typprüfung für die Entwicklung robuster Programme wesentlich sind. Sowohl berichtete als auch eigene Erfahrung[11] mit Sprachen, deren dynamischer Anteil größer ist als ihr statischer, veranlaßt mich, heute zu denken, daß die wesentlichen Vorteile eigentlich gegeben sind durch:

- Ein einheitliches Fehlermeldungsmodell mit Ausnahmen, unabhängig davon ob der Compiler vom Programmierer die Behandlung der Ausnahmen verlangt.
- Typprüfung, unabhängig davon wann sie stattfindet, das heißt solange die korrekte Verwendung der Typen erzwungen wird, kommt es nicht darauf an, ob die Prüfung zur Übersetzungs- oder zur Laufzeit stattfindet.

[146] Das Verringern der Beschränkungen zur Übersetzungszeit hat außerdem erhebliche Produktivitätsvorteile für den Programmierer. Der Reflexionsmechanismus und die generischen Typen sind notwendig, um das übermäßig einschränkende Wesen der statischen Typprüfung auszugleichen, wie Sie an einer Reihe von Beispielen in diesem Buch studieren können.

[147] Man hat mir bereits gesagt, daß ich mit den hier geäußerten Ansichten, Gotteslästerung begehe, meinen Ruf vernichte, den Untergang der Zivilisation herbeiführe und einen beträchtlichen Anteil der Softwareprojekte zum Scheitern verurteile. Der Glaube ist stark, daß der Compiler Ihr Projekt durch die Anzeige von Fehlern zur Übersetzungszeit retten kann, aber es ist noch wichtiger, die Grenzen dessen zu erkennen, was der Compiler leisten kann. Im Anhang von *http://www.mindview.net/Books/BetterJava* betone ich den Wert automatischer Übersetzungsprozesse und Modultests, wodurch Sie viel mehr Einfluß nehmen können, als wenn Sie versuchen, alles in einen Syntaxfehler umzumünzen. Denken Sie an das folgende Zitat:

> „A good programming language is one that helps programmers write good programs. No programming language will prevent its users from writing bad programs.“[12]
>
> Übersetzt etwa: „Eine gute Programmiersprache unterstützt Programmierer dabei, gute Programme zu schreiben. Keine Programmiersprache hindert ihre Benutzer daran, schlechte Programme zu schreiben.“

[148] Die Aussicht, daß die geprüften Ausnahmen aus Java entfernt werden, ist jedenfalls düster. Es wäre eine zu radikale Änderung der Sprache und die Gegner solcher Änderungen bei Sun Microsystems haben Gewicht. Absolute Rückwärtskompatibilität ist Teil der Geschichte und Unternehmensrichtlinie von Sun Microsystems. Nahezu jede Software von Sun Microsystems läuft auf jeder Hardware von Sun Microsystems, gleichgültig wie alt sie ist (um Ihnen ein Gefühl für diesen Grundsatz zu vermitteln). Wenn Ihnen aber eine Ausnahme im Weg steht oder Sie gezwungen sind, eine Ausnahme abzufangen, aber nicht wissen, wie Sie sie behandeln sollen, gibt es einige Alternativen.

13.12.3 Ausgabe von Ausnahmen über die Konsole

[149] Bei einfachen Programmen, wie bei vielen Beispielen in diesem Buch, besteht die einfachste Lösung zum Erhalt einer Ausnahme ohne viel Quelltext darin, die Ausnahme über die `main()`-

[11] Berichtete Erfahrung mit Smalltalk durch Austausch mit vielen erfahrenen Programmierern in dieser Sprache. Eigene Erfahrung mit Python (*http://www.python.com*).

[12] Kees Koster, Designer der Sprache CDL, zitiert von Bertrand Meyer, dem Designer der Sprache Eiffel.

Methode auf der Konsole auszugeben. Wenn Sie beispielsweise eine Datei zum Lesen öffnen möchten (mehr darüber in Kapitel 19), müssen Sie einen Eingabestrom in Gestalt eines Objektes der Klasse `FileInputStream` öffnen und schließen, wobei Ausnahmen ausgeworfen werden können. Für einfache Programme bietet sich die folgende Vorgehensweise an (Sie finden diesen „Ansatz“ an vielen Stellen in diesem Buch):

```
//: exceptions/MainException.java
import java.io.*;

public class MainException {
    // Pass all exceptions to the console:
    public static void main(String[] args) throws Exception {
        // Open the file:
        FileInputStream file =
            new FileInputStream("MainException.java");
        // Use the file ...
        // Close the file:
        file.close();
    }
} ///:~
```

[150] Beachten Sie, daß auch die `main()`-Methode Ausnahmeverhalten deklarieren kann und in diesem Beispiel der Ausnahmetyp `Exception` gewählt wurde, die Wurzelklasse aller geprüften Ausnahmen. Durch die Weitergabe der Ausnahme an die Konsole, brauchen Sie in der `main()`-Methode keine `try`/`catch`-Klausel anzulegen. (Leider ist die Dateiein-/ausgabe deutlich komplizierter, als dieses Beispiel erahnen läßt. Zügeln Sie also Ihre Begeisterung, bis Sie Kapitel 19 gelesen haben.)

Übungsaufgabe 26: (1) Ändern Sie den Dateinamen im Beispiel *MainException.java* in den Namen einer nicht vorhandenen Datei, rufen Sie das Programm auf und beachten Sie was geschieht. ■

13.12.4 Verpacken einer geprüften in einer ungeprüfte Ausnahme

[151] Das Auswerfen einer Ausnahme aus der `main()`-Methode ist bei kleinen Programmen für Ihren privaten Gebrauch komfortabel, aber nicht im allgemeinen nützlich. Das eigentliche Problem tritt ein, wenn Sie einen gewöhnlichen Methodenkörper implementieren, dabei eine andere Methode aufrufen und erkennen, daß Sie nicht wissen, wie Sie eine von dieser Methode ausgeworfene Ausnahme behandeln sollen, die Ausnahme aber auch nicht „verschlucken“ oder eine banale Meldung ausgeben wollen. With chained exceptions, a new and simple solution prevents itself. Sie können eine geprüfte Ausnahme einfach in einer Ausnahme vom Typ `RuntimeException` „verpacken“, indem Sie sie dem Konstrukor der Klasse `RuntimeException` übergeben:

```
try {
    // ... to do something useful
} catch (IDontKnowWhatToDoWithThisCheckedException e) {
    throw new RuntimeException(e);
}
```

[152] Dies scheint die ideale Lösung zu sein, um eine geprüfte Ausnahme „abzuschalten“: die Ausnahme wird nicht „verschluckt“, muß nicht als Ausnahmeverhalten der Methode deklariert werden und konserviert durch die Verkettung der Ausnahmen auch die Informationen über die ursprünliche Ausnahme.

[153] Das Verpacken einer geprüften in einer ungeprüften Ausnahme ermöglicht, die Ausnahme zu ignorieren und den gesamten Aufrufstapel durchlaufen zu lassen, ohne daß eine `try`/`catch`-Kombination und/oder eine Deklaration des Ausnahmeverhaltens notwendig ist. Dennoch können

Sie die verpackte Ausnahme mit Hilfe der `getCause()`-Methode abfangen und behandeln:

```
//: exceptions/TurnOffChecking.java
// ''Turning off'' Checked exceptions.
import java.io.*;
import static net.mindview.util.Print.*;

class WrapCheckedException {
    void throwRuntimeException(int type) {
        try {
            switch(type) {
            case 0: throw new FileNotFoundException();
            case 1: throw new IOException();
            case 2: throw new RuntimeException(''Where am I?'');
            default: return;
            }
        } catch(Exception e) { // Adapt to unchecked:
            throw new RuntimeException(e);
        }
    }
}

class SomeOtherException extends Exception {}

public class TurnOffChecking {
    public static void main(String[] args) {
        WrapCheckedException wce = new WrapCheckedException();
        // You can call throwRuntimeException() without a try
        // block, and let RuntimeExceptions leave the method:
        wce.throwRuntimeException(3);
        // Or you can choose to catch exceptions:
        for(int i = 0; i < 4; i++)
            try {
                if(i < 3)
                    wce.throwRuntimeException(i);
                else
                    throw new SomeOtherException();
            } catch(SomeOtherException e) {
                print(''SomeOtherException: '' + e);
            } catch(RuntimeException re) {
                try {
                    throw re.getCause();
                } catch(FileNotFoundException e) {
                    print(''FileNotFoundException: '' + e);
                } catch(IOException e) {
                    print(''IOException: '' + e);
                } catch(Throwable e) {
                    print(''Throwable: '' + e);
                }
            }
    }
} /* Output:
      FileNotFoundException: java.io.FileNotFoundException
      IOException: java.io.IOException
      Throwable: java.lang.RuntimeException: Where am I?
      SomeOtherException: SomeOtherException
   *///:~
```

[154] Die Methode `throwRuntimeException()` der Klasse `WrapCheckedException` erzeugt Ausnahmeobjekte verschiedener Typen. Diese Ausnahmen werden abgefangen und in Ausnahmeobjekten vom Typ `RuntimeException` verpackt, sind also jeweils die „Ursache“.

[155] Sie sehen in der Klasse `TurnOffChecking`, daß die `throwRuntimeException()`-Methode ohne `try`-Klausel aufgerufen werden kann, da sie keine geprüften Ausnahmen auswirft. Wenn Sie es wünschen, besteht die Möglichkeit jede dieser Ausnahmen abzufangen, indem Sie den Methodenaufruf in eine `try`/`catch`-Kombination einsetzen. Sie beginnen nun, alle Ausnahmen abzufangen, über deren Auftreten in der `try`-Klausel Sie explizit Bescheid wissen, hier steht der Ausnahmetyp `SomeOtherException` zuerst. Zuletzt legen Sie eine `catch`-Klausel für den Typ `RuntimeException` an und werfen den Rückgabewert der `getCause()`-Methode (die verpackte Ausnahme) erneut aus. Dadurch wird die ursprüngliche Ausnahme extrahiert, die anschließend in einer eigenen `catch`-Klausel behandelt werden kann.

[156] Das Verpacken einer geprüften Ausnahme in einer Ausnahme vom Typ `RuntimeException` wird in den restlichen Kapiteln dieses Buches angewendet, wo angemessen. Sie können auch eine eigene Ausnahmeklasse von `RuntimeException` ableiten. Eine solche Ausnahme muß nicht abgefangen werden, obwohl das Abfangen bei Bedarf möglich ist.

Übungsaufgabe 27: (1) Ändern Sie Übungsaufgabe 3 (Seite 357), so daß die dortige Ausnahme von Typ `ArrayIndexOutOfBoundsException` in einer Ausnahme vom Typ `RuntimeException` verpackt wird. ■

Übungsaufgabe 28: (1) Ändern Sie Übungsaufgabe 4 (Seite 357), so daß die dortige selbstgeschriebene Ausnahmeklasse von `RuntimeException` abgeleitet wird und zeigen Sie, daß der Compiler Ihnen nun erlaubt, die `try`-Klausel fortzulassen. ■

Übungsaufgabe 29: (1) Ändern Sie alle Ausnahmetypen im Beispiel *Stormylnning.java* (Seite 377f), so daß sie von der Klasse `RuntimeException` abgeleitet werden und zeigen Sie, daß weder eine Deklaration des Ausnahmeverhaltens noch eine `try`-Klausel notwendig ist. Löschen Sie die Kommentarzeichen `//!` und zeigen Sie, daß sich die Methoden ohne Deklaration des Ausnahmeverhaltens übersetzen lassen. ■

Übungsaufgabe 30: (2) Ändern Sie das Beispiel *Human.java* (Seite 384), so daß alle Ausnahmen von der Klasse `RuntimeException` abgeleitet werden. Ändern Sie die `main()`-Methode, um die verschiedenen Ausnahmetypen mit dem Ansatz des Beispiels *TurnOffChecking.java* (Seite 392) zu behandeln. ■

13.13 Anwendungsrichtlinien für Ausnahmen

[157]

- Behandeln Sie Probleme in einem geeigneten Kontext. (Vermeiden Sie das Abfangen von Ausnahmen, bis Sie wissen, wie Sie sie behandeln können.)
- Beheben Sie das Problem und rufen Sie die Methode, die die Ausnahme verursacht hat, noch einmal auf.
- Beheben Sie das Problem, rufen aber die Methode, die die Ausnahme verursacht hat, *nicht* noch einmal auf.
- Liefern Sie statt des erwarteten Ergebnisses der Methode ein alternatives Ergebnis.

- Tun Sie im aktuellen Kontext was Sie können und werfen Sie *dieselbe* Ausnahme nochmals in einem übergeordneten Kontext aus.
- Tun Sie im aktuellen Kontext was Sie können und werfen Sie *eine andere* Ausnahme in einem übergeordneten Kontext aus.
- Benden Sie das Programm.
- Vereinfachen Sie Ihr Ausnahmeschema. (Wenn Ihr Ausnahmeschema die Dinge verkompliziert, ist es mühsam und ärgerlich es zu benutzen.)
- Machen Sie Ihre Bibliothek und Ihr Programm sicherer. (Dies ist eine kurzfristige Investition bei der Fehlersuche und eine langfristige in die Robustheit des Programms).

13.14 Zusammenfassung

[158] Ausnahmen sind ein wesentlicher Bestandteil der Java-Programmierung. Der Umfang dessen, was Sie erreichen können, ohne zu wissen, wie Sie mit Ausnahmen umgehen müssen, ist begrenzt. Aus diesem Grund werden Ausnahmen an dieser Stelle des Buches vorgestellt. Es gibt viele Bibliotheken, mit denen Sie ohne Ausnahmebehandlung nicht arbeiten können, beispielsweise die bereits erwähnte Ein-/Ausgabebibliothek (siehe Kapitel 19).

[159] Ein Vorteil der Ausnahmebehandlung besteht darin, daß Sie sich an einer Stelle auf das zu lösende Problem und anschließend an einer anderen Stelle auf die dabei möglichen Fehler konzentieren können. Obwohl Ausnahmen im allgemeinen als ein Mittel erklärt werden, mit dem Sie Fehler zur Laufzeit melden *und* gegebenenfalls einen funktionsfähigen Programmzustand wiederherstellen können, frage ich mich seit geraumer Zeit, wie häufig der Wiederherstellungsaspekt implementiert wird oder überhaupt möglich ist. Meiner Wahrnehmung nach sind es weniger als zehn Prozent der Fälle und auch dann wird wahrscheinlich nur der Aufrufstapel bis zu einem bekannten stabilen Zustand abgewickelt und kein echtes Wiederaufnahmeverhalten implementiert. Ob diese Schätzung nun zutrifft oder nicht; ich bin zu der Überzeugung gekommen, daß der entscheidende Wert der Ausnahmen in ihrer Meldefunktion liegt. Daß Java effektiv darauf besteht, alle Fehler in Form von Ausnahmen zu melden ist ein großer Vorteil gegenüber Sprachen wie C++, bei denen Sie Fehler auf mehrere unterschiedliche Arten oder auch überhaupt nicht melden können. Ein konsistentes Fehlermeldungssystem bedeutet, daß Sie sich nicht länger bei jedem geschriebenen Stück Quelltext fragen müssen, ob Ihnen eine Fehlermeldung „durch die Lappen geht“ (vorausgesetzt, daß Sie keine Ausnahmen „verschlucken“).

[160] Sie werden in den folgenden Kapitel erkennen, daß Sie, indem Sie diese Frage beiseite schieben können, Ihre Anstrengungen im Design und der Implementierung auf interessante und anspruchsvollere Dinge konzentrieren können.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Teil IV

Intermediäre Konzepte und Bibliotheken

Kapitel 14

Die Klasse String

Inhaltsübersicht

[0] Die Verarbeitung von Zeichenketten dürfte eine der häufigsten Programmieraufgaben sein.

[1] Dies gilt in besonderem Maße für Internetanwendungen, ein Bereich in dem Java stark vertreten ist. In diesem Kapitel betrachten wir die sicherlich am häufigsten verwendete Java-Klasse (`String`) sowie die dazugehörigen Klassen und Hilfsmittel im Detail.

14.1 String-Objekte sind nicht modifierbar

[2] Die Objekte der Klasse `String` sind nicht modifizierbar. Wenn Sie in der API-Dokumentation der Klasse `String` nachlesen, sehen Sie, daß jede Methode, die scheinbar ein `String`-Objekt modifiziert, in Wirklichkeit ein neues `String`-Objekt erzeugt und zurückgibt, welches die veränderte Zeichenkette beinhaltet. Das ursprüngliche `String`-Objekt wird nicht berührt.

[3] Sehen Sie sich das folgende Beispiel an:

```
//: strings/Immutable.java
import static net.mindview.util.Print.*;

public class Immutable {
    public static String upcase(String s) {
        return s.toUpperCase();
    }
    public static void main(String[] args) {
        String q = ''howdy'';
        print(q); // howdy
        String qq = upcase(q);
        print(qq); // HOWDY
        print(q); // howdy
    }
} /* Output:
      howdy
      HOWDY
      howdy
   *///:~
```

Bei der Übergabe der Referenzvariablen `q` an die `upcase()`-Methode wird eigentlich eine Kopie des Wertes von `q` übergeben. Das Objekt, auf welches diese Referenz verweist, verbleibt an seinem physikalischen Ort. Nur die Referenz wird bei der Übergabe kopiert.

[4] Sie erkennen an der Definition der `upcase()`-Methode, daß die übergebene Referenz in der lokalen Variablen `s` gespeichert wird und nur für die Dauer der Verarbeitung des Methodenkörpers existiert. Die lokale Referenzvariable `s` verschwindet, wenn die Ausführung der Methode beendet ist. Das von `upcase()` gelieferte Ergebnis ist die ursprüngliche Zeichenkette, wobei alle Buchstaben in Großbuchstaben umgewandelt sind. Die Methode gibt selbstverständlich eigentlich die Referenz auf das Ergebnis zurück. Es zeigt sich aber, daß die zurückgegebene Referenz zu einem neuen Objekt gehört, während das ursprüngliche, von `q` referenzierte Objekt nicht verändert wurde.

[5] Dieses Verhalten ist im allgemeinen auch erwünscht. Stellen Sie sich zum Beispiel die folgende Situation vor:

```
String s = ''asdf'';
String x = Immutable.upcase(s);
```

Möchten Sie tatsächlich, daß die `upcase()`-Methode ihr Argument *verändert*? Der Leser eines Quelltextes betrachtet ein Argument in der Regel als eine Information, die der Methode zur Verfügung gestellt wird, nicht aber als Größe, die modifiziert werden soll. Dies ist eine wichtige Garantie, mit der ein Quelltext leichter zu lesen, zu schreiben und zu verstehen ist.

14.2 Vergleich des Konkatenationsoperators mit der Klasse StringBuilder

[6] Da `String`-Objekte nicht modifizierbar sind, können Sie beliebig viele Referenzvariablen auf ein solches Objekt verweisen lassen. Da ein `String`-Objekt nur abgefragt werden kann, besteht keine Gelegenheit, daß über eine Referenzvariable eine Änderung vorgenommen wird, welche die übrigen Referenzvariablen betrifft.

[7–8] Die Unveränderlichkeit kann Auswirkungen auf die Effizienz haben. Der zur Verknüpfung („Konkatenation“) von `String`-Objekten überladene `+`-Operator ist ein typisches Beispiel. Operatorüberladung bedeutet, daß die Funktionsweise eines Operators vom Typ seiner Operanden abhängt. Die beiden Operatoren `+` und `+=` (auch bei `String`-Objekten anwendbar) sind die einzigen überladenen Operatoren bei Java. Darüber hinaus gestattet Java dem Programmierer *nicht*, andere Operatoren zu überladen.[1] Das folgende Beispiel zeigt die Verknüpfung von `String`-Objekten per `+`-Operator:

```
//: strings/Concatenation.java
public class Concatenation {
    public static void main(String[] args) {
        String mango = ''mango'';
        String s = ''abc'' + mango + ''def'' + 47;
        System.out.println(s);
    }
} /* Output:
      abcmangodef47
   *///:~
```

Wir stellen nun eine *Vermutung* an, wie dieses Programm funktionieren *könnte*. Das `String`-Objekt mit dem Inhalt „abc“ könnte eine `append()`-Methode haben, die ein neues `String`-Objekt erzeugt, welches die Zeichenkette „abc“ sowie den Inhalt des `mango`-Feldes repräsentiert. Das neue `String`-Objekt würde anschließend wieder ein neues `String`-Objekt erzeugen, um die Zeichenkette „def“ anzuhängen und so weiter.

[9] Diese Vorgehensweise würde sicherlich funktionieren, setzt aber das Erzeugen einer Reihe von `String`-Objekten voraus, um das `String`-Objekt mit dem Endergebnis zu erzeugen. Danach bleiben eine Anzahl von `String`-„Zwischenobjekten“ zurück, die der automatischen Speicherbereinigung übergeben werden müssen. Ich habe den Verdacht, daß die Java-Designer diesen Ansatz zuerst ausprobiert haben. (Eine Lektion zum Thema „Softwaredesign“: Sie wissen in Wirklichkeit nichts über ein System, bevor Sie es programmiert und funktionstüchtig gemacht haben.) Ich habe auch den Verdacht, daß sich die Performanz dieser Lösung als nicht akzeptabel erwiesen hat.

[10] Der Decompiler `javap` ist ein Teil des Java Development Kits und ermöglicht Ihnen, zu sehen, was tatsächlich geschieht. Der Aufruf des Decompilers lautet:

```
javap -c Concatenation
```

Der Schalter `-c` liefert den von der Laufzeitumgebung interpretierten Bytecode. Nach dem Löschen der für uns uninteressanten Teile und einigen Änderungen an der Formatierung, lautet der relevante

[1] C++ gestattet dem Programmierer Operatoren beliebig zu überladen. Da die Operatorüberladung ein komplizierter Vorgang ist (siehe Kapitel 10 in der zweiten Auflage von *Thinking in C++*, Prentice Hall (2000)), ächteten die Designer von Java diese Fähigkeit als eine „schlechte“ Eigenschaft, die nicht in den Sprachumfang von Java aufgenommen werden sollte. Die Operatorüberladung war aber offenbar nicht schlecht genug, um nicht doch Eingang in Java zu finden. Ironischerweise wäre die Operatorüberladung in Java erheblich leichter anzuwenden als bei C++, wie die beiden Sprachen Python (*http://www.python.com*) und C# zeigen, welche beide sowohl über automatische Speicherbereinigung als auch über Operatorüberladung verfügen.

Anteil des Bytecodes:

```
public static void main(java.lang.String[]);
  Code:
   0:  ldc #2;             // String mango
   2:  astore_1
   3:  new #3;             // StringBuilder
   6:  dup
   7:  invokespecial #4;   // StringBuilder.''<init>'':()
   10: ldc #5;             // String abc
   12: invokevirtual #6;   // StringBuilder.append:(String)
   15: aload_1
   16: invokevirtual #6;   // StringBuilder.append:(String)
   19: ldc #7;             // String def
   21: invokevirtual #6;   // StringBuilder.append:(String)
   24: bipush 47
   26: invokevirtual #8;   // StringBuilder.append:(I)
   29: invokevirtual #9;   // StringBuilder.toString:()
   32: astore_2
   33: getstatic #10;      // System.out:PrintStream;
   36: aload_2
   37: invokevirtual #11;  // PrintStream.println:(String)
   40: return
}
```

Wenn Sie Erfahrung in der Assembler-Programmierung haben, ist Ihnen dieser Anblick vertraut. Anweisungen wie `dup` und `invokevirtual` sind die Äquivalente der Java-Laufzeitumgebung zu Assembler-Kommandos. Falls Sie noch nie Assembler-Kommandos gesehen haben, zerbrechen Sie sich nicht den Kopf: Es kommt in diesem Beispiel darauf an, daß der Compiler die Klasse `java.lang.StringBuilder` wählt. Der Quelltext enthält keinen Hinweis auf die Klasse `StringBuilder` und dennoch hat der Compiler entschieden diese Klasse zu verwenden, da sie viel effizienter ist als die zunächst vermutete Lösung.

[11] Der Compiler erzeugt in diesem Beispiel ein `StringBuilder`-Objekt und ruft viermal die `append()`-Methode auf, um den Inhalt des von `s` referenzierten `String`-Objektes zusammenzusetzen. Schließlich ruft der Compiler die `toString()`-Methode des `StringBuilder`-Objektes auf, um das Ergebnis zurückzugeben und anschließend (per `astore_2`) eine Referenz auf das Ergebnis in der lokalen Variablen `s` zu speichern.

[12] Bevor Sie annehmen, daß Sie einfach überall `String`-Objekte verwenden und es dem Compiler überlassen können, sich um die Effizienz zu kümmern, wollen wir etwas genauer betrachten, wie der Compiler arbeitet. Das folgende Beispiel liefert auf zwei Wegen ein `String`-Objekt als Ergebnis: Einmal mit einzelnen `String`-Objekten (`implicit()`) und einmal per `StringBuilder`-Objekt (`explicit()`):

```
//: strings/WhitherStringBuilder.java
public class WhitherStringBuilder {
    public String implicit(String[] fields) {
        String result = '''';
        for(int i = 0; i < fields.length; i++)
            result += fields[i];
        return result;
    }
    public String explicit(String[] fields) {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < fields.length; i++)
```

```
            result.append(fields[i]);
        return result.toString();
    }
} ///:~
```

Die (vereinfachte) Ausgabe des Decompilers (`javap -c WhitherStringBuilder`) zeigt den Bytecode der beiden verschiedenen Methoden, zunächst `implicit()`:

```
public java.lang.String implicit(java.lang.String[]);
  Code:
   0:  ldc #2;            // String
   2:  astore_2
   3:  iconst_0
   4:  istore_3
   5:  iload_3
   6:  aload_1
   7:  arraylength
   8:  if_icmpge 38
   11: new #3;            // StringBuilder
   14: dup
   15: invokespecial #4;  // StringBuilder.”<init>”:()
   18: aload_2
   19: invokevirtual #5;  // StringBuilder.append:(String)
   22: aload_1
   23: iload_3
   24: aaload
   25: invokevirtual #5;  // StringBuilder.append:(String)
   28: invokevirtual #6;  // StringBuilder.toString:()
   31: astore_2
   32: iinc 3, 1
   35: goto 5
   38: aload_2
   39: areturn
```

Beachten Sie die Zeilen 8 und 35, die zusammen eine Schleife bilden. Zeile 8 führt eine „größer oder gleich“-Vergleichsoperation der ganzzahligen Operanden auf dem Stack durch und springt nach dem Schleifenende in Zeile 38. Zeile 35 ist eine `goto`-Anweisung zurück zum Schleifenanfang in Zeile 5. Es kommt darauf an, daß das `StringBuilder`-Objekt *innerhalb der Schleife* erzeugt wird, das heißt bei jedem Schleifendurchgang wird ein neues `StringBuilder`-Objekt erzeugt.

[13] Nun der Bytecode von `explicit()`:

```
public java.lang.String explicit(java.lang.String[]);
  Code:
   0:  new #3;            // StringBuilder
   3:  dup
   4:  invokespecial #4;  // StringBuilder.”<init>”:()
   7:  astore_2
   8:  iconst_0
   9:  istore_3
   10: iload_3
   11: aload_1
   12: arraylength
   13: if_icmpge 30
   16: aload_2
   17: aload_1
   18: iload_3
   19: aaload
```

```
    20: invokevirtual #5;  // StringBuilder.append:()
    23: pop
    24: iinc 3, 1
    27: goto 10
    30: aload_2
    31: invokevirtual #6;  // StringBuilder.toString:()
    34: areturn
}
```

Der Bytecodeabschnitt mit der Schleife ist nicht nur kürzer und einfacher, sondern die `explicit()`-Methode erzeugt auch nur ein einziges `StringBuilder`-Objekt. Das explizite Erzeugen eines `StringBuilder`-Objektes ermöglicht Ihnen außerdem, wenn Sie über entsprechende zusätzliche Informationen verfügen, eine passende Kapazität voreinzustellen, statt das `StringBuilder`-Objekt zu veranlassen, seinen Puffer immer wieder zu vergrößern.

[14] Wenn Sie also eine `toString()`-Methode schreiben und nur einfache Operationen verwenden, können Sie die Arbeit dem Compiler überlassen und davon ausgehen, daß er ein vernünftiges Ergebnis liefert. Benötigen Sie aber Schleifen zur Implementierung der `toString()`-Methode, so sollten Sie ein explizites `StringBuilder`-Objekt wählen:

```
//: strings/UsingStringBuilder.java
import java.util.*;

public class UsingStringBuilder {
    public static Random rand = new Random(47);
    public String toString() {
        StringBuilder result = new StringBuilder("[");
        for(int i = 0; i < 25; i++) {
            result.append(rand.nextInt(100));
            result.append(", ");
        }
        result.delete(result.length()-2, result.length());
        result.append("]");
        return result.toString();
    }
    public static void main(String[] args) {
        UsingStringBuilder usb = new UsingStringBuilder();
        System.out.println(usb);
    }
} /* Output:
      [58, 55, 93, 61, 61, 29, 68, 0, 22, 7, 88, 28, 51, 89, 9, 78, 98, 61,
        20, 58, 16, 40, 11, 22, 4]
   *///:~
```

Beachten Sie, daß jedes Stück des Ergebnisses mit Hilfe einer `append()`-Anweisung angefügt wird. Wenn Sie eine Abkürzung nehmen und eine Konstruktion wie `append(a+": "+c)` verwenden, greift der Compiler ein und erzeugt zusätzliche `StringBuilder`-Objekte.

[15] Wenn Sie unsicher sind, welchen Ansatz Sie wählen sollen, können Sie sich stets per `javap` rückversichern.

[16] Die Klasse `StringBuilder` hat viele Methoden, darunter `insert()`, `replace()`, `substring()` und sogar `reverse()`, aber in der Regel werden Sie nur `append()` und `toString()` brauchen. Beachten Sie die Verwendung der `delete()`-Methode im obigen Beispiel, um das letzte Komma und das letzte Leerzeichen zu entfernen, bevor die schließende eckige Klammer angefügt wird.

[17] Die Klasse `StringBuilder` wurde in Version 5 der Java Standard Edition (SE 5) eingeführt.

Davor mußte die Klasse `StringBuffer` verwendet werden, die zwar Threadsicherheit gewährleistet (siehe Kapitel 22), dadurch aber auch deutlich weniger effizient („teurer“) ist. Operationen auf Zeichenketten sollten in der SE 5/6 schneller verarbeitet werden.

Übungsaufgabe 1: (2) Untersuchen Sie die `toString()`-Methode des Beispiels *reusing/SprinklerSystem.java* (Abschnitt 8.1, Seite 192). Würde eine `toString()`-Methode mit einem explizit erzeugten `StringBuilder`-Objekt bewirken, daß der Compiler weniger `StringBuilder`-Objekte erzeugt? ■

14.3 Unbeabsichtigte Rekursion

[18–19] Die die Containerklassen der Standardbibliothek von Java, wie alle Java-Klassen, letztendlich von `Object` abgeleitet sind, verfügt jede Containerklasse über eine `toString()`-Methode. Die `toString()`-Methode ist in der Implementierung der jeweiligen Containerklasse überschrieben, so daß ihre Objekte eine Darstellung ihrer selbst in Form eines `String`-Objektes erzeugen können, die auch die gespeicherten Elemente umfaßt. Die `toString()`-Methode der Klasse `ArrayList` wählt ein gespeichertes Element nach dem anderen aus und ruft dessen `toString()`-Methode auf:

```
//: strings/ArrayListDisplay.java
import generics.coffee.*;
import java.util.*;

public class ArrayListDisplay {
    public static void main(String[] args) {
        ArrayList<Coffee> coffees = new ArrayList<Coffee>();
        for(Coffee c : new CoffeeGenerator(10))
            coffees.add(c);
        System.out.println(coffees);
    }
} /* Output:
      [Americano 0, Latte 1, Americano 2, Mocha 3, Mocha 4, Breve 5,
        Americano 6, Latte 7, Cappuccino 8, Cappuccino 9]
   *///:~
```

Falls Sie nur die Speicheradresse Ihres Objektes auszugeben brauchen, liegt es nahe in der `toString()`-Methode die Selbstreferenz `this` zu verwenden:

```
//: strings/InfiniteRecursion.java
// Accidental recursion.
// {RunByHand}
import java.util.*;

public class InfiniteRecursion {
    public String toString() {
        return '' InfiniteRecursion address: '' + this + ''\n'';
    }
    public static void main(String[] args) {
        List<InfiniteRecursion> v = new ArrayList<InfiniteRecursion>();
        for(int i = 0; i < 10; i++)
            v.add(new InfiniteRecursion());
        System.out.println(v);
    }
} ///:~
```

[20] Wenn Sie ein Objekt der Klasse `InfiniteRecursion` erzeugen und auf der Konsole ausgeben, bekommen Sie eine sehr lange Liste von Ausnahmen. Dies gilt auch, wenn Sie die `InfiniteRecursion`-Objekte in einen Container vom Typ `ArrayList` einsetzen und diesen ausgeben. Ursache ist

die automatische Typumwandlung bei Anwesenheit eines `String`-Objektes:

```
''InfiniteRecursion address: '' + this
```

Der Compiler erkennt ein `String`-Objekt, gefolgt von einem `+`-Operator, dessen zweiter Operand (`this`) nicht der Klasse `String` angehört und versucht `this` in ein `String`-Objekt umzuwandeln. Diese Umwandlung geschieht durch Aufrufen der `toString()`-Methode, so daß sich ein rekursiver Methodenaufruf ergibt.

[21] Wenn Sie tatsächlich die Speicheradresse des Objektes brauchen, können Sie die `Object`-Version der `toString()`-Methode aufrufen: `super.toString()`.

Übungsaufgabe 2: (1) Reparieren Sie das Beispiel *InfiniteRecursion.java*. ■

14.4 Wichtige Methoden der Klasse String

[22–24] Die folgende Liste zeigt die wichtigsten Methoden der Klasse `String`. Bei Überladung werden die verschiedenen Versionen beschrieben:

- Der Konstruktor der Klasse `String` erzeugt `String`-Objekte und ist überladen. Es gibt einen Standardkonstruktor und Versionen für Argumente der Typen `String`, `StringBuilder` und `StringBuffer` sowie für `char`- und `byte`-Arrays.
- `length()` gibt die Anzahl der Zeichen des `String`-Objektes zurück. Die Methode ist nicht überladen und erwartet keine Argumente.
- `charAt()` gibt das Zeichen mit dem angeforderten Index als `char`-Wert zurück. Die Methode ist nicht überladen und erwartet ein Argument vom Typ `int` (den Index).
- `getChars()` und `getBytes()` kopiert Zeichen beziehungsweise Bytes in ein externes Array. Die Methode `getChars()` ist nicht überladen und erwartet die Indizes des ersten und des letzten Zeichens des zu kopierenden Bereiches, das Zielarray sowie den Index im Zielarray, von dem an die kopierten Zeichen eingesetzt werden sollen. Die Methode `getBytes()` ist überladen und erwartet entweder kein Argument, ein Argument vom Typ `Charset`, ein Argument vom Typ `String` oder Anfangs- und Endindex eines Abschnitts, ein `byte`-Array (Zielarray) sowie den Versatz im Zielarray (die letzte Version ist als *deprecated* deklariert).
- `toCharArray()` gibt ein `char`-Array zurück, welches die Zeichen der in diesem `String`-Objekt enthaltenen Zeichenfolge beinhaltet. Die Methode ist nicht überladen und erwartet keine Argumente.
- `equals()` und `equalsIgnoreCase()` prüfen die Inhalte zweier `String`-Objekte auf Gleichheit. Beide Methoden sind nicht überladen und erwarten je ein Argument vom Typ `String` (das Vergleichsobjekt).
- `compareTo()` gibt einen negativen Wert, Null oder einen positiven Wert zurück, abhängig von der relativen Position des `String`-Objektes, zu dem die `compareTo()`-Methode gehört und ihrem Argument bezüglich der lexikographischen Ordnung. Groß- und Kleinbuchstaben sind *nicht* äquivalent! Die Methode ist nicht überladen und erwartet ein Argument vom Typ `String` (das Vergleichsobjekt).
- `contains()` gibt `true` zurück, wenn das `String`-Objekt das Argument enthält. Die Methode ist nicht überladen und erwartet ein Argument vom Typ *`CharSequence`* (die eventuell enthaltene Zeichenkette).

- `contentEquals()` gibt `true` zurück, wenn das `String`-Objekt exakt mit dem Argument übereinstimmt. Die Methode ist überladen und erwartet entweder ein Argument vom Typ *`CharSequence`* oder ein Argument vom Typ *`CharSequence`* (die Vergleichszeichenkette).
- `equalsIgnoreCase()` gibt `true` zurück, wenn das `String`-Objekt mit dem Argument übereinstimmt, wobei die Groß-/Kleinschreibung nicht beachtet wird. Die Methode ist nicht überladen und erwartet ein Argument vom Typ `String` (das Vergleichsobjekt).
- `regionMatches()` gibt einen `boolean`-Wert zurück, der angibt, ob der gewählte Abschnitt des `String`-Objektes zu dem die Methode gehört mit dem gewählten Abschnitt des Argumentes übereinstimmt. Die Methode erwartet den Versatz bezüglich ihres `String`-Objektes, ein zweites `String`-Objekt mit Versatz (die Vergleichszeichenkette) und die Länge des zu vergleichenden Abschnitts. Die Methode hat eine überladene Version, welche die Groß-/Kleinschreibung nicht beachtet.
- `startsWith()` gibt einen `boolean`-Wert zurück, der angibt, ob der Inhalt des `String`-Objektes mit dem Argument beginnt. Die Methode erwartet ein Argument vom Typ `String` (das Anfangsstück). Die Methode hat eine überladene Version, die zusätzlich einen Versatz erwartet.
- `endsWith()` gibt einen `boolean`-Wert zurück, der angibt, ob der Inhalt des `String`-Objektes mit dem Argument endet. Die Methode ist nicht überladen und erwartet ein Argument vom Typ `String` (das Endstück)
- `indexOf()` und `lastIndexOf()` geben -1 zurück, wenn ihr `String`-Objekt das Argument nicht enthält und andernfalls den Index bei dem das Argument beginnt. Die `lastIndexOf()`-Methode sucht vom Ende an rückwärts. Beide Methoden sind überladen und erwarten jeweils entweder ein `char`-Argument, ein `char`-Argument und einen Versatz (`int`), ein `String`-Argument oder ein `String`-Argument und einen Versatz (`int`).
- `substring()` (auch `subSequence()`) geben die Referenz auf ein neues `String`-Objekt zurück, welches den angeforderten Abschnitt der Zeichenkette enthält. Die `substring()`-Methode ist überladen und erwartet entweder den Startindex oder Start- und Endindex des angeforderten Bereiches. (Die `subSequence()`-Methode ist nicht überladen und erwartet den Start- und Endindex des angeforderten Bereiches.)
- `concat()` gibt die Referenz auf ein neues `String`-Objekt zurück, welches den Inhalt ihres `String`-Objektes gefolgt vom Inhalt ihres Argumentes enthält. Die Methode ist nicht überladen und erwartet ein Argument vom Typ `String` (die anzufügende Zeichenkette).
- `replace()` gibt die Referenz auf ein neues `String`-Objekt zurück, welches die angeforderten Ersetzungen enthält. Die Methode gibt die Referenz auf das ursprüngliche `String`-Objekt zurück, wenn keine Ersetzung erfolgt ist. Die Methode ist überladen und erwartet entweder zwei Argument vom Typ `char` (das zu ersetzende Zeichen und das Ersatzzeichen) oder zwei Argumente vom Typ *`CharSequence`* (die zu ersetzende Zeichenkette und die Ersatzzeichenkette).
- `toLowerCase()` und `toUpperCase()` geben die Referenz auf ein neues `String`-Objekt zurück, welches die Umwandlung aller Großbuchstaben in Kleinbuchstaben beziehungsweise umgekehrt enthält. Die Methode gibt die Referenz auf das ursprüngliche `String`-Objekt zurück, wenn keine Änderung erfolgt ist. Beide Methode sind überladen und erwartet entweder keine Argumente oder ein Argument vom Typ `Locale`.
- `trim()` gibt die Referenz auf ein neues `String`-Objekt zurück, welches den Inhalt des ursprünglichen `String`-Objekte ohne führenden und nachlaufenden Leerraum enthält. Die Methode gibt die Referenz auf das ursprüngliche `String`-Objekt zurück, wenn keine Leerraum entfernt wurde. Die Methode ist nicht überladen und erwartet keine Argumente.

- `valueOf()` gibt die Referenz auf ein neues `String`-Objekt zurück, welches eine Darstellung ihres Argumentes als Zeichenkette enthält. Die Methode ist überladen und erwartet entweder ein Argument primitiven Typs, ein `char`-Array, ein `char`-Array mit Versatz und der Länge des gewünschten Abschnitts oder ein `String`-Objekt.
- `intern()` liefert für jede bezüglich des Programms eindeutige Zeichenkette die Referenz auf ein entsprechendes `String`-Objekt. Die Methode ist nicht überladen und erwartet keine Argumente.

An dieser Liste können Sie beobachten, daß jede Methode der Klasse `String` ein neues `String`-Objekt zurückgibt, falls der Inhalt des ursprünglichen `String`-Objektes verändert werden muß. Beachten Sie, daß die Methode einfach die Referenz auf das ursprüngliche Objekt zurückgibt, wenn keine Änderung erforderlich ist. Das spart sowohl Speicher als auch Aufwand ein. Die `String`-Methoden mit regulären Ausdrücken werden später in diesem Kapitel behandelt.

14.5 Formatierte Ausgabe

[25] Eine langersehnte Fähigkeit, die in der SE 5 endlich realisiert wurde, ist die formatierte Ausgabe im Stil der C-Funktion `printf()`. Diese Methode vereinfacht nicht nur die Programmierung der Ausgabe, sondern gestattet dem Java-Programmierer die wirksame Steuerung von formatierter Ausgabe und Ausrichtung.[2]

14.5.1 Die C-Funktion printf()

[26] Die `printf()`-Funktion von C setzt Zeichenketten, im Gegensatz zu Java nicht zusammen, sondern erwartet eine *Formatdefinition*, in die sie die Werte nacheinander einsetzt. Statt Zeichenkettenliterale (Zeichenketten zwischen doppelten Anführungszeichen) und Variablen mit Hilfe des überladenen `+`-Operators (bei C nicht überladen) zu verknüpfen, definiert die `printf()`-Funktion mittels spezieller Platzhalter die Stellen, an denen die Daten eingesetzt werden sollen. Die in die Formatdefinition eingesetzten Argumente folgen in Form einer kommaseparierten Liste.

[27] Beispiel für die Anwendung der C-Funktion `printf()`:

```
printf('Row 1: [%d %f]\n', x, y);
```

Zur Laufzeit wird der Wert von `x` bei `%d` eingesetzt und der Wert von `y` bei `%f`. Diese Platzhalter werden als *Formatierungselemente* (*format specifier*)[3] bezeichnet und drücken zusätzlich zu der Position, an welcher der Wert eingesetzt werden soll aus, welcher Variablentyp zu verwenden und wie der eingesetzte Wert zu formatieren ist. Beispielsweise repräsentiert `%d` im obigen Beispiel einen ganzzahligen Wert, `%f` dagegen eine Fließkommazahl (`float` oder `double`).

14.5.2 Die format()-Methoden der Klassen PrintStream und PrintWriter

[28] Die SE 5 definiert in den Klassen `PrintStream` und `PrintWriter` (siehe Kapitel 19) die Methode `format()`, die somit auch bei `System.out` zur Verfügung steht. Die `format()`-Methode ist der C-

[2] Mark Welsh hat mich bei der Arbeit an diesem Abschnitt sowie an Abschnitt 14.7 unterstützt.

[3] Anmerkung des Übersetzers: Axel-Tobias Schreiner und Ernst Janich, die Übersetzer von Brian Wilson Kernighans und Dennis MacAlistair Ritchies berühmtem Buch *The C Programming Language*, haben die Bezeichnungen „Format-Zeichenkette“ und „Umwandlungsangabe“ gewählt. Das Zeichen nach dem Prozentsymbol heißt dort „Umwandlungszeichen“. (Siehe Unterabschnitt *7.2 Formatierte Ausgabe — printf*, Seite 147f.)

Funktion `printf()` nachempfunden. Für Nostalgiker gibt es sogar eine `printf()`-Methode, die lediglich `format()` aufruft. Das folgende einfache Beispiel zeigt beide Methoden in Aktion:

```
//: strings/SimpleFormat.java
public class SimpleFormat {
    public static void main(String[] args) {
        int x = 5;
        double y = 5.332542;
        // The old way:
        System.out.println("Row 1: [" + x + " " + y + "]");
        // The new way:
        System.out.format("Row 1: [%d %f]\n", x, y);
        // or
        System.out.printf("Row 1: [%d %f]\n", x, y);
    }
} /* Output:
      Row 1: [5 5.332542]
      Row 1: [5 5.332542]
      Row 1: [5 5.332542]
   *///:~
```

Die Ausgaben von `format()` und `printf()` sind äquivalent. Beide Methoden verwenden eine Formatdefinition, gefolgt von einem Argument pro Formatierungselement.

14.5.3 Die Klasse Formatter

[29] Die gesamte neue Formatierungsfunktionalität von Java ist in der Klasse `java.util.Formatter` implementiert. Sie können sich ein Objekt dieser Klasse wie einen Übersetzer vorstellen, der Ihre Formatdefinition und Ihre Argumente in das gewünschte Ergebnis umwandelt. Beim Erzeugen eines `Formatter`-Objektes teilen Sie dem Konstruktor mit, wohin dieses Ergebnis gesendet werden soll:

```
//: strings/Turtle.java
import java.io.*;
import java.util.*;

public class Turtle {
    private String name;
    private Formatter f;
    public Turtle(String name, Formatter f) {
        this.name = name;
        this.f = f;
    }
    public void move(int x, int y) {
        f.format("%s The Turtle is at (%d,%d)\n", name, x, y);
    }
    public static void main(String[] args) {
        PrintStream outAlias = System.out;
        Turtle tommy = new Turtle("Tommy", new Formatter(System.out));
        Turtle terry = new Turtle("Terry", new Formatter(outAlias));
        tommy.move(0,0);
        terry.move(4,8);
        tommy.move(3,4);
        terry.move(2,5);
        tommy.move(3,3);
        terry.move(3,3);
    }
} /* Output:
```

```
    Tommy The Turtle is at (0,0)
    Terry The Turtle is at (4,8)
    Tommy The Turtle is at (3,4)
    Terry The Turtle is at (2,5)
    Tommy The Turtle is at (3,3)
    Terry The Turtle is at (3,3)
  *///:~
```

Die Ausgabe des von `tommy` referenzierten `Turtle`-Objektes geht an `System.out`, die Ausgabe des von `terry` referenzierten `Turtle`-Objektes ebenfalls an `System.out`, aber unter einem anderen Namen. Der Konstruktor der Klasse `Formatter` ist überladen und akzeptiert viele verschiedene Ausgabeziele. Die nützlichsten sind aber `PrintStream` (wie oben), `OutputStream` und `File`. Mehr zu diesem Thema in Kapitel 19.

Übungsaufgabe 3: (1) Ändern Sie das Beispiel *Turtle.java*, so daß es alle Ausgaben an den Standardfehlerkanal (`System.err`) sendet. ■

[30] Das vorige Beispiel verwendet ein neues Formatierungselement: `%s`. Dieses Formatierungselement repräsentiert ein Argument vom Typ `String` und ist ein Beispiel für die einfachste Sorte von Formatdefinition, die lediglich aus einem Formatierungselement (ohne zusätzliche Zeichenkettenliterale) besteht.

14.5.4 Das Format der Formatierungselemente

[31] Die Kontrolle von Abstand und Ausrichtung beim Einsetzen der Daten erfordert aufwändigere Formatierungselemente. Die allgemeine Syntax lautet:

```
%[argument_index$][flags][width][.precision]conversion
```

Häufig müssen Sie für ein Feld eine Mindestbreite festsetzen. Diese Einstellung wird im Abschnitt `width` vorgenommen. Das `Formatter`-Objekt garantiert, daß das Feld wenigstens eine bestimmte Anzahl von Zeichen breit ist, indem nicht benötigte Stellen mit Leerzeichen aufgefüllt werden. Die Daten werden per Voreinstellung rechtsbündig angeschlagen. Sie können dieses Standardverhalten aber durch ein Minuszeichen im Abschnitt `flags` überschreiben.

[32] Der Abschnitt `precision` wird umgekehrt zum Abschnitt `width` interpretiert, gibt also einen Höchstwert an. Im Gegensatz zu dem bei allen Datentypen anwendbaren und einheitlich wirksamen `width`-Abschnitt, hängt die Wirkung von `precision` vom Datentyp ab. Bei `String`-Objekten gibt `precision` an, wieviele Zeichen höchstens ausgegeben werden. Bei Fließkommazahlen definiert `precision` dagegen die Anzahl der Nachkommastellen (voreingestellt sind sechs Stellen). Der Fließkommawert wird bei zu vielen Nachkommastellen gerundet und bei zu wenigen Stellen mit Nullen aufgefüllt. Der `precision`-Abschnitt ist bei ganzen Zahlen nicht verwendbar, da diese keinen gebrochenen Anteil besitzen und die Definition eines `precision`-Abschnitt bei einem ganzzahligen Formatierungselement ruft eine Ausnahme vom Typ `IllegalFormatPrecisionException` hervor.

[33] Das folgende Beispiel nutzt Formatierungselemente zur Formatierung einer Einkaufsliste:

```
//: strings/Receipt.java
import java.util.*;

public class Receipt {
    private double total = 0;
    private Formatter f = new Formatter(System.out);
    public void printTitle() {
        f.format("%-15s %5s %10s\n", "Item", "Qty", "Price");
        f.format("%-15s %5s %10s\n", "--", "--", "---");
```

```
    }
    public void print(String name, int qty, double price) {
        f.format("%-15.15s %5d %10.2f\n", name, qty, price);
        total += price;
    }
    public void printTotal() {
        f.format("%-15s %5s %10.2f\n", "Tax", "", total*0.06);
        f.format("%-15s %5s %10s\n", "", "", "---");
        f.format("%-15s %5s %10.2f\n", "Total", "",
                total * 1.06);
    }
    public static void main(String[] args) {
        Receipt receipt = new Receipt();
        receipt.printTitle();
        receipt.print("Jack's Magic Beans", 4, 4.25);
        receipt.print("Princess Peas", 3, 5.1);
        receipt.print("Three Bears Porridge", 1, 14.29);
        receipt.printTotal();
    }
} /* Output:
      Item Qty Price
      -- -- ---
      Jack's Magic Be 4 4.25
      Princess Peas 3 5.10
      Three Bears Por 1 14.29
      Tax 1.42
      ---
      Total 25.06
   *///:~
```

Die Klasse `Formatter` verbindet leistungsfähige Steuerungsmöglichkeiten für Abstand und Ausrichtung mit knapper und präziser Syntax. Die Formatdefinitionen in diesem Beispiel wurden einfach kopiert, um konsistente passende Abstände zu bekommen.

Übungsaufgabe 4: (3) Ändern Sie das Beispiel *Receipt.java*, so daß die einzelnen Feldbreiten über einen Satz von Konstanten eingestellt werden können. Das Ziel besteht darin, daß Sie die Breite eines Feldes einfach ändern können, indem Sie nur einen einzigen Wert anpassen müssen. ■

14.5.5 Die Umwandlungszeichen der Formatierungselemente

[34] Die folgenden *Umwandlungszeichen* (siehe Fußnote 3 auf Seite 406) werden Ihnen am häufigsten begegnen:

- `d`: Ganzzahliger Wert (dezimal)
- `c`: Unicodezeichen
- `b`: Boolescher Wert
- `s`: `String`-Objekt
- `f`: Fließkommazahl (dezimal)
- `e`: Fließkommazahl in wissenschaftlicher Notation
- `x`: Ganzzahliger Wert (hexadezimal)
- `h`: Hashwerte (hexadezimal)

- %: Literales Prozentzeichen (%)

[35] Das folgende Beispiel zeigt Formatierungselemente mit diesen Umwandlungszeichen in Aktion:

```
//: strings/Conversion.java
import java.math.*;
import java.util.*;

public class Conversion {
    public static void main(String[] args) {
        Formatter f = new Formatter(System.out);

        char u = 'a';
        System.out.println("u = 'a'");
        f.format("s: %s\n", u);
        // f.format("d: %d\n", u);
        f.format("c: %c\n", u);
        f.format("b: %b\n", u);
        // f.format("f: %f\n", u);
        // f.format("e: %e\n", u);
        // f.format("x: %x\n", u);
        f.format("h: %h\n", u);

        int v = 121;
        System.out.println("v = 121");
        f.format("d: %d\n", v);
        f.format("c: %c\n", v);
        f.format("b: %b\n", v);
        f.format("s: %s\n", v);
        // f.format("f: %f\n", v);
        // f.format("e: %e\n", v);
        f.format("x: %x\n", v);
        f.format("h: %h\n", v);

        BigInteger w = new BigInteger("50000000000000");
        System.out.println("w = new BigInteger(\"50000000000000\")");
        f.format("d: %d\n", w);
        // f.format("c: %c\n", w);
        f.format("b: %b\n", w);
        f.format("s: %s\n", w);
        // f.format("f: %f\n", w);
        // f.format("e: %e\n", w);
        f.format("x: %x\n", w);
        f.format("h: %h\n", w);

        double x = 179.543;
        System.out.println("x = 179.543");
        // f.format("d: %d\n", x);
        // f.format("c: %c\n", x);
        f.format("b: %b\n", x);
        f.format("s: %s\n", x);
        f.format("f: %f\n", x);
        f.format("e: %e\n", x);
        // f.format("x: %x\n", x);
        f.format("h: %h\n", x);

        Conversion y = new Conversion();
        System.out.println("y = new Conversion()");
        // f.format("d: %d\n", y);
        // f.format("c: %c\n", y);
        f.format("b: %b\n", y);
```

```
        f.format("s: %s\n", y);
        // f.format("f: %f\n", y);
        // f.format("e: %e\n", y);
        // f.format("x: %x\n", y);
        f.format("h: %h\n", y);

        boolean z = false;
        System.out.println("z = false");
        // f.format("d: %d\n", z);
        // f.format("c: %c\n", z);
        f.format("b: %b\n", z);
        f.format("s: %s\n", z);
        // f.format("f: %f\n", z);
        // f.format("e: %e\n", z);
        // f.format("x: %x\n", z);
        f.format("h: %h\n", z);
    }
} /* Output: (Sample)
      u = 'a'
      s: a
      c: a
      b: true
      h: 61
      v = 121
      d: 121
      c: y
      b: true
      s: 121
      x: 79
      h: 79
      w = new BigInteger("50000000000000")
      d: 50000000000000
      b: true
      s: 50000000000000
      x: 2d79883d2000
      h: 8842a1a7
      x = 179.543
      b: true
      s: 179.543
      f: 179.543000
      e: 1.795430e+02
      h: 1ef462c
      y = new Conversion()
      b: true
      s: Conversion@9cab16
      h: 9cab16
      z = false
      b: false
      s: false
      h: 4d5
   *///:~
```

Die auskommentierten Zeilen zeigen Umwandlungzeichen, deren Kombination mit dem entsprechenden Datentyp unzulässig ist. Die Verarbeitung einer solchen Anweisung ruft eine Ausnahme vom Typ `IllegalFormatConversionException` hervor.

[36] Beachten Sie, daß das Umwandlungszeichen `b` bei allen Datentypen funktioniert, sich aber eventuell nicht so verhält, wie Sie erwarten. Bei primitiven `boolean`-Werten oder `Boolean`-Objekten lau-

tet das Ergebnis `true` beziehungsweise `false`. Jeder andere, von `null` verschiedene Argumentwert liefert stets `true`. Selbst der numerische Wert Null, der in vielen Programmiersprachen (darunter auch C) mit `false` identifiziert wird, liefert `true`. Seien Sie also vorsichtig, wenn Sie dieses Umwandlungszeichen zusammen mit nicht-booleschen Typen verwenden.

[37] Es gibt noch andere undurchsichtige Umwandlungszeichen und Optionen für Formatierungselemente. Sie finden weiterführende Informationen dazu in der API-Dokumentation der Klasse `Formatter`.

Übungsaufgabe 5: (5) Konstruieren Sie zu jedem der grundlegenden Umwandlungszeichen in der Liste auf Seite 409 das kompliziertest mögliche Formatierungselement (siehe allgemeines Format von Formatierungselementen auf Seite 408). ■

14.5.6 Die format()-Methode der Klasse String

[38] Die SE 5 hat auch die C-Funktion `sprintf()` zum Vorbild genommen, die zum Erzeugen von Zeichenketten verwendet wird. Die statische `String`-Methode `format()` erwartet dieselben Argumente wie die gleichnamige (nicht statische) Methode der Klasse `Formatter`. Die `String`-Methode kommt gelegen, wenn Sie `format()` nur ein einziges Mal aufrufen müssen:

```
//: strings/DatabaseException.java
public class DatabaseException extends Exception {
    public DatabaseException(int transactionID, int queryID, String message) {
        super(String.format(''(t%d, q%d) %s'', transactionID, queryID, message));
    }
    public static void main(String[] args) {
        try {
            throw new DatabaseException(3, 7, ''Write failed'');
        } catch(Exception e) {
            System.out.println(e);
        }
    }
} /* Output:
      DatabaseException: (t3, q7) Write failed
   *///:~
```

Die `format()`-Methode der Klasse `String` erzeugt hinter der Kulissen ebenfalls ein `Formatter`-Objekt und reicht Ihre Argument weiter. Die Wahl der `String`-Methode ist allerdings häufig klarer und leichter, als die manuelle Lösung.

14.5.6.1 Hexadezimale Formatierung des Inhaltes binärer Dateien

[39] Ein zweites Beispiel: Gelegentlich möchten Sie die Bytes in einer binären Datei im Hexadezimalformat betrachten. Die folgende kleine Hilfsklasse zeigt ein binäres Array von Bytes mit Hilfe der `String`-Methode `format()` in lesbarer hexadezimaler Formatierung an:

```
//: net/mindview/util/Hex.java
package net.mindview.util;
import java.io.*;

public class Hex {
    public static String format(byte[] data) {
        StringBuilder result = new StringBuilder();
        int n = 0;
        for(byte b : data) {
```

```
            if(n % 16 == 0)
                result.append(String.format("%05X: ", n));
            result.append(String.format("%02X ", b));
            n++;
            if(n % 16 == 0) result.append("\n");
        }
        result.append("\n");
        return result.toString();
    }
    public static void main(String[] args) throws Exception {
        if(args.length == 0)
            // Test by displaying this class file:
            System.out.println(format(BinaryFile.read("Hex.class")));
        else
            System.out.println(format(BinaryFile.read(new File(args[0]))));
    }
} /* Output: (Sample)
      00000: CA FE BA BE[4] 00 00 00 31 00 52 0A 00 05 00 22 07
      00010: 00 23 0A 00 02 00 22 08 00 24 07 00 25 0A 00 26
      00020: 00 27 0A 00 28 00 29 0A 00 02 00 2A 08 00 2B 0A
      00030: 00 2C 00 2D 08 00 2E 0A 00 02 00 2F 09 00 30 00
      00040: 31 08 00 32 0A 00 33 00 34 0A 00 15 00 35 0A 00
      00050: 36 00 37 07 00 38 0A 00 12 00 39 0A 00 33 00 3A
      ...
   *///:~
```

Das Beispiel stützt sich auf die Hilfsklasse `BinaryFile`, die in Unterabschnitt 19.7.2 eingeführt wird. Die `read()`-Methode liefert den gesamten Inhalt der Datei als `byte`-Array.

Übungsaufgabe 6: (2) Schreiben Sie eine Klasse mit `int`-, `long`-, `float`- und `double`-Feldern sowie eine `toString()`-Methode, welche die statische `String`-Methode `format()` aufruft und zeigen Sie, daß Ihre Klasse korrekt funktioniert. ■

14.6 Reguläre Ausdrücke

[40] Reguläre Ausdrücke waren lange Zeit ein wesentlicher Bestandteil von Unix-Kommandos wie `sed` und `awk` sowie von Sprachen wie Python und Perl (manche Programmierer behaupten, reguläre Ausdrücke seien der eigentliche Grund für den Erfolg von Perl). Die Verarbeitung von Zeichenketten wurde bei Java früher an die Klassen `String`, `StringBuffer` und `StringTokenizer` delegiert, die allerdings, verglichen mit regulären Ausdrücken, nur einfache Möglichkeiten zur Verfügung stellten.

[41] Reguläre Ausdrücke sind mächtige und flexible Werkzeuge zur Verarbeitung von Zeichenketten und gestatten Ihnen, programmatisch komplexe Textmuster zu definieren, nach denen eine gegebene Zeichenkette anschließend durchsucht wird. Sie können auf Treffer beliebig reagieren. Wirkt ihre Syntax auch auf den ersten Blick einschüchternd, so repräsentieren regulärer Ausdrücke doch eine kompakte und dynamische Sprache, mit der sich sämtliche Varianten von Zeichenkettenverarbeitung, Suchen/Ersetzen, Textänderungen und Verifizierungsproblemen auf universelle Art und Weise lösen lassen.

[4] Anmerkung des Übersetzers: Siehe hierzu Fußnote 11 in Abschnitt 21.5.3 (Seite 852).

14.6.1 Grundlagen und Beispiele

[42] Ein regulärer Ausdruck beschreibt Eigenschaften einer Zeichenkette. Ein Treffer liegt vor, wenn eine Zeichenkette bestimmte kennzeichnende Eigenschaften hat. Die Notation `-?` (ein Minuszeichen, gefolgt von einem Fragezeichen) drückt beispielsweise aus, daß eine Zahl mit einem Minuszeichen beginnen kann. Eine ganze Zahl wird als Vorkommen einer oder mehrerer Ziffern beschrieben. Reguläre Ausdrücke verwenden das Symbol `\d`, um eine Ziffer zu beschreiben. Falls Sie bereits in einer anderen Programmiersprache Erfahrung mit regulären Ausdrücken gesammelt haben, wird Ihnen sofort ein Unterschied bei der Verwendung von Backslashes (\) auffallen: Bei anderen Sprachen repräsentiert \\ in einem regulären Ausdruck einen literalen Backslash ohne weitere Funktion. Bei Java stellt \\ dagegen einen einfachen Backlash dar, dem ein Zeichen mit besonderer Bedeutung folgt. Eine Ziffer wird bei Java durch `\\d` (doppelter Backslash) beschrieben, ein literaler Backslash durch \\\\ (vierfacher Backslash). Andererseits genügt bei den Symbolen für Zeilenumbruch und Tabulator ein einzelnder Backslash: `\n\t`.

[43–44] Der sogenannte *Quantor* `+` zeigt eines oder mehrere Vorkommen des Ausdrucks an, an den er sich anschließt. Der Ausdruck `-?\\d+` beschreibt demnach ein optionales Minuszeichen, gefolgt von einer oder mehreren Ziffern. Die in der Klasse `String` implementierte Funktionalität bietet die einfachste Möglichkeit zur Anwendung eines regulären Ausdrucks. Das folgende Beispiel zeigt, welche Zeichenketten zu `-?\\d+` passen:

```
//: strings/IntegerMatch.java
public class IntegerMatch {
    public static void main(String[] args) {
        System.out.println(''-1234''.matches(''-?\\d+''));
        System.out.println(''5678''.matches(''-?\\d+''));
        System.out.println(''+911''.matches(''-?\\d+''));
        System.out.println(''+911''.matches(''(-|\\+)?\\d+''));
    }
} /* Output:
      true
      true
      false
      true
   *///:~
```

Während die beiden ersten Zeichenketten passen, beginnt die dritte mit einem Pluszeichen. Die Notation ist zwar gültig, paßt aber nicht zum regulären Ausdruck `-?\\d+`. Wir brauchen also eine Möglichkeit, um ein optionales Plus- oder Minuszeichen auszudrücken. Klammern bewirken bei regulären Ausdrücken die Gruppierung eines Teilausdrucks und der vertikale Strich (`|`) bedeutet „oder“. Somit beschreibt `(-|\\+)?`, daß der folgende Teil der Zeichenkette mit einem Minus-, einem Pluszeichen oder ohne Vorzeichen (wegen des Fragezeichens) beginnen darf. Da das Pluszeichen in regulären Ausdrücken eine besondere Bedeutung hat, muß es mittels eines doppelten Backslashes (\\) geschützt werden, um als literales Pluszeichen interpretiert zu werden.

[45] Die `split()`-Methode der Klasse `String` ist ein nützliches Hilfsmittel und trennt die im `String`-Objekt enthaltene Zeichenkette an den durch einen regulären Ausdruck definierten Teilzeichenketten auf:

```
//: strings/Splitting.java
import java.util.*;

public class Splitting {
    public static String knights =
        ''Then, when you have found the shrubbery, you must '' +
        ''cut down the mightiest tree in the forest... '' +
```

```
        ''with... a herring!'';
    public static void split(String regex) {
        System.out.println(Arrays.toString(knights.split(regex)));
    }
    public static void main(String[] args) {
        split('' ''); // Doesn't have to contain regex chars
        split(''\\W+''); // Non-word characters
        split(''n\\W+''); // 'n' followed by non-word characters
    }
} /* Output:
      [Then,, when, you, have, found, the, shrubbery,, you, must, cut, down, the,
        mightiest, tree, in, the, forest..., with..., a, herring!]
      [Then, when, you, have, found, the, shrubbery, you, must, cut, down, the,
        mightiest, tree, in, the, forest, with, a, herring]
      [The, whe, you have found the shrubbery, you must cut dow,
        the mightiest tree i, the forest... with... a herring!]
   *///:~
```

Beachten Sie als erstes, daß Sie auch gewöhnliche Zeichen als reguläre Ausdrücke verwenden können. Ein regulärer Ausdruck muß nicht zwingend aus speziellen Symbolen bestehen, wie der erste Aufruf der `split()`-Methode zeigt, der ein Leerzeichen als Trennzeichen definiert.

[46–47] Der zweite und dritte Aufruf der `split()`-Methode definieren *Nicht-Wortzeichen* als Trennzeichen (`\\W` mit einem Großbuchstaben; `\\w` mit einem Kleinbuchstaben stellt ein *Wortzeichen* dar). Beachten Sie, daß die Interpunktionszeichen im zweiten Beispiel verschwunden sind. Der reguläre Ausdruck des dritten Aufrufs der `split()`-Methode beschreibt den Buchstaben „n“, gefolgt von einem oder mehreren Nicht-Wortzeichen. Beachten Sie auch, daß die Treffer des regulären Ausdrucks nicht im Ergebnis vorkommen. Die `split()`-Methode der Klasse `String` ist überladen. Die andere Version der `split()`-Methode erwartet ein Argument, das angibt, in wieviele Teile die Zeichenkette höchstens aufgetrennt werden soll.

[48–49] Die letzten beiden `String`-Methoden mit regulären Ausdrücken in diesem Unterabschnitt sind `replaceFirst()` und `replaceAll()`. Sie ersetzten entweder nur das erste oder alle Vorkommen:

```
//: strings/Replacing.java
import static net.mindview.util.Print.*;

public class Replacing {
    static String s = Splitting.knights;
    public static void main(String[] args) {
        print(s.replaceFirst(''f\\w+'', ''located''));
        print(s.replaceAll(''shrubbery|tree|herring'',''banana''));
    }
} /* Output:
      Then, when you have located the shrubbery, you must cut down the mightiest
        tree in the forest... with... a herring!
      Then, when you have found the banana, you must cut down the mightiest banana
        in the forest... with... a banana!
   *///:~
```

Der erste Ausdruck paßt zu Vorkommen des Buchstabens „f“, gefolgt von mindestens einem Wortzeichen (beachten Sie, daß „w“ diesesmal als Kleinbuchstabe geschrieben ist). Der erste Ausdruck ersetzt den ersten entdeckten Treffer, so daß das Wort „found“ durch „located“ ersetzt wird.

[50] Der zweite Ausdruck paßt zu jedem Vorkommen der drei, durch vertikale Striche getrennten Worte und ersetzt alle entdeckten Vorkommen.

[51] Wie Sie auf den folgenden Seiten lernen werden, verfügen reguläre Ausdrücke, die nicht an die

Klasse `String` gebunden sind, über noch mächtigere Ersetzungsmöglichkeiten. Sie können beispielsweise Methoden aufrufen, um Ersetzungen vorzunehmen. Nicht an die Klasse `String` gebundene reguläre Ausdrücke sind außerdem deutlich effizienter, wenn Sie einen regulären Ausdruck mehr als einmal brauchen.

Übungsaufgabe 7: (5) Konstruieren und testen Sie einen regulären Ausdruck, der einen Satz dahingehend überprüft, ob er mit einem Großbuchstaben beginnt und mit einem Punkt endet. Lesen Sie, falls erforderlich, in der API-Dokumentation der Klasse `java.util.regex.Pattern` nach. ■

Übungsaufgabe 8: (2) Trennen Sie die Zeichenkette in `Splitting.knights` (Seite 414) an den Worten „the“ oder „you“. ■

Übungsaufgabe 9: (4) Ersetzen Sie die Vokale in der Zeichenkette in `Splitting.knights` (Seite 414) durch Unterstriche. Lesen Sie, falls erforderlich, in der API-Dokumentation der Klasse `java.util.regex.Pattern` nach. ■

14.6.2 Konstruktion regulärer Ausdrücke

[52] Wir beginnen die Einführung in den Aufbau der regulären Ausdrücke mit einer Teilmenge der verfügbaren Bausteine. Die API-Dokumentation der Klasse `java.util.regex.Pattern` beinhaltet eine vollständige Liste aller Bausteine zur Konstruktion regulärer Ausdrücke. Die folgende Liste zeigt Bausteine für einzelne Zeichen:

- `B`: Der spezifische Buchstabe „B“.
- `\xhh`: Das Zeichen mit dem Hexadezimalwert `oxhh` (`h` ist ein Platzhalter für eine hexadezimale Ziffer).
- `\uhhhh`: Das Unicodezeichen mit dem Hexadezimalwert `oxhhhh`.
- `\t`: Ein Tabulator.
- `\n`: Ein Zeilenumbruch.
- `\r`: Ein Wagenrücklauf.
- `\f`: Ein Seitenvorschub.
- `\e`: Ein Fluchtsymbol.

[53] Die Macht der regulären Ausdrücke zeigt sich ansatzweise bei der Verwendung von Zeichenklassen. Die folgende Liste zeigt typische Möglichkeiten zur Definition eigener sowie einige vordefinierte Zeichenklassen:

- `.` (ein Punkt): Ein beliebiges Zeichen.
- `[abc]`: Eines der Zeichen `a`, `b` oder `c` (gleichbedeutend mit `a|b|c`).
- `[^abc]`: Ein beliebiges Zeichen, nicht aber `a`, `b` oder `c` (Invertierung von `[abc]`).
- `[a-zA-Z]`: Ein beliebiges Zeichen zwischen `a` und `z` oder zwischen `A` und `Z` (Bereich).
- `[abc[hij]]`: Eines der Zeichen `a`, `b`, `c`, `h`, `i` oder `j` (gleichbedeutend mit `a|b|c|h|i|j`, Vereinigungsmenge).
- `[a-z&&[hij]]`: Eines der Zeichen `h`, `i` oder `j` (Schnittmenge).
- `\s`: Ein Leerraumzeichen (*whitespace*), das heißt Leerzeichen, Tabulator, Zeilenumbruch, Seitenvorschub oder Wagenrücklauf.

- `\S`: Ein Nicht-Leerraumzeichen (gleichbedeutend mit `[^\s]`).
- `\d`: Eine Ziffer (gleichbedeutend mit `[0-9]`).
- `\D`: Eine Nicht-Ziffer (gleichbedeutend mit `[^0-9]`).
- `\w`: Ein Wortzeichen (gleichbedeutend mit `[a-zA-Z_0-9]`).
- `\W`: Ein Nicht-Wortzeichen (gleichbedeutend mit `[^a-zA-Z_0-9]`).

[54] Die Listen in diesem Unterabschnitt zeigen nur Beispiele. Am besten setzen Sie in Ihrem Webbrowser ein Lesezeichen für die API-Dokumentation der Klasse `java.util.regex.Pattern`, so daß Sie stets schnell auf die dortige Darstellung der regulären Ausdrücke bei Java zurückgreifen können. Die folgende Liste zeigt einige logische Operatoren (obere Hälfte) beziehungsweise *Anker* (untere Hälfte) bei regulären Ausdrücken:

- `XY`: Buchstabe „X“, gefolgt vom Buchstaben „Y“.
- `X|Y`: Buchstabe „X“ oder Buchstabe „Y“.
- `(X)`: Ein Paar runder Klammern definiert eine *Gruppe*. Sie können sich mittels `\i` später im Ausdruck auf den Inhalt der i-ten eingeklammerten Gruppe beziehen.
- `^`: Zeilenanfang.
- `$`: Zeilenende.
- `\b`: Eine Wortgrenze.
- `\B`: Eine Nicht-Wortgrenze.
- `\G`: Ende des letzten Treffers.

[55] Jeder der vier regulären Ausdrücke im folgenden Beispiel paßt zur Zeichenkette „Rudolph“:

```
//: strings/Rudolph.java
public class Rudolph {
    public static void main(String[] args) {
        for(String pattern : new String[] { ''Rudolph'', ''[rR]udolph'',
                                            ''[rR][aeiou][a-z]ol.*'', ''R.*'' })
            System.out.println(''Rudolph''.matches(pattern));
    }
} /* Output:
      true
      true
      true
      true
   *///:~
```

[56] Ihr Ziel sollte natürlich nicht sein, den undurchsichtigsten regulären Ausdruck zu konstruieren, sondern die einfachste Lösung finden, um eine Aufgabe zu erledigen. Wenn Sie beginnen, sich mit regulären Ausdrücken zu beschäftigen, werden Sie feststellen, daß Sie beim Zusammensetzen eines neuen Ausdrucks häufig in älteren Ausdrucken nachsehen.

14.6.3 Quantoren

[57] Ein Quantor beschreibt das „Absorptionsverhalten“ eines regulären Ausdrucks:

- *Gierige Quantoren* (*greedy quantifiers*) repräsentieren das Standardverhalten. Ein gieriger Ausdruck findet einen Treffer maximaler Länge. Es ist ein typischer Fehler, davon auszugehen,

daß ein regulärer Ausdruck nur auf das erste Vorkommen einer Gruppe von Zeichen paßt, während es sich in Wirklichkeit gierig verhält und solange weitersucht, bis der längste mögliche Treffer gefunden ist.

- *Genügsame Quantoren* (*reluctant quantifiers*) suchen den kürzesten möglichen Treffer zu einem regulären Ausdruck und werden mit Hilfe eines Fragezeichens definiert. Genügsame Quantoren werden in der englischsprachigen Literatur auch als *lazy-*, *minimal matching-*, *non-greedy-* oder *ungreedy quantifiers* bezeichnet.

- *Possessive Quantoren* (*possessive quantifiers*) sind zur Zeit nur bei Java verfügbar (in keiner anderen Sprache) und haben anspruchsvolle Eigenschaften und Fähigkeiten, so daß Sie sie wahrscheinlich nicht sofort verwenden werden. Ein auf eine Zeichenkette angewendeter regulärer Ausdruck nimmt mehrere Zustände ein, die beim Scheitern der Suche zurückverfolgt werden können. Possessive Quantoren „merken“ sich diese Zwischenzustände *nicht*, wodurch die Rückverfolgung verhindert wird, können verwendet werden, um zu vermeiden, daß ein regulärer Ausdruck „ausreißt“ und seine Ausführung effizienter zu machen.

Gieriges Muster	**Genügsames Muster**	**Possessives Muster**	**Verhalten**
`X?`	`X??`	`X?+`	ein `X` oder keines
`X*`	`X*?`	`X*+`	keines oder beliebig viele `X`
`X+`	`X+?`	`X++`	mindestens ein `X`
`X{n}`	`X{n}?`	`X{n}+`	genau n `X`
`X{n,}`	`X{n,}?`	`X{n,}+`	mindestens n `X`
`X{n,m}`	`X{n,m}?`	`X{n,m}+`	mindestens n und höchstens m `X`

[58] Beachten Sie, daß der durch den Platzhalter `X` in der Tabelle versinnbildlichte Ausdruck häufig in runde Klammern gesetzt werden muß, um sich erwartungsgemäß zu verhalten. Der Ausdruck `abc+` paßt *scheinbar* zu einem oder mehreren Vorkommen der Zeichenkette „abc“, würde also bei „abcabcabc“ drei Treffer finden. Der Ausdruck beschreibt allerdings in Wirklichkeit eine Zeichenkette, bestehend aus dem Anfangsstück „ab“ und wenigstens einem „c“. Das Suchmuster `abc` muß gruppiert werden, um ein oder mehrmals gefunden zu werden: `(abc)+`. Sie können sich bei regulären Ausdrücken leicht täuschen. Reguläre Ausdrücke sind eine auf Java aufgesetzte Sprache.

14.6.3.1 Das Interface CharSequence

[59] Das Interface *`java.lang.CharSequence`* repräsentiert eine verallgemeinerte Definition des Begriffs „Zeichenkette“ und abstrahiert die Klassen `java.nio.CharBuffer`, `String`, `StringBuffer` und `StringBuilder`:

```
interface CharSequence {
    charAt(int i);
    length();
    subSequence(int start, int end);
    toString();
}
```

Die vorgenannten vier Klassen implementieren dieses Interface. Viele Methoden mit regulären Ausdrücken, zum Beispiel die `Matcher`-Methode `split()` (siehe Unterabschnitt 14.6.5) und die `Pattern`-Methode `matches()` (siehe Seite 420), akzeptieren Argumente vom Typ *`CharSequence`*.

14.6.4 Die Klassen Pattern und Matcher

[60] Im allgemeinen werden reguläre Ausdrücke übersetzt und in Form von Objekten darstellt, statt die begrenzten Möglichkeiten der Hilfsmethoden der Klasse `String` anzuwenden. Sie importieren dazu das Package `java.util.regex` und übersetzen einen regulären Ausdruck mit Hilfe der statischen `Pattern`-Methode `compile()`. Die `compile()`-Methode gibt die Referenz auf ein `Pattern`-Objekt zurück, das den als `String`-Argument übergebenen regulären Ausdruck repräsentiert. Nun rufen Sie die Methode `matcher()` des `Pattern`-Objektes auf und übergeben dabei die Zeichenkette, auf die der reguläre Ausdruck angewendet werden soll. Die `matcher()`-Methode gibt wiederum ein `Matcher`-Objekt zurück, über dessen Methoden Sie die Anwendung des regulären Ausdrucks auf die Zeichenkette auswerten können (siehe API-Dokumentation der Klasse `java.util.regex.Matcher`). Beispielsweise ersetzt die Methode `replaceAll()` alle Treffer durch ihr Argument.

[61] Das erste Beispiel zeigt eine Klasse, mit der reguläre Ausdrücke auf eine Zeichenkette angewendet werden können. Das erste Kommandozeilenargument ist die Zeichenkette. Daran schließen sich einer oder mehrere reguläre Ausdrücke an, die auf die Zeichenkette angewendet werden. Bei Unix/Linux müssen die regulären Ausdrücke auf der Kommandozeile in Anführungszeichen gesetzt werden. Das Programm ist nützlich, um reguläre Ausdrücke während des Zusammensetzens zu testen und zu sehen, ob sie sich erwartungsgemäß verhalten:

```
//: strings/TestRegularExpression.java
// Allows you to easily try out regular expressions.
// {Args: abcabcabcdefabc ''abc+'' ''(abc)+'' ''(abc){2,}'' }
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class TestRegularExpression {
    public static void main(String[] args) {
        if(args.length < 2) {
            print(''Usage:\njava TestRegularExpression '' +
                  ''characterSequence regularExpression+'');
            System.exit(0);
        }
        print(''Input: \'''' + args[0] + ''\'''');
        for(String arg : args) {
            print(''Regular expression: \'''' + arg + ''\'''');
            Pattern p = Pattern.compile(arg);
            Matcher m = p.matcher(args[0]);
            while(m.find()) {
                print(''Match \'''' + m.group() + ''\'' at positions '' +
                      m.start() + ''-'' + (m.end() - 1));
            }
        }
    }
} /* Output:
      Input: ''abcabcabcdefabc''
      Regular expression: ''abcabcabcdefabc''
      Match ''abcabcabcdefabc'' at positions 0-14
      Regular expression: ''abc+''
      Match ''abc'' at positions 0-2
      Match ''abc'' at positions 3-5
      Match ''abc'' at positions 6-8
      Match ''abc'' at positions 12-14
      Regular expression: ''(abc)+''
      Match ''abcabcabc'' at positions 0-8
      Match ''abc'' at positions 12-14
```

```
   Regular expression: ''(abc){2,}''
   Match ''abcabcabc'' at positions 0-8
*///:~
```

[62] Ein `Pattern`-Objekt stellt die übersetzte Version eines regulären Ausdrucks dar. Wie Sie an diesem Beispiel nachvollziehen können, wird die `matcher()`-Methode mit der zu durchsuchenden Zeichenkette aufgerufen, um über das `Pattern`-Objekt mit dem übersetzten regulären Ausdruck ein `Matcher`-Objekt zu erzeugen. Die Klasse `Pattern` definiert außerdem die statische Methode `matches()`:

```
static boolean matches(String regex, CharSequence input).
```

Diese Methode prüft, ob der reguläre Ausdruck `regex` die gesamte übergebene Zeichenkette `input` erfaßt sowie eine `split()`-Methode, die ein `String`-Array zurückgibt, welches die verbleibenden Teile einer Zeichenkette nach Trennung bezüglich eines regulären Ausdrucks enthält.

[63] Ein `Matcher`-Objekt wird durch Aufrufen der `matcher()`-Methode eines `Pattern`-Objektes erzeugt, wobei die zu untersuchende Zeichenkette als Argument übergeben wird. Die Klasse `Matcher` definiert die folgenden vier Operationen[5], um die Anwendung des regulären Ausdrucks auf die Zeichenkette auszulösen, sowie zahlreiche Methoden zur Auswertung des Ergebnisses einer `Matcher`-Operation:

```
boolean matches()
boolean lookingAt()
boolean find()
boolean find(int start)
```

Die Operation `matches()` gibt `true` zurück, wenn der reguläre Ausdruck die gesamte Zeichenkette erfaßt. Die Operation `lookingAt()` liefert `true`, wenn der reguläre Ausdruck zu einem Anfangsstück der Zeichenkette paßt (insbesondere also eventuell auch zur gesamten Zeichenkette). Die Operation `find()` ist Gegenstand des folgenden Unterunterabschnitts 14.6.4.1.

Übungsaufgabe 10: (2) Bestimmen Sie, ob die folgenden regulären Ausdrücke, angewendet auf die Zeichenkette „Java now has regular expressions“ einen Treffer liefern: (a) „`^Java`“, (b) „`\Breg.*`“, (c) „`n.w\s+h(a|i)s`“, (d) „`s?`“, (e) „`s*`“, (f) „`s+`“, (g) „`s{4}`“, (h) „`S{1}`“ und (i) „`s{0,3}`“. ■

Übungsaufgabe 11: (2) Wenden Sie den regulären Ausdruck

```
(?i)((^[aeiou])|(\s+[aeiou]))\w+?[aeiou]\b
```

auf die Zeichenkette „Arline ate eight apples and one orange while Anita hadn't any“ an. ■

14.6.4.1 Die Matcher-Operation find()

[64] Die `Matcher`-Operation `find()` löst die Suche nach Treffern des regulären Ausdrucks in der Zeichenkette aus, auf die der Ausdruck angewendet wird, zum Beispiel:

```
//: strings/Finding.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Finding {
    public static void main(String[] args) {
```

[5] Anmerkung des Übersetzers: Die API-Dokumentation der Klasse `Matcher` faßt die drei Methoden `matches()`, `lookingAt()` und `find()` (beide Versionen) unter dem Begriff „match operations“ zusammen. Die deutsche Übersetzung bezeichnet „match operations“ als `Matcher`-*Operationen*, um diese drei Methoden von den „gewöhnlichen“ Methoden der Klasse `Matcher` zu unterscheiden.

```
        Matcher m = Pattern.compile("\\w+")
            .matcher("Evening is full of the linnet's wings");
        while(m.find())
            printnb(m.group() + " ");
        print();
        int i = 0;
        while(m.find(i)) {
            printnb(m.group() + " ");
            i++;
        }
    }
} /* Output:
      Evening is full of the linnet s wings
      Evening vening ening ning ing ng g is is s full full ull ll l of of f the
        the he e linnet linnet innet nnet net et t s s wings wings ings ngs gs s
   *///:~
```

Der reguläre Ausdruck `\\w+` teilt die Zeichenkette in einzelne Worte auf. Die `Matcher`-Operation `find()` verhält sich wie ein Iterator, der sich vorwärts durch die Zeichenkette bewegt. Die zweite Version der `find()`-Operation erwartet ein Argument vom Typ `int`, das die Position des Zeichens angibt, bei dem die Suche beginnt (den Versatz). Die zweite Version der `find()`-Operation setzt die Suchposition auf den Wert ihres Arguments zurück (siehe Ausgabe).

14.6.4.2 Gruppen

[65] Eine Gruppe ist ein durch ein Paar runder Klammern definierter Teil eines regulären Ausdrucks, der anschließend über einen Index referenziert werden kann. Gruppe 0 ist der gesamte Treffer, Gruppe 1 ist das erste Klammerpaar, Gruppe 2 ist das zweite Klammerpaar, und so weiter. Beispielsweise sind durch `A(B(C))D` drei Gruppen definiert: Gruppe 0 ist „ABCD“, Gruppe 1 ist „BC“ und Gruppe 2 ist „C“.

[66–67] Das `Matcher`-Objekt verfügt über die folgenden Methoden zum Abfragen von Informationen über Gruppen:

- `public int groupCount()` liefert die Anzahl der im regulären Ausdruck des `Matcher`-Objektes definierten Gruppen (ohne Gruppe 0).
- `public String group()` liefert Gruppe 0 (den gesamten Treffer) bezüglich der zuletzt aufgerufenen `Matcher`-Operation (zum Beispiel `find()`, beachten Sie Fußnote 5 auf Seite 420).
- `public String group(int i)` liefert die i-te Gruppe bezüglich der zuletzt aufgerufenen `Matcher`-Operation. Falls ein Treffer festgestellt wird, die angeforderte Gruppe aber zu keinem Teil der Zeichenkette paßt, so gibt die Methode `null` zurück.
- `public int start(int group)` liefert den Index des ersten Zeichens der angeforderten Gruppe bezüglich der zuletzt aufgerufenen `Matcher`-Operation.
- `public int end(int group)` liefert den Index des direkten Nachfolgers des letzten Zeichens der angeforderten Gruppe bezüglich der zuletzt aufgerufenen `Matcher`-Operation.

Ein Beispiel:

```
//: strings/Groups.java
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class Groups {
```

```
    static public final String POEM =
        ''Twas brillig, and the slithy toves\n'' +
        ''Did gyre and gimble in the wabe.\n'' +
        ''All mimsy were the borogoves,\n'' +
        ''And the mome raths outgrabe.\n\n'' +
        ''Beware the Jabberwock, my son,\n'' +
        ''The jaws that bite, the claws that catch.\n'' +
        ''Beware the Jubjub bird, and shun\n'' +
        ''The frumious Bandersnatch.'';
    public static void main(String[] args) {
        Matcher m =
            Pattern.compile(''(?m)(\\S+)\\s+((\\S+)\\s+(\\S+))$'').matcher(POEM);
        while(m.find()) {
            for(int j = 0; j <= m.groupCount(); j++)
                printnb(''[' + m.group(j) + '']'');
            print();
        }
    }
} /* Output:
      [the slithy toves][the][slithy toves][slithy][toves]
      [in the wabe.][in][the wabe.][the][wabe.]
      [were the borogoves,][were][the borogoves,][the][borogoves,]
      [mome raths outgrabe.][mome][raths outgrabe.][raths][outgrabe.]
      [Jabberwock, my son,][Jabberwock,][my son,][my][son,]
      [claws that catch.][claws][that catch.][that][catch.]
      [bird, and shun][bird,][and shun][and][shun]
      [The frumious Bandersnatch.][The][frumious Bandersnatch.][frumious] \
        [Bandersnatch.]
   *///:~
```

[68] Das Gedicht „Jabberwocky“ („Der Zipferlak“, in der deutschen Übersetzung von Christian Enzensberger) stammt aus Lewis Carrolls „Through the Looking Glass“ („Alice hinter den Spiegeln“). Der reguläre Ausdruck definiert mehrere eingeklammerte Gruppen, die jeweils eine beliebige Anzahl von Nicht-Leerraumzeichen (`\S+`, großes „S“), gefolgt von einer beliebigen Anzahl von Leerraumzeichen (`\s+`, kleines „s“) erfassen. Das Ziel besteht darin, die letzten drei Worte jeder Zeile abzufangen. Das Dollarzeichen (`$`) repräsentiert das Zeilenende. Im Standardverhalten bezieht sich `$` auf das Ende der gesamten eingegebenen Zeichenkette, so daß Sie dem regulären Ausdruck ausdrücklich mitteilen müssen, daß Zeilenumbrüche in der Zeichenkette beachtet werden sollen. Dies wird mit dem Schalter `(?m)` zu Beginn des Ausdrucks bewerkstelligt (zu Schaltern in regulären Ausdrücken siehe Unterunterabschnitt 14.6.4.4).

Übungsaufgabe 12: (5) Ändern Sie das Beispiel *Groups.java*, so daß alle eindeutige Worte bezählt werden, die nicht mit einem Großbuchstaben beginnen. ■

14.6.4.3 Die Methoden start() und end() der Klasse Matcher

[69] Nach einer erfolgreichen `Matcher`-Operation liefert `start()` den Index des ersten Zeichens des letzten Treffers und `end()` den Index des Nachfolgers des letzten Zeichens, das zum letzten Treffer gehört. Nach einer gescheiterten `Matcher`-Operation (oder vor dem Versuch einer `Matcher`-Operation) rufen beide Methoden eine Ausnahme vom Typ `IllegalStateException` hervor. Das folgende Programm demonstriert auch die `Matcher`-Operationen `matches()` und `lookingAt()`:

```
//: strings/StartEnd.java
import java.util.regex.*;
import static net.mindview.util.Print.*;
```

```
public class StartEnd {
    public static String input =
        ''As long as there is injustice, whenever a\n'' +
        ''Targathian baby cries out, wherever a distress\n'' +
        ''signal sounds among the stars ... We'll be there.\n'' +
        ''This fine ship, and this fine crew ...\n'' +
        ''Never give up! Never surrender!'';[6]
    private static class Display {
        private boolean regexPrinted = false;
        private String regex;
        Display(String regex) { this.regex = regex; }
        void display(String message) {
            if(!regexPrinted) {
                print(regex);
                regexPrinted = true;
            }
            print(message);
        }
    }
    static void examine(String s, String regex) {
        Display d = new Display(regex);
        Pattern p = Pattern.compile(regex);
        Matcher m = p.matcher(s);
        while(m.find())
            d.display(''find() '''' + m.group() +
                      '''' start = ''+ m.start() + '' end = '' + m.end());
        if(m.lookingAt()) // No reset() necessary
            d.display(''lookingAt() start = ''
                      + m.start() + '' end = '' + m.end());
        if(m.matches()) // No reset() necessary
            d.display(''matches() start = ''
                      + m.start() + '' end = '' + m.end());
    }
    public static void main(String[] args) {
        for(String in : input.split(''\n'')) {
            print(''input : '' + in);
            for(String regex : new String[]
                {''\\w*ere\\w*'', ''\\w*ever'', ''T\\w+'', ''Never.*?!''})
                examine(in, regex);
        }
    }
} /* Output:
      input : As long as there is injustice, whenever a
      \w*ere\w*
      find() 'there' start = 11 end = 16
      \w*ever
      find() 'whenever' start = 31 end = 39
      input : Targathian baby cries out, wherever a distress
      \w*ere\w*
      find() 'wherever' start = 27 end = 35
      \w*ever
      find() 'wherever' start = 27 end = 35
      T\w+
      find() 'Targathian' start = 0 end = 10
      lookingAt() start = 0 end = 10
```

[6] Zitat aus einer Rede von Commander Peter Quincy Taggart aus dem Film „Galaxy Quest“.

```
    input : signal sounds among the stars ... We'll be there.
    \w*ere\w*
    find() 'there' start = 43 end = 48
    input : This fine ship, and this fine crew ...
    T\w+
    find() 'This' start = 0 end = 4
    lookingAt() start = 0 end = 4
    input : Never give up! Never surrender!
    \w*ever
    find() 'Never' start = 0 end = 5
    find() 'Never' start = 15 end = 20
    lookingAt() start = 0 end = 5
    Never.*?!
    find() 'Never give up!' start = 0 end = 14
    find() 'Never surrender!' start = 15 end = 31
    lookingAt() start = 0 end = 14
    matches() start = 0 end = 31
*///:~
```

[70] Beachten Sie, daß die `Matcher`-Operation `find()` Treffer des regulären Ausdrucks überall in der Zeichenkette sucht, während `lookingAt()` und `matches()` nur dann einen Treffer melden, wenn der Treffer ein Anfangsstück der Zeichenkette ist. Die Operation `matches()` zeigt nur dann einen Treffer an, wenn der reguläre Ausdruck die gesamte Zeichenkette erfaßt. Die Operation `lookingAt()`[7] liefert bereits `true`, wenn ein Anfangsstück der Zeichenkette zum Ausdruck paßt.

Übungsaufgabe 13: (2) Ändern Sie das Beispiel *StartEnd.java*, so daß es die von `Groups.POEM` referenzierte Zeichenkette verwendet, die `Matcher`-Operationen `find()`, `lookingAt()` und `matches()` aber dennoch `true` zurückgeben. ■

14.6.4.4 Schalter in regulären Ausdrücken

[71] Die Klasse `Pattern` definiert noch eine zweite Version der `compile()`-Methode, die das Verhalten bei der Treffersuche beinflußt:

```
Pattern Pattern.compile(String regex, int flag)
```

Der Parameter `flag` ist eine der folgenden, ebenfalls in der Klasse `Pattern` definierten Konstanten

- `Pattern.CANON_EQ`: Zwei Zeichen passen genau dann zueinander, wenn sie „kanonisch äquivalent“ sind. Beispielsweise paßt der reguläre Ausdruck `\u003F` zur Zeichenkette „?“, wenn dieser Modus aktiviert wird. Im Standardmodus wird die kanonische Äquivalenz nicht berücksichtigt.
- `Pattern.CASE_INSENSITIVE`, `(?i)`: Im Standardmodus erstreckt sich der Verzicht auf die Unterscheidung zwischen Groß- und Kleinschreibung bei der Treffersuche nur auf die Zeichen aus dem ASCII Zeichensatz. Durch Kombination der Schalter `UNICODE_CASE` und `CASE_INSENSITIVE` erweitern Sie die Treffersuche ohne Unterscheidung zwischen Groß- und Kleinschreibung auf Unicodezeichen.
- `Pattern.COMMENTS`, `(?x)`: In diesem Modus wird Leerraum ignoriert und eingebettete Kommentare, die mit dem `#`-Zeichen beginnen, werden bis zum Zeilende ignoriert. Der Unixzeilenmodus (`UNIX_LINES`) kann über die eingebettete Kurznotation `(?d)` zugeschaltet werden.

[7] Ich kann mir nicht erklären, wie die Wahl auf diesen Methodennamen gefallen sein könnte oder worauf er hinweisen soll. Es ist aber beruhigend, zu wissen, daß jeder Mitarbeiter von Sun Microsystems, der unintuitive Bezeichner für Methoden vorschlägt, in der Firma bleibt und daß die Richtlinie, das Design von Quelltexten nicht noch einmal durchzusehen, noch immer praktiziert wird. Entschuldigen den Sarkasmus, aber diese Dinge werden mit der Zeit lästig.

- `Pattern.DOTALL`, `(?s)`: In diesem Modus paßt der Punkt (`.`) zu jedem Zeichen, insbesondere zu Zeilenendezeichen. Im Standardverhalten erfaßt der Punkt Zeilenendezeichen dagegen nicht.
- `Pattern.MULTILINE`, `(?m)`: Im mehrzeiligen Modus passen die Zeichen `^` und `$` sowohl zum Anfang beziehungsweise zum Ende der gesamten Zeichenkette als auch zum Anfang beziehungsweise Ende einer Zeile. Im Standardmodus erfassen die Zeichen `^` und `$` dagegen nur Anfang und Ende der gesamten Zeichenkette.
- `Pattern.UNICODE_CASE`, `(?u)`: In diesem Modus wird die Treffersuche, unter dem per `CASE_INSENSITIVE` zugeschalteten Verzicht auf die Unterscheidung zwischen Groß- und Kleinschreibung, auf alle Unicodezeichen erweitert. Im Standardmodus erstreckt sich der Verzicht auf die Unterscheidung zwischen Groß- und Kleinschreibung bei der Treffersuche nur auf die Zeichen aus dem US-ASCII Zeichensatz.
- `Pattern.UNIX_LINES`, `(?d)`: In diesem Modus erkennen die Zeichen `.`, `^` und `$` nur `\n` als Zeilenendezeichen.

[72] Vor allem die drei Schalter `CASE_INSENSITIVE`, `MULTILINE` und `COMMENTS` sind nützlich (letzterer zur verständlicheren Strukturierung und Dokumentation). Beachten Sie, daß Sie das Verhalten der meisten Schalter auch mit Hilfe der eingeklammerten Kurznotation direkt in Ihrem regulären Ausdruck, vor der Position an welcher der Modus wirksam werden soll, aktivieren können.

[73] Die Wirkungen dieser Schalter können mittels `|`-Operator („Oder“) kombiniert werden:

```
//: strings/ReFlags.java
import java.util.regex.*;

public class ReFlags {
    public static void main(String[] args) {
        Pattern p =
            Pattern.compile(''^java'', Pattern.CASE_INSENSITIVE | Pattern.MULTILINE);
        Matcher m = p.matcher(''java has regex\nJava has regex\n'' +
                              ''JAVA has pretty good regular expressions\n'' +
                              ''Regular expressions are in Java'');
        while(m.find())
            System.out.println(m.group());
    }
} /* Output:
      java
      Java
      JAVA
   *///:~
```

Die `compile()`-Methode übersetzt einen regulären Ausdruck, der auf Zeilen paßt, die mit „java“, „Java“, „JAVA“ und so weiter beginnen und in jeder Zeile einer mehrzeiligen Zeichenkette nach Treffern sucht (potentielle Treffer beginnen am Anfang der Zeichenkette sowie unmittelbar nach jedem Zeilenendezeichen innerhalb der Zeichenkette). Beachten Sie, daß die `group()`-Methode nur den Treffer zurückgibt.

14.6.5 Die split()-Methode der Klasse Pattern

[74] Die `split()`-Methode der Klasse `Matcher` trennt eine Zeichenkette an den durch einen regulären Ausdruck definierten Stellen und gibt ein `String`-Array mit den Teilen der Zeichenkette zurück:

```
String[] split(CharSequence input)
String[] split(CharSequence input, int limit)
```

Die `split()`-Methode eignet sich, um eine Zeile an einem Trennzeichen zu unterteilen:

```
//: strings/SplitDemo.java
import java.util.regex.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class SplitDemo {
    public static void main(String[] args) {
        String input = "This!!unusual use!!of exclamation!!points";
        print(Arrays.toString(Pattern.compile("!!").split(input)));
        // Only do the first three:
        print(Arrays.toString(Pattern.compile("!!").split(input, 3)));
    }
} /* Output:
      [This, unusual use, of exclamation, points]
      [This, unusual use, of exclamation!!points]
   *///:~
```

Die zweite Version der `split()`-Methode begrenzt die Anzahl der Trennungen.

Übungsaufgabe 14: (1) Schreiben Sie das Beispiel *SplitDemo.java* um, so daß es die `String`-Methode `split()` verwendet. ■

14.6.6 Die Ersetzungsmethoden der Klasse Matcher

[75] Reguläre Ausdrücke sind beim Ersetzen von Zeichenketten besonders nützlich. Folgende Methoden stehen zur Verfügung:

- `replaceFirst(String replacement)` ersetzt den ersten Treffer in der Zeichenkette durch den Ersatztext (`replacement`).
- `replaceAll(String replacement)` ersetzt jeden Treffer in der Zeichenkette durch den Ersatztext.
- `appendReplacement(StringBuffer sbuf, String replacement)` ersetzt schrittweise einen Treffer nach dem anderen, wobei die Einzelteile des Ergebnisses in einem Puffer (`sbuf`) gespeichert werden, statt nur den ersten oder alle Treffer zu ersetzen, wie die Methode `replaceFirst()` beziehungsweise `replaceAll()`. Die Methode `appendReplacement()` ist eine sehr wichtige Methode, da sie das Aufrufen einer Methode zur Konstruktion eines flexiblen Ersatztextes ermöglicht (die Methoden `replaceFirst()` und `replaceAll()` können im Gegensatz dazu nur feste Zeichenketten einsetzen). Die Methode `appendReplacement()` gestatten Ihnen, einzelne Gruppen programmatisch auszuwählen und einen anspruchsvollen Ersetzungsmechanismus zu implementieren.
- `appendTail(StringBuffer sbuf, String replacement)` wird nach einem oder mehreren Aufrufen der `appendReplacement()`-Methode aufgerufen um das verbliebene Endstück der Zeichenkette in den Puffer zu übernehmen.

[76] Das folgende Beispiel zeigt die Anwendung aller vier Methoden. Der mehrzeilige Kommentarblock zu Beginn des Beispiels wird mit Hilfe eines regulären Ausdrucks aus der Datei extrahiert und dient als Zeichenkette, auf welche die vier Methoden mit weiteren regulären Ausdrücken angewendet werden:

```
//: strings/TheReplacements.java
import java.util.regex.*;
import net.mindview.util.*;
```

```
import static net.mindview.util.Print.*;

/*! Here's a block of text to use as input to
   the regular expression matcher. Note that we'll
   first extract the block of text by looking for
   the special delimiters, then process the
   extracted block. !*/

public class TheReplacements {
    public static void main(String[] args) throws Exception {
        String s = TextFile.read("TheReplacements.java");
        // Match the specially commented block of text above:
        Matcher mInput =
            Pattern.compile("/\\*!(.*)!\\*/", Pattern.DOTALL).matcher(s);
        if(mInput.find())
            s = mInput.group(1); // Captured by parentheses
        // Replace two or more spaces with a single space:
        s = s.replaceAll(" {2,}", " ");
        // Replace one or more spaces at the beginning of each
        // line with no spaces. Must enable MULTILINE mode:
        s = s.replaceAll("(?m)^ +", "");
        print(s);
        s = s.replaceFirst("[aeiou]", "(VOWEL1)");
        StringBuffer sbuf = new StringBuffer();
        Pattern p = Pattern.compile("[aeiou]");
        Matcher m = p.matcher(s);
        // Process the find information as you
        // perform the replacements:
        while(m.find())
            m.appendReplacement(sbuf, m.group().toUpperCase());
        // Put in the remainder of the text:
        m.appendTail(sbuf);
        print(sbuf);
    }
} /* Output:
      Here's a block of text to use as input to
      the regular expression matcher. Note that we'll
      first extract the block of text by looking for
      the special delimiters, then process the
      extracted block.
      H(VOWEL1)rE's A blOck Of tExt tO UsE As InpUt tO
      thE rEgUlAr ExprEssIOn mAtchEr. NOtE thAt wE'll
      fIrst ExtrAct thE blOck Of tExt by lOOkIng fOr
      thE spEcIAl dElImItErs, thEn prOcEss thE
      ExtrActEd blOck.
   *///:~
```

[77] Der Quelltext des Beispiels *TheReplacements.java* wird mit Hilfe der Klasse `net.mindview.util.TextFile` (siehe Unterabschnitt 19.7.1) geöffnet und eingelesen. Die statische Methode `read()` liest den gesamten Inhalt der Datei und gibt ihn als `String`-Objekt zurück. Das von `mInput` referenzierte `Matcher`-Objekt repräsentiert den Text zwischen den Zeichen `/*!` und `!*/` (beachten Sie die Gruppierung). Anschließend werden je zwei benachbarte Leerzeichen in ein einziges Leerzeichen umgewandelt und führende Leerzeichen aus jeder Zeile entfernt (der reguläre Ausdruck muß im mehrzeiligen Modus angewendet werden, um die Ersetzung in allen Zeilen, statt nur zu Beginn der gesamten Eingabe vorzunehmen). Beide Ersetzungen werden mit Hilfe der äquivalenten und gleichnamigen `String`-Methode `replaceAll()` vorgenommen. Da jede Ersetzung in diesem Programm nur einmal vorgenommen wird, verursacht die Entscheidung für die obige Vorgehensweise, anstelle

der Übersetzung eines eigenen regulären Ausdrucks, keine zusätzlichen Unkosten.

[78] Die `replaceFirst()`-Methode ersetzt nur den ersten Treffer. Der Ersatztext der Methoden `replaceFirst()` und `replaceAll()` ist darüber hinaus stets nur ein Literal und scheidet folglich aus, wenn der Ersatztext flexibel konstruiert werden soll. In diesem Fall müssen Sie auf die Methode `appendReplacement()` zurückgreifen, die während des Ersetzungsvorgangs beliebige Verarbeitung des Treffers gestattet. Im obigen Beispiel wird eine Gruppe ausgewählt, durchläuft eine Verarbeitung (der von dem regulären Ausdruck gefundene Vokal wird in einen Großbuchstaben umgewandelt), woraufhin das Ergebnis in dem von `sbuf` referenzierten `StringBuffer`-Objekt zusammengefügt wird. Normalerweise wählen Sie einen Treffer nach dem anderen aus, nehmen die jeweilige Ersetzung vor und rufen dann `appendTail()` auf. Wenn Sie das Verhalten von `replaceFirst()` nachahmen wollen, nehmen Sie nur eine Ersetzung vor und rufen anschließend `appendTail()` auf, um die restliche Zeichenkette an den Inhalt des `StringBuffer`-Objektes anzuhängen.

[79] Die `appendReplacement()`-Methode gestattet das Referenzieren von Gruppen im Ersatztext über die Notation `$g` für die g-te Gruppe. However, this is for simpler processing und würde im obigen Programm nicht zum gewünschten Ergebnis führen.

14.6.7 Die reset()-Methode der Klasse Matcher

[80] Ein vorhandenes `Matcher`-Objekt kann mit Hilfe seiner überladenen `reset()`-Methode auf eine neue Zeichenkette umgesetzt werden:

```
//: strings/Resetting.java
import java.util.regex.*;

public class Resetting {
    public static void main(String[] args) throws Exception {
        Matcher m =
            Pattern.compile("[frb][aiu][gx]").matcher("fix the rug with bags");
        while(m.find())
            System.out.print(m.group() + " ");
        System.out.println();
        m.reset("fix the rig with rags");
        while(m.find())
            System.out.print(m.group() + " ");
    }
} /* Output:
      fix rug bag
      fix rig rag
   *///:~
```

Die argumentlose Version setzt das `Matcher`-Objekt auf den Anfang der aktuellen Zeichenkette zurück.

14.6.8 Anwendungsbeispiel: Reguläre Ausdrücke zur Suche in Textdateien

[81] In den meisten bisherigen Beispielen wurden reguläre Ausdrücke auf statische Zeichenketten angewendet. Das folgende Beispiel zeigt die Anwendung regulärer Ausdrücke, um im Inhalt einer Datei nach Treffern zu suchen. Angeregt durch das Unix-Kommando `grep`, erwartet *JGrep.java* zwei Argumente: den Dateinamen und den regulären Ausdruck, der das Suchmuster beschreibt. Die Ausgabe zeigt jede Zeile, die einen Treffer enthält und gibt auch die Positionen des/der Treffer in der Zeile an:

```
//: strings/JGrep.java
// A very simple version of the ''grep'' program.
// {Args: JGrep.java ''\\b[Ssct]\\w+''}
import java.util.regex.*;
import net.mindview.util.*;

public class JGrep {
    public static void main(String[] args) throws Exception {
        if(args.length < 2) {
            System.out.println('Usage: java JGrep file regex');
            System.exit(0);
        }
        Pattern p = Pattern.compile(args[1]);
        // Iterate through the lines of the input file:
        int index = 0;
        Matcher m = p.matcher('');
        for(String line : new TextFile(args[0])) {
            m.reset(line);
            while(m.find())
                System.out.println(index++ + ': ' + m.group() + ': ' + m.start());
        }
    }
} /* Output: (Sample)
      0: strings: 4
      1: simple: 10
      2: the: 28
      3: Ssct: 26
      4: class: 7
      5: static: 9
      6: String: 26
      7: throws: 41
      8: System: 6
      9: System: 6
      10: compile: 24
      11: through: 15
      12: the: 23
      13: the: 36
      14: String: 8
      15: System: 8
      16: start: 31
   *///:~
```

[82] Die Textdatei wird mit Hilfe eines `net.mindview.util.TextFile`-Objektes (siehe Unterabschnitt 19.7.1) geöffnet, welches die Zeilen der Datei in einen Container vom Typ `ArrayList` einliest. Somit kann die erweiterte `for`-Schleife verwendet werden, um eine Zeile der Textdatei nach der anderen auszuwählen.

[83] Es wäre zwar möglich, bei jedem Schleifendurchgang ein neues `Matcher`-Objekt zu erzeugen, aber eine bessere Lösung besteht darin, außerhalb der Schleife ein leeres `Matcher`-Objekt zu erzeugen und per `reset()`-Methode bei jeder Iteration eine weitere Zeile zuzuweisen.

[84] Die im Kopfbereich des Quelltextes angegebenen Testargumente öffnen die Datei *JGrep.java* selbst und suchen nach Wörtern, die mit einem der vier Buchstaben „S“, „s“, „c“ oder „t“ beginnen.

[85] In Jeffrey E. F. Friedls *Mastering Regular Expressions*, 3rd ed., O'Reilly (2006)[8] finden Sie umfassende und tiefgreifende Informationen über das Thema „Reguläre Ausdrücke". Es gibt zahlreiche Einführungen im Internet und Sie können in der Dokumentation von Skriptsprachen wie Perl und Python häufig hilfreiche Tips finden.

Übungsaufgabe 15: (5) Ändern Sie das Beispiel *JGrep.java*, so daß der reguläre Ausdruck mit Schaltern übersetzt wird, die als Kommandozeilenargumente übergegeben werden können (zum Beispiel `Pattern.CASE_INSENSITIVE` oder `Pattern.MULTILINE`). ■

Übungsaufgabe 16: (5) Ändern Sie das Beispiel *JGrep.java*, so daß entweder ein Datei- oder ein Verzeichnisname als Kommandozeilenargument übergeben werden kann (bei Übergabe eines Verzeichnisnamens umfaßt die Suche alle Dateien in diesem Verzeichnis). Tip: Die folgende Anweisung erzeugt eine Liste von Dateinamen: `File[] files = new File(''.'').listFiles();` ■

Übungsaufgabe 17: (8) Schreiben Sie ein Programm, das eine *.java* Datei einliest (Dateiname wird per Kommandozeile übergeben) und alle Kommentare anzeigt. ■

Übungsaufgabe 18: (8) Schreiben Sie ein Programm, das eine *.java* Datei einliest (Dateiname wird per Kommandozeile übergeben) und alle literal definierten Zeichenketten anzeigt. ■

Übungsaufgabe 19: (8) Schreiben Sie, aufbauend auf den beiden vorigen Übungsaufgaben, ein Programm, das eine *.java* Datei einliest und alle darin verwendeten Klassennamen anzeigt. ■

14.7 Die Klasse Scanner

[86] Das Einlesen von Daten aus einer Datei mit menschenlesbarem Format oder von der Standardeingabe war bis jetzt relativ anstrengend. Üblicherweise wird eine Textzeile eingelesen und an einem Trennzeichen unterteilt. Anschließend mußten die Felder mit den entsprechenden Methoden der Wrapperklassen `Integer`, `Double` und so weiter geparst werden:

```
//: strings/SimpleRead.java
import java.io.*;

public class SimpleRead {
    public static BufferedReader input =
        new BufferedReader(new StringReader(''Sir Robin of Camelot\n22 1.61803''));
    public static void main(String[] args) {
        try {
            System.out.println(''What is your name?'');
            String name = input.readLine();
            System.out.println(name);
            System.out.println(''How old are you? What is your favorite double?'');
            System.out.println(''(input: <age> <double>)'');
            String numbers = input.readLine();
            System.out.println(numbers);
            String[] numArray = numbers.split('' '');
            int age = Integer.parseInt(numArray[0]);
            double favorite = Double.parseDouble(numArray[1]);
            System.out.format(''Hi %s.\n'', name);
            System.out.format(''In 5 years you will be %d.\n'', age + 5);
            System.out.format(''My favorite double is %f.'', favorite / 2);
        } catch(IOException e) {
```

[8] Anmerkung des Übersetzers: Die deutsche Übersetzung der dritten Auflage von Andreas Karrer ist unter dem Titel „Reguläre Ausdrücke" im Jahre 2007 bei O'Reilly erschienen.

```
            System.err.println("I/O exception");
        }
    }
} /* Output:
      What is your name?
      Sir Robin of Camelot
      How old are you? What is your favorite double?
      (input: <age> <double>)
      22 1.61803
      Hi Sir Robin of Camelot.
      In 5 years you will be 27.
      My favorite double is 0.809015.
   *///:~
```

[87] Das `input`-Feld bedient sich zweier Klassen aus der Ein-/Ausgabebibliothek von Java, die offiziell erst in Kapitel 19 eingeführt werden. Die Klasse `StringReader` wandelt ein `String`-Objekt in einen lesbaren Zeichenstrom um, der verwendet wird, um ein `BufferedReader`-Objekt zu erzeugen, da die Klasse `BufferedReader` eine `readLine()`-Methode definiert. Das Ergebnis besteht darin, daß der Inhalt des von `input` referenzierten Objektes zeilenweise abgefragt werden kann, wie der Standardeingabekanal von der Konsole.

[88] Die `readLine()`-Methode liefert jede einzelne Zeile der Eingabe als `String`-Objekt. Solange Sie pro Zeile nur einen einzigen Wert erfassen, ist der Ansatz im vorigen Beispiel eine praktikable Lösung. Enthält eine Zeile aber zwei Werte, wird die Situation unangenehm: Die Zeile muß getrennt werden, um jeden Wert separat parsen zu können. Die Trennung findet beim Bewerten des `String`-Arrays `numArray` statt. Die `split()`-Methode wurde erst in Version 1.4 des Java Development Kits vorgestellt, das heißt zuvor war wiederum ein anderer Lösungsweg erforderlich.

[89] Die seit der SE 5 vorhandene Klasse `java.util.Scanner` entlastet den Programmierer beim Einlesen von Eingaben von einem großen Teil der Bürde:

```
//: strings/BetterRead.java
import java.util.*;

public class BetterRead {
    public static void main(String[] args) {
        Scanner stdin = new Scanner(SimpleRead.input);
        System.out.println("What is your name?");
        String name = stdin.nextLine();
        System.out.println(name);
        System.out.println("How old are you? What is your favorite double?");
        System.out.println("(input: <age> <double>)");
        int age = stdin.nextInt();
        double favorite = stdin.nextDouble();
        System.out.println(age);
        System.out.println(favorite);
        System.out.format("Hi %s.\n", name);
        System.out.format("In 5 years you will be %d.\n", age + 5);
        System.out.format("My favorite double is %f.", favorite / 2);
    }
} /* Output:
      What is your name?
      Sir Robin of Camelot
      How old are you? What is your favorite double?
      (input: <age> <double>)
      22
      1.61803
```

```
   Hi Sir Robin of Camelot.
   In 5 years you will be 27.
   My favorite double is 0.809015.
*///:~
```

[90] Die Konstruktoren der Klasse `Scanner` akzeptieren viele Typen von Eingabeobjekten, darunter `File` (siehe Abschnitt 19.1), `InputStream`, `String` oder wie im obigen Fall *`Readable`*, ein in der SE 5 erstmals auftretendes Interfaces zur Beschreibung aller Typen, die eine `read()`-Methode besitzen. Die Klasse `BufferedReader` aus dem Beispiel *SimpleRead.java* von Seite 430 fällt in diese Kategorie.

[91] Die Klasse `Scanner` implementiert das Einlesen, die Teilung von Zeichenketten an einem Trennzeichen und das Parsen mittels einer Vielzahl von `nextXXX()`-Methoden. Die einfache `next()`-Methode (ohne typspezifische „Endung“) gibt den nächsten vollständigen Wert als `String`-Objekt zurück. `Scanner` definiert `nextXXX()`-Methoden für alle primitiven Typen außer `char` sowie für `BigDecimal` und `BigInteger`. Alle `nextXXX()`-Methoden *blockieren*, kehren also erst zurück, nachdem ein vollständiger Wert verfügbar geworden ist. Die korrespondierenden `hasNextXXX()`-Methoden geben `true` zurück, wenn der nächste verfügbare Wert den entsprechenden Typ hat.

[92] Das Fehlen der `try`-Klausel für Ausnahmen vom Typ `IOException` in *BetterRead.java*, ist ein interessanter Unterschied zwischen den beiden Beispielen in diesem Abschnitt. Die Klasse `Scanner` nimmt beim Auswerfen einer Ausnahme vom Typ `IOException` an, daß die Eingabe beendet ist und verschluckt Ausnahmen dieses Typs. Die letzte Ausnahme kann aber über die Methode `ioException()` abgefragt und bei Bedarf ausgewertet werden.

Übungsaufgabe 20: (2) Schreiben Sie eine Klasse mit Feldern der Typen `int`, `long`, `float`, `double` und `String`. Legen Sie einen Konstruktor an, der ein einzelnes `String`-Objekt erwartet, sein Argument auftrennt und die verschiedenen Felder bewertet. Legen Sie auch eine `toString()`-Methode an, um zu zeigen, daß Ihre Klasse korrekt funktioniert. ■

14.7.1 Reguläre Ausdrücke als Trennzeichen

[93] Ein `Scanner`-Objekt verwendet per Voreinstellung Leerraum als Trennzeichen, aber Sie können auch ein eigenes Trennzeichen in Form eines regulären Ausdrucks definieren:

```
//: strings/ScannerDelimiter.java
import java.util.*;

public class ScannerDelimiter {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(''12, 42, 78, 99, 42'');
        scanner.useDelimiter(''\\s*,\\s*'');
        while(scanner.hasNextInt())
            System.out.println(scanner.nextInt());
    }
} /* Output:
      12
      42
      78
      99
      42
   *///:~
```

[94] Dieses Beispiel definiert Kommata, umgeben von beliebig viel Leerraum als Trennzeichen beim Einlesen der gegebenen Zeichenkette. Dieselbe Vorgehensweise gestattet das Einlesen einer Datei mit kommaseparierten Werten. Neben der `useDelimiter()`-Methode, die einen regulären Ausdruck

als Trennzeichen definiert, liefert die `delimiter()`-Methode das `Pattern`-Objekt mit dem aktuell verwendeten Trennzeichen.

14.7.2 Einlesen komplex strukturierter Daten

[95] Außer Werten für die vordefinierten primitiven Typen, können Sie Eingaben auch bezüglich eines selbstdefinierten regulären Ausdrucks parsen. Diese Möglichkeit ist beim Einlesen komplex strukturierter Daten nützlich. Das folgende Beispiel liest Daten aus der Protokolldatei einer Firewall ein:

```
//: strings/ThreatAnalyzer.java
import java.util.regex.*;
import java.util.*;

public class ThreatAnalyzer {
    static String threatData =
        ''58.27.82.161@02/10/2005\n'' +
        ''204.45.234.40@02/11/2005\n'' +
        ''58.27.82.161@02/11/2005\n'' +
        ''58.27.82.161@02/12/2005\n'' +
        ''58.27.82.161@02/12/2005\n'' +
        ''[Next log section with different data format]'';
    public static void main(String[] args) {
        Scanner scanner = new Scanner(threatData);
        String pattern = ''(\\d+[.]\\d+[.]\\d+[.]\\d+)@'' +
            ''(\\d{2}/\\d{2}/\\d{4})'';
        while(scanner.hasNext(pattern)) {
            scanner.next(pattern);
            MatchResult match = scanner.match();
            String ip = match.group(1);
            String date = match.group(2);
            System.out.format(''Threat on %s from %s\n'', date,ip);
        }
    }
} /* Output:
      Threat on 02/10/2005 from 58.27.82.161
      Threat on 02/11/2005 from 204.45.234.40
      Threat on 02/11/2005 from 58.27.82.161
      Threat on 02/12/2005 from 58.27.82.161
      Threat on 02/12/2005 from 58.27.82.161
   *///:~
```

[96] Wenn Sie die `next()`-Methode mit einem regulären Ausdruck aufrufen, so wird das nächste input token mit diesem Ausdruck verglichen. Das Ergebnis des Vergleichs wird über die `match()`-Methode verfügbar gemacht. Die Vorgehensweise entspricht der Trefferauswertung bei regulären Ausdrücken, die Sie bereits kennen.

[97] Eine Warnung zum Einlesen mit Hilfe eines regulären Ausdrucks: Der Ausdruck wird nur mit dem nächsten input token verglichen. Enthält Ihr Ausdruck ein Trennzeichen, so wird kein Treffer gefunden.

14.8 Die Klasse StringTokenizer

[98] Vor der Aufnahme der regulären Ausdrücke (im JDK 1.4) beziehungsweise der Klasse `Scanner` (in der SE 5) in den Sprachumfang, wurden Zeichenketten mit Hilfe der Klasse `java.util.StringTokenizer` zerlegt. Reguläre Ausdrücke und die Klasse `Scanner` ermöglichen heute viel einfachere und prägnantere Lösungswege. Das folgende Beispiel dient zum Vergleichen des älteren „`StringTokenizer`-Ansatzes“ mit den beiden anderen Verfahren:

```
//: strings/ReplacingStringTokenizer.java
import java.util.*;

public class ReplacingStringTokenizer {
    public static void main(String[] args) {
        String input = ''But I'm not dead yet! I feel happy!'';
        StringTokenizer stoke = new StringTokenizer(input);
        while(stoke.hasMoreElements())
            System.out.print(stoke.nextToken() + '' '');
        System.out.println();
        System.out.println(Arrays.toString(input.split('' '')));
        Scanner scanner = new Scanner(input);
        while(scanner.hasNext())
            System.out.print(scanner.next() + '' '');
    }
} /* Output:
      But I'm not dead yet! I feel happy!
      [But, I'm, not, dead, yet!, I, feel, happy!]
      But I'm not dead yet! I feel happy!
   *///:~
```

[99] Reguläre Ausdrücke und die Klasse `Scanner` gestatten außerdem die Trennung einer Zeichenkette an Stellen, die durch komplexe Muster definiert sind. Derartige Anforderungen sind mit der Klasse `StringTokenizer` schwierig zu realisieren. Es scheint gesichert, daß die Klasse `StringTokenizer` überholt ist.

14.9 Zusammenfassung

[100] In der Vergangenheit wurde die Verarbeitung von Zeichenketten von Java nur rudimentär unterstützt. Die letzten Sprachversionen haben eine äußerst anspruchsvolle Unterstützung entwickelt, wie sie auch bei anderen Sprachen implementiert ist. Die Unterstützung der Verarbeitung von Zeichenketten ist heute ausgereift. Dennoch müssen Sie gelegentlich im Hinblick auf die Effizienz auf Einzelheiten beachten, beispielsweise in passenden Situationen `StringBuilder`-Objekte wählen.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 15

RTTI und Reflexion: Typidentifikation zur Laufzeit

Inhaltsübersicht

[0] Der Mechanismus der *Typidentifikation zur Laufzeit* (RTTI, *runtime type identification*) gestattet Ihnen, Informationen über Typen auszuwerten und zu nutzen, während ein Programm oder eine Anwendung ausgeführt wird.

[1] Die Typidentifikation zur Laufzeit befreit Sie von der Einschränkung, typbezogene Operationen nur zur Übersetzungszeit ausführen zu können und ermöglicht einige sehr mächtige Ansätze. Die Notwendigkeit des RTTI-Mechanismus' deckt eine Vielzahl interessanter (und nicht selten verwirrender) Gesichtspunkte der objektorientierten Programmierung auf und führt zu fundamentalen Fragen hinsichtlich der Art und Weise, ein Programm zu strukturieren.

[2] Dieses Kapitel betrachtet die Möglichkeiten, mit denen Ihnen Java gestattet, zur Laufzeit Informationen über Objekte und Klassen einzuholen. Es gibt zwei Varianten: „Traditionelles“ RTTI setzt

voraus, daß die Typen zur Übersetzungszeit verfügbar sind, während der *Reflexionsmechanismus* ausschließlich zur Laufzeit Typinformationen liefert.

15.1 Motivation zur Typidentifikation zur Laufzeit

[3] Abbildung 15.1 zeigt das mittlerweise vertraute Beispiel einer Klassenhierarchie mit einer polymorphen Methode. Die abstrakte Basisklasse `Shape` repräsentiert den allgemeinen Typ und die abgeleiteten Klassen `Circle`, `Square` und `Triangle` die spezialisierten Typen. Die Abbildung zeigt das typische Diagramm einer Klassenhierarchie mit oben notierter Basisklasse und den darunter angeordneten abgeleiteten Klassen. In der objektorientierten Programmierung bezieht sich Ihr Quelltext normalerweise auf eine Referenzvariable des Basistyps (in diesem Beispiel also `Shape`), so daß Sie den größten Teil des Quelltextes unverändert belassen können, wenn Sie das Programm um eine neue von `Shape` abgeleitete Klasse erweitern (beispielsweise `Rhomboid`). Die dynamisch gebundene Methode ist in diesem Beispiel die `draw()`-Methode, das heißt ein Programmierer, der diese Klassenhierarchie verwendet, sollte die `draw()`-Methode über eine allgemeine Referenzvariable vom Typ `Shape` aufrufen. Die `draw()`-Methode ist in jeder von `Shape` abgeleiteten Klasse überschrieben und weil die Methode dynamisch gebunden ist, stellt sich das passende Verhalten ein, obwohl die Methode über eine Referenzvariable vom Typ `Shape` aufgerufen wird. Dies ist das Wesen der Polymorphie.

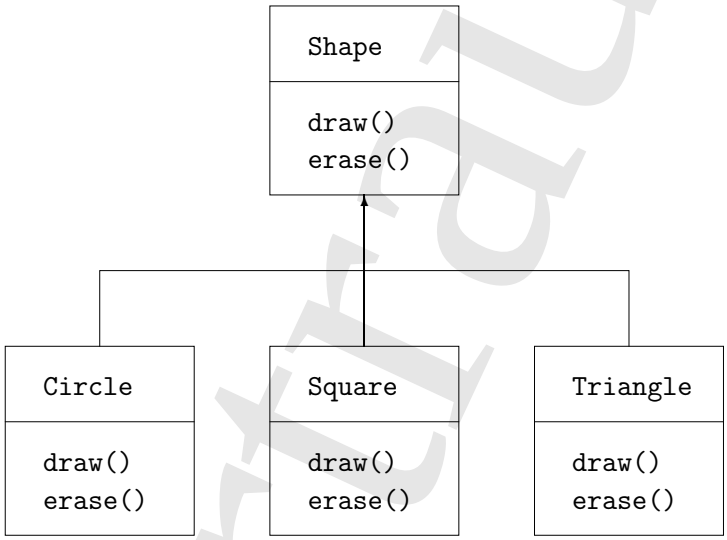


Abbildung 15.1: [illegible]

[4] Im allgemeinen erzeugen Sie also ein Objekt einer spezialisierten Klasse (`Circle`, `Square` oder `Triangle`), wandeln die Objektreferenz aufwärts in `Shape` um (wobei Sie den eigentlichen Typ des Objektes „vergessen“) und arbeiten im restlichen Programm mit dieser anonymisierten `Shape`-Referenz.

[5] Sie können die Hierarchie unter der abstrakten Basisklasse `Shape` zum Beispiel so implementieren:

```
//: typeinfo/Shapes.java
import java.util.*;

abstract class Shape {
    void draw() { System.out.println(this + ".draw()"); }
    abstract public String toString();
}
```

```
class Circle extends Shape {
    public String toString() { return ''Circle''; }
}

class Square extends Shape {
    public String toString() { return ''Square''; }
}

class Triangle extends Shape {
    public String toString() { return ''Triangle''; }
}

public class Shapes {
    public static void main(String[] args) {
        List<Shape> shapeList =
            Arrays.asList(new Circle(), new Square(), new Triangle());
        for(Shape shape : shapeList)
            shape.draw();
    }
} /* Output:
      Circle.draw()
      Square.draw()
      Triangle.draw()
   *///:~
```

Die Basisklasse `Shape` definiert eine `draw()`-Methode, die indirekt die `toString()`-Methode aufruft (durch die Übergabe der Selbstreferenz `this` an die `System.out`-Methode `println()`), um eine Zeichenkette auszugeben, über welche die Klasse identifiziert werden kann. (Beachten Sie, daß die `toString()`-Methode als abstrakte Methode deklariert ist, um die Überschreibung in abgeleiteten Klassen zu erzwingen und zu Verhindern, daß `Shape`-Objekte erzeugt werden.) Enthält ein Konkatenierungsausdruck für Zeichenketten (ein Ausdruck, der den Konkatenationsoperator `+` und ein `String`-Objekt enthält) ein Objekt, so wird automatisch die `toString()`-Methode dieses Objektes aufgerufen, um eine Darstellung des Objektes als Zeichenkette zu bekommen. Jede von `Shape` abgeleitete Klasse überschreibt die von `Object` vererbte `toString()`-Methode, so daß `draw()` aufgrund der Polymorphie für jede abgeleitete Klasse eine andere Ausgabe liefert.

[6] In diesem Beispiel vollzieht sich die aufwärts gerichtete Typumwandlung, wenn die Objekte in den *`List<Shape>`*-Container eingesetzt werden. Während der Typumwandlung geht die Tatsache verloren, daß die Objekte *spezialisierte Untertypen* von `Shape` sind. Aus der Perspektive des Arrays sind alle Objekte vom Typ `Shape`.

[7] Bei der Entnahme eines Objektes aus dem Array wandelt der Container, der eigentlich alle Elemente dem Typ `Object` zuordnet, die herausgegebene Referenz automatisch in den Typ `Shape` zurück. Diese ist die einfachste Form der Typidentifikation zur Laufzeit, da alle Typumwandlungen zur Laufzeit auf ihre Korrektheit hin geprüft werden. Typidentifikation zur Laufzeit bedeutet, daß der Typ eines Objektes zur Laufzeit bestimmt wird.

[8] Im obigen Fall wird die abwärts gerichtete Typumwandlung nur teilweise vollzogen, nämlich von `Object` nach `Shape`, nicht aber bis hin zu `Circle`, `Square` oder `Triangle`. Das liegt daran, daß bei der Entnahme eines Objektes aus einem Container vom Typ *`List<Shape>`* nicht mehr bekannt ist, als daß der Container nur Objekte vom Typ `Shape` enthält. Diese Typsicherheit wird zur Übersetzungszeit vom Container und den generischen Typen von Java gewährleistet, zur Laufzeit aber durch Typumwandlung.

[9] Die Polymorphie bewirkt nun, daß die exakte über die `Shape`-Referenz aufgerufene Methode danach ermittelt wird, ob die Referenz auf ein `Circle`-, `Square`- oder `Triangle`-Objekt verweist. So sollte es in der Regel sein: Der größte Teil Ihres Quelltext sollte so wenig wie möglich über den

eigentlichen Typ eines Objektes „wissen“ und nur mit einer allgemein gültigen Schnittstelle einer Familie von Objekten arbeiten (hier Objekte vom Typ `Shape`). Ihr Quelltext ist dadurch leichter zu schreiben, zu lesen und zu pflegen. Ihr Design ist leichter zu implementieren, zu verstehen und zu ändern. Polymorphie ist ein generelles Ziel in der objektorientierten Programmierung.

[10] Stellen Sie sich nun eine Programmieraufgabe vor, die sich am leichtesten lösen läßt, wenn Sie den exakten Typ des Objektes kennen, auf das eine allgemeine Referenzvariable verweist. Angenommen, Sie möchten den Benutzern Ihres Programms die Möglichkeit anbieten, alle Figuren eines bestimmten Typs hervorzuheben, indem sie in einer speziellen Farbe gezeichnet werden. Vielleicht gibt es eine Methode, die eine Anzahl von Figuren um einen bestimmten Winkel rotiert. Es ist unsinnig, die Kreise ebenfalls zu drehen und Sie möchten die `Circle`-Objekte daher ausschließen. Typidentifikation zur Laufzeit gestattet Ihnen, eine Referenzvariable vom Typ `Shape` zu befragen, welchem Typ das von ihr referenzierte Objekt angehört, also Speziallfälle auszuwählen und zu isolieren.

15.2 Das Klassenobjekt

[11] Das Verstehen der Funktionsweise des RTTI-Mechanismus' bei Java setzt voraus, daß Sie wissen, wie Typinformationen zur Laufzeit dargestellt werden. Zu diesem Zweck gibt es einen speziellen Objekttyp, der Informationen über eine Klasse enthält, nämlich das sogenannte *Klassenobjekt* (ein Objekt der Klasse `java.lang.Class`). Das Klassenobjekt dient eigentlich dazu, alle „gewöhnlichen“ Objekte der jeweiligen Klasse zu erzeugen. Der RTTI-Mechanismus von Java bezieht sich auf das Klassenobjekt, selbst bei Typumwandlungen. Die Klasse `Class` implementiert noch weitere Möglichkeiten zur Typidentifikation zur Laufzeit.

[12] Zu jeder Klasse Ihres Programms gibt es ein Klassenobjekt. Jedesmal wenn Sie eine neue Klasse schreiben und übersetzen, wird auch ein einzelnes Klassenobjekt erzeugt (und in einer gleichnamigen Datei mit der Endung *.class* gespeichert). Die Laufzeitumgebung die Ihr Programm ausführt, ruft ein Subsystem auf, um ein Objekt dieser Klasse zu erzeugen, nämlich den *Klassenlader*.

[13] Das Klassenladersubsystem kann aus einer Verkettung von Klassenladern bestehen, wobei es nur einen *primordialen Klassenlader* gibt, der ein Bestandteil der Implementierung der Laufzeitumgebung ist. Der primordiale Klassenlader lädt, typischerweise von der Festplatte, sogenannte *gesicherte Klassen* (*trusted classes*), zu denen die Klassen der Java-API gehören. In der Regel sind keine weiteren Klassenlader notwendig, können aber bei Bedarf in die Kette der Klassenlader aufgenommen werden (beispielsweise Klassen von Webapplikationen oder über ein Netzwerk heruntergeladene Klassen).

[14] Alle Klassen werden dynamisch beim ersten Vorkommen in die Laufzeitumgebung geladen, genauer, wenn das Programm die erste Referenz auf eine statische Komponente der jeweiligen Klasse anfordert. Der Konstruktor ist ebenfalls eine statische Methode, obwohl Konstruktoren nicht explizit mit dem Schlüsselwort `static` gekennzeichnet werden. Das Erzeugen eines neuen Objektes einer Klasse per `new`-Operator zählt somit als Anfordern einer Referenz auf eine statische Komponente der Klasse.

[15] Ein Java-Programm wird also vor Beginn seiner Ausführung nicht vollständig geladen, sondern die einzelnen Teile werden bei Bedarf nachgeladen. Java unterscheidet sich in dieser Hinsicht von vielen traditionellen Sprachen. Das dynamische Laden von Klassen ermöglicht Verhaltsmerkmale, die in einer statisch geladenen Sprache wie C++ schwierig oder sogar unmöglich nachgebildet werden können.

[16] Der Klassenlader prüft zuerst, ob das Klassenobjekt des benötigten Typs bereits geladen wurde.

Falls nicht, sucht der Standardklassenlader die entsprechende *.class* Datei (ein zusätzlicher Klassenlader könnte zum Beispiel den Bytecode stattdessen in einer Datenbank suchen). Der Bytecode wird nach dem Laden verifiziert, um sicherzustellen, daß er nicht korrumpiert wurde und keinen schädlichen Java-Code enthält (dies ist eine der zur Sicherheit von Java errichteten „Verteidigungslinien“).

[17] Das Klassenobjekt wird nach der Ablage im Arbeitsspeicher verwendet, um sämtliche Objekte des jeweiligen Typs zu erzeugen. Das folgende Beispiel beweist diese Behauptung experimentell:

```
//: typeinfo/SweetShop.java
// Examination of the way the class loader works.
import static net.mindview.util.Print.*;

class Candy {
    static { print("Loading Candy"); }
}

class Gum {
    static { print("Loading Gum"); }
}

class Cookie {
    static { print("Loading Cookie"); }
}

public class SweetShop {
    public static void main(String[] args) {
        print("inside main");
        new Candy();
        print("After creating Candy");
        try {
            Class.forName("Gum");
        } catch(ClassNotFoundException e) {
            print("Couldn't find Gum");
        }
        print("After Class.forName('Gum')");
        new Cookie();
        print("After creating Cookie");
    }
} /* Output:
      inside main
      Loading Candy
      After creating Candy
      Loading Gum
      After Class.forName("Gum")
      Loading Cookie
      After creating Cookie
   *///:~
```

Jede der drei Klassen `Candy`, `Gum` und `Cookie` definiert einen statischen Block, der beim erstmaligen Laden der Klasse verarbeitet wird. Jeder Block gibt eine Meldung aus, um anzuzeigen, wann die Klasse geladen wird. In der `main()`-Methode wechseln sich Objekterzeugungen und Ausgabeanweisungen ab, damit Sie den Zeitpunkt des Ladevorgangs besser erkennen können.

[18] An der Ausgabe können Sie erkennen, daß ein Klassenobjekt nur bei Bedarf geladen und ein statischer Initialisierungsblock nach dem Laden der Klasse verarbeitet wird.

[19] Die folgende Zeile ist besonders interessant:

```
Class.forName("Gum");
```

Alle Klassenobjekte gehören der Klasse `Class` an. Ein Klassenobjekt ist ein gewöhnliches Objekt, das heißt Sie können eine Referenz auf das Klassenobjekt anfordern und mit dieser Referenz arbeiten (der Klassenlader tut nichts anderes). Die statische `Class`-Methode `forName()` ist eine Möglichkeit, um eine Referenz auf das Klassenobjekt zu bekommen und erwartet ein `String`-Objekt, welches den Namen einer Klasse als Zeichenkette repräsentiert (achten Sie auf korrekte Buchstabierung und auf Groß-/Kleinschreibung). Die Methode gibt eine Referenz vom Typ `Class` zurück, die im obigen Beispiel nicht beachtet wird. Die `forName()`-Methode wird wegen ihres Seiteneffektes aufgerufen, nämlich um die Klasse `Gum` zu laden, falls sie noch nicht geladen wurde. Im Rahmen des Ladevorgangs wird der statische Block der Klasse `Gum` verarbeitet.

[20] Die statische `Class`-Methode ruft eine Ausnahme vom Typ `ClassNotFoundException` hervor, wenn sie die angeforderte Klasse nicht finden kann. Im obigen Beispiel genügt es, eine Meldung auszugeben und die Programmausführung fortzusetzen. Bei einem anspruchsvolleren Programm würden Sie eventuell versuchen, das Problem im Ereignisbehandler zu lösen.

[21] Wenn Sie Typinformationen zur Laufzeit auswerten möchten, müssen Sie stets zuerst eine Referenz auf das entsprechende Klassenobjekt anfordern. Die statische `forName()`-Methode bietet hierfür eine bequeme Möglichkeit, da Sie kein Objekt des entsprechenden Typs brauchen, um eine Referenz auf das erforderliche Klassenobjekt zu bekommen. Wenn Sie bereits ein Objekt der gewünschten Klasse zur Verfügung haben, können Sie mit Hilfe der von `Object` vererbten Methode `getClass()` eine Referenz auf das Klassenobjekt dieses Typs anfordern. Die `getClass()`-Methode gibt die Referenz auf das Klassenobjekt des tatsächlichen Typs des Objektes zurück. Die Klasse `Class` hat noch zahlreiche andere interessante Methoden. Das folgende Beispiel zeigt einige davon:

```
//: typeinfo/toys/ToyTest.java
// Testing class Class.
package typeinfo.toys;
import static net.mindview.util.Print.*;

interface HasBatteries {}
interface Waterproof {}
interface Shoots {}

class Toy {
    // Comment out the following default constructor
    // to see NoSuchMethodError from (*1*)
    Toy() {}
    Toy(int i) {}
}

class FancyToy extends Toy implements HasBatteries, Waterproof, Shoots {
    FancyToy() { super(1); }
}

public class ToyTest {
    static void printInfo(Class cc) {
        print("Class name: " + cc.getName() +
              " is interface? [" + cc.isInterface() + "]");
        print("Simple name: " + cc.getSimpleName());
        print("Canonical name : " + cc.getCanonicalName());
    }
    public static void main(String[] args) {
        Class c = null;
        try {
            c = Class.forName("typeinfo.toys.FancyToy");
        } catch(ClassNotFoundException e) {
            print("Can't find FancyToy");
            System.exit(1);
```

```
        }
        printInfo(c);
        for(Class face : c.getInterfaces())
            printInfo(face);
        Class up = c.getSuperclass();
        Object obj = null;
        try {
            // Requires default constructor:
            obj = up.newInstance();
        } catch(InstantiationException e) {
            print("Cannot instantiate");
            System.exit(1);
        } catch(IllegalAccessException e) {
            print("Cannot access");
            System.exit(1);
        }
        printInfo(obj.getClass());
    }
} /* Output:
      Class name: typeinfo.toys.FancyToy is interface? [false]
      Simple name: FancyToy
      Canonical name : typeinfo.toys.FancyToy
      Class name: typeinfo.toys.HasBatteries is interface? [true]
      Simple name: HasBatteries
      Canonical name : typeinfo.toys.HasBatteries
      Class name: typeinfo.toys.Waterproof is interface? [true]
      Simple name: Waterproof
      Canonical name : typeinfo.toys.Waterproof
      Class name: typeinfo.toys.Shoots is interface? [true]
      Simple name: Shoots
      Canonical name : typeinfo.toys.Shoots
      Class name: typeinfo.toys.Toy is interface? [false]
      Simple name: Toy
      Canonical name : typeinfo.toys.Toy
   *///:~
```

Die Klasse `FancyToy` ist von `Toy` abgeleitet und implementiert die Interfaces *`HasBatteries`*, *`Waterproof`* und *`Shoots`*. Die `main()`-Methode legt eine Referenzvariable vom Typ `Class` an und initialisiert sie in einem `try`-Block per `forName()` mit der Referenz auf das Klassenobjekt der Klasse `FancyToy`. Beachten Sie, daß Sie `forName()` den vollqualifizierten Namen der Klasse (mit Packagenamen) übergeben müssen.

[22] Die Methode `printInfo()` ruft die `getName()`-Methode auf, um den vollqualifizierten Klassennamen abzufragen. Die seit Version 5 der Java Standard Edition (SE 5) in der Klasse `Class` definierten Methoden `getSimpleName()` und `getCanonicalName()` geben den Klassennamen ohne Packagenamen beziehungsweise den vollqualifizierten Klassennamen zurück. Die Methode `isInterface()` gibt an, ob ihr Klassenobjekt ein Interface repräsentiert. Das Klassenobjekt liefert fast alle Informationen über einen Typ.

[23] Die `Class`-Methode `getInterfaces()` gibt ein Array von Klassenobjekten zurück, welche die von dieser Klasse implementierten (beziehungsweise von diesem Interface erweiterten) Interfaces repräsentieren.

[24] Die `Class`-Methode `getSuperclass()` gibt die Referenz auf das Klassenobjekt der unmittelbaren Basisklasse zurück, die weitere Abfragen ermöglicht. Sie können also die gesamte Klassenhierarchie eines Objektes zur Laufzeit auswerten.

[25] Die `Class`-Methode `newInstance()` implementiert einen „virtuellen Konstruktor“, über den Sie ein Objekt korrekt erzeugen können, ohne seinen exakten Typ zu kennen. Im obigen Beispiel ist `up` lediglich eine Referenzvariable vom Typ `Class` ohne zusätzliche, zur Übersetzungszeit verfügbare Informationen. Wenn Sie ein neues Objekt erzeugen, bekommen Sie eine Referenz vom Typ `Object` zurück. Die Referenz verweist aber auf ein Objekt vom Typ `Toy`. Bevor Sie eine andere, als die von `Object` vererbten Methoden aufrufen können, müssen Sie natürlich eine Typumwandlung durchführen. Außerdem muß die Klasse eines per `newInstance()` erzeugten Objektes einen Standardkonstruktor definieren. Sie lernen ~~später in diesem Kapitel~~, wie Sie mit Hilfe des Reflexionsmechanismus jeden Konstruktor einer Klasse zum Erzeugen von Objekten dynamisch aufrufen können.

Übungsaufgabe 1: (1) Kommentieren Sie im Beispiel *ToyTest.java* (Seite 440) den Standardkonstruktor aus und erläutern Sie was geschieht. ■

Übungsaufgabe 2: (2) Legen Sie im Beispiel *ToyTest.java* (Seite 440) ein weiteres Interface an und verifizieren Sie, daß es richtig erkannt und angezeigt wird. ■

Übungsaufgabe 3: (2) Legen Sie im Beispiel *Shapes.java* (Seite 436) eine Klasse `Rhomboid` an. Erzeugen Sie ein `Rhomboid`-Objekt, wandeln Sie die Objektreferenz aufwärts in `Shape` um und anschießend abwärts wieder in `Rhomboid` zurück. Versuchen Sie, die `Rhomboid`-Referenz in den Typ `Circle` umzuwandeln und beobachten Sie was geschieht. ■

Übungsaufgabe 4: (2) Ändern Sie Übungsaufgabe 3, so daß das Programm vor der Typumwandlung eine Typprüfung per `instanceof`-Operator ausführt. ■

Übungsaufgabe 5: (3) Implementieren Sie im Beispiel *Shapes.java* (Seite 436) eine `rotate(Shape)`-Methode, die prüft, ob Sie ein `Circle`-Objekt drehen soll und die Drehung in diesem Fall überspringt. ■

Übungsaufgabe 6: (4) Ändern Sie das Beispiel *Shapes.java* (Seite 436), so daß alle Figuren eines bestimmten Typs gekennzeichnet werden können (in dem ein Schalter gesetzt wird). Die `toString()`-Methode jeder von `Shape` abgeleiteten Klasse gibt an, ob die Figur gekennzeichnet ist. ■

Übungsaufgabe 7: (3) Ändern Sie das Beispiel *SweetShop.java* (Seite 439) so daß der Typ der erzeugten Objekt per Kommandozeile angegeben werden kann. Wenn Sie das Programm beispielsweise per `java SweetShop Candy` aufrufen, wird nur das `Candy`-Objekt erzeugt. Beachten Sie, daß Sie per Kommandozeile steuern können, welche Klassenobjekte geladen werden. ■

Übungsaufgabe 8: (5) Schreiben Sie eine Methode, die ein Objekt erwartet und rekursiv alle Klassen in der Verbungslinie dieses Objektes ausgibt. ■

Übungsaufgabe 9: (5) Ändern Sie Übungsaufgabe 8, so daß die Methode mit Hilfe der `Class`-Methode `getDeclaredFields()` auch die in jeder Klasse definierten Felder ausgibt. ■

Übungsaufgabe 10: (3) Schreiben Sie ein Programm, um zu ermitteln, ob ein `char`-Array von primitivem Typ oder ein echtes Objekt ist. ■

15.2.1 Klassenliterale

[26] Java bietet noch eine zweite Möglichkeit, um die Referenz auf das Klassenobjekt zu bekommen: das *Klassenliteral*. Das Klassenliteral der Klasse `FancyToy` aus dem Beispiel *ToyTest.java* auf Seite 440 lautet:

```
FancyToy.class;
```

Diese Schreibweise ist nicht nur einfacher, sondern auch sicherer, da sie zur Übersetzungszeit geprüft wird (und daher nicht in einem `try`-Block gesetzt zu werden braucht). Das Klassenliteral ist außerdem effizienter, da der Aufruf der `forName()`-Methode wegfällt.

[27–28] Die Klassenliteralsyntax funktioniert bei gewöhnlichen Klassen, Interfaces, Arrays und primitiven Typen. Darüber hinaus hat jede der Wrapperklassen der primitiven Typen ein Feld namens `TYPE`, welches eine Referenz auf das Klassenobjekt des zugehörigen primitiven Typs beinhaltet. Äquivalent sind:

- `boolean.class` und `Boolean.TYPE`
- `char.class` und `Char.TYPE`
- `byte.class` und `Byte.TYPE`
- `short.class` und `Short.TYPE`
- `int.class` und `Integer.TYPE`
- `long.class` und `Long.TYPE`
- `float.class` und `Float.TYPE`
- `double.class` und `Double.TYPE`
- `void.class` und `Void.TYPE`

Ich bevorzuge die `.class`-Notation, da sie mit den gewöhnlichen Klassenliteralen im Einklang ist.

[29] Beachten Sie, daß das Klassenobjekt beim Anfordern einer Referenz über ein Klassenliteral nicht automatisch initialisiert wird. Die Vorbereitung einer Klasse besteht aus drei Schritten:

- Das *Laden der Klasse* wird vom Klassenlader bewerkstelligt. Der Klassenlader sucht den Bytecode (gewöhnlich, aber nicht notwendig auf Ihrer Festplatte und bezüglich Ihres Klassenpfades) und erzeugt daraus ein Klassenobjekt.
- Während der *Linking-Phase* wird der Bytecode der Klasse verifiziert, Arbeitsspeicher für die statischen Felder allokiert und erforderlichenfalls alle Referenzen auf weitere Klassen aufgelöst.
- Beim *Initialisieren der Klasse* wird eine eventuell vorhandene Basisklasse zuerst initialisiert. Anschließend werden statische Initialisierungsausdrücke und -blöcke verarbeitet.

[30] Die Initialisierung wird solange verzögert, bis die erste statische Methode (der Konstruktor ist implizit statisch) aufgerufen oder das erste statische nicht konstante Feld referenziert wird:

```
//: typeinfo/ClassInitialization.java
import java.util.*;

class Initable {
    static final int staticFinal = 47;
    static final int staticFinal2 = ClassInitialization.rand.nextInt(1000);
    static {
        System.out.println("Initializing Initable");
    }
}

class Initable2 {
    static int staticNonFinal = 147;
    static {
        System.out.println("Initializing Initable2");
    }
}
```

```
class Initable3 {
    static int staticNonFinal = 74;
    static {
        System.out.println("Initializing Initable3");
    }
}

public class ClassInitialization {
    public static Random rand = new Random(47);
    public static void main(String[] args) throws Exception {
        Class initable = Initable.class;
        System.out.println("After creating Initable ref");
        // Does not trigger initialization:
        System.out.println(Initable.staticFinal);
        // Does trigger initialization:
        System.out.println(Initable.staticFinal2);
        // Does trigger initialization:
        System.out.println(Initable2.staticNonFinal);
        Class initable3 = Class.forName("Initable3");
        System.out.println("After creating Initable3 ref");
        System.out.println(Initable3.staticNonFinal);
    }
} /* Output:
      After creating Initable ref
      47
      Initializing Initable
      258
      Initializing Initable2
      147
      Initializing Initable3
      After creating Initable3 ref
      74
   *///:~
```

Die Initialisierung ist gewissermaßen „so faul wie möglich“. An der Deklaration der Referenzvariablen `initable` können Sie beobachten, daß die alleinige Verwendung der `.class`-Syntax, um die Referenz auf ein Klassenobjekt anzufordern, noch keine Initialisierung bewirkt. Die `Class`-Methode `forName()` initialisiert die Klasse dagegen unmittelbar, bevor sie die Referenz auf das Klassenobjekt zurückgibt (siehe Deklaration der Referenzvariablen `initable3`).

[31] Definiert ein statisches finales Feld, wie `staticFinal` in der Klasse `Initable`, eine Konstante zur Übersetzungszeit, so kann der Feldwert ohne Initialisierung der Klasse `Initable` abgefragt werden. Die alleinige Deklaration eines Feldes als statisch und final garantiert dieses Verhalten allerdings nicht: Das Abfragen des `staticFinal2`-Feldes der Klasse `Initable2` erzwingt die Initialisierung der Klasse, da der Feldinhalt keine Konstante zur Übersetzungszeit sein kann.

[32] Ist ein statisches Feld nicht zugleich final, so erfordert ein Zugriff stets Linking (Allokieren von Arbeitsspeicher für das Feld) und Initialisierung (des allokierten Arbeitsspeichers), bevor das Feld abgefragt werden kann (siehe `staticNonFinal`-Feld der Klasse `Initable2`).

15.2.2 Generische Referenzvariablen für Klassenobjekte

[33] Eine Referenzvariable vom Typ `Class` verweist auf ein Klassenobjekt, welches zum Erzeugen von Objekten dient und den Bytecode sämtlicher Methoden dieser Objekte enthält. Das Klassenobjekt enthält auch die statischen Komponenten der Klasse. Eine Referenzvariable vom Typ `Class` zeigt den exakten Typ ihres Verweisziels an: Ein Objekt vom Typ `Class`.

[34] Die Designer der SE 5 haben die günstige Gelegenheit erkannt, Referenzvariablen dadurch etwas genauer spezifizieren zu können, daß Sie den Typ des referenzierten Klassenobjektes, auf den eine Referenzvariable vom Typ `Class` verweist, mittels generischer Syntax einschränken können. Im folgenden Beispiel sind beide Syntaxen korrekt:

```
//: typeinfo/GenericClassReferences.java
public class GenericClassReferences {
    public static void main(String[] args) {
        Class intClass = int.class;
        Class<Integer> genericIntClass = int.class;
        genericIntClass = Integer.class; // Same thing
        intClass = double.class;
        // genericIntClass = double.class; // Illegal
    }
} ///:~
```

Die gewöhnliche Referenzvariable bewirkt keine Warnung, wenn sie einem anderen Klassenobjekt zugewiesen wird, während die generische Referenzvariable nur einem Klassenobjekt des deklarierten Typs zugewiesen werden kann. Durch die generische Syntax ermöglichen Sie dem Compiler, eine zusätzliche Typprüfung zu erzwingen.

[35] Ist es möglich, die Einschränkung ein wenig zu lockern? Auf den ersten Blick sollte die folgende Zuweisung erlaubt sein:

```
Class<Number> genericNumberClass = int.class;
```

Die Zuweisung wirkt sinnvoll, da die Klasse `Integer` von `Number` abgeleitet ist, scheitert aber, weil der Typ des Klassenobjektes von `Integer` kein Untertyp des Klassenobjektes von `Number` ist (diese Unterscheidung mag subtil erscheinen; wir betrachten den Unterschied in Kapitel 16 genauer).

[36] Zur Lockerung der Einschränkung generischer Referenzvariablen vom Typ `Class` müssen wir das Platzhalterzeichen (`?`) einführen, das zu den generischen Typen von Java gehört. Das Fragezeichen bedeutet „beliebiger Typ“. Die folgende Änderung in der Deklaration der gewöhnlichen Referenzvariablen aus dem vorigen Beispiel liefert dasselbe Ergebnis:

```
//: typeinfo/WildcardClassReferences.java
public class WildcardClassReferences {
    public static void main(String[] args) {
        Class<?> intClass = int.class;
        intClass = double.class;
    }
} ///:~
```

In der SE 5 wird die Notation `Class<?>` gegenüber `Class` bevorzugt, obwohl beide Schreibweisen äquivalent sind und die nicht-generische Variante keine Compilerwarnung auslöst. Der Vorteil an der generischen Variante `Class<?>` besteht darin, daß Sie anzeigen, daß Sie die unspezifierte Syntax bewußt und nicht versehentlich gewählt oder die generische Schreibweise ignoriert haben.

[37] Sie deklarieren eine Referenzvariable vom Typ `Class`, die auf einen Typ oder einen von diesem abgeleiteten Typ eingeschränkt ist, indem Sie das Platzhalterzeichen mit dem `extends`-Schlüsselwort zu einem *beschränkten Typparameter* kombinieren. Anstelle von `Class<Number>` lautet die richtige Deklaration `Class<? extends Number>`:

```
//: typeinfo/BoundedClassReferences.java
public class BoundedClassReferences {
    public static void main(String[] args) {
        Class<? extends Number> bounded = int.class;
        bounded = double.class;
```

```
        bounded = Number.class;
        // Or anything else derived from Number.
    }
} ///:~
```

Die Deklaration einer Referenzvariablen vom Typ `Class` wurde nur um die generische Syntax erweitert, um die Typprüfung zur Übersetzungszeit zu nutzen, so daß Sie etwas früher auf Fehler aufmerksam werden. Eigentlich können Sie auch mit gewöhnlichen Referenzvariablen vom Typ `Class` nicht auf Abwege geraten, aber wenn sich ein Fehler einschleicht, zeigt er sich lästigerweise erst zur Laufzeit.

[38] Das folgende Beispiel verwendet die generische Syntax. Es speichert die Referenz auf ein Klassenobjekt und erzeugt später einen *`List`*-Container dessen Elemente per `newInstance()` erzeugt werden:

```
//: typeinfo/FilledList.java
import java.util.*;

class CountedInteger {
    private static long counter;
    private final long id = counter++;
    public String toString() { return Long.toString(id); }
}

public class FilledList<T> {
    private Class<T> type;
    public FilledList(Class<T> type) { this.type = type; }
    public List<T> create(int nElements) {
        List<T> result = new ArrayList<T>();
        try {
            for(int i = 0; i < nElements; i++)
                result.add(type.newInstance());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        return result;
    }
    public static void main(String[] args) {
        FilledList<CountedInteger> fl =
            new FilledList<CountedInteger>(CountedInteger.class);
        System.out.println(fl.create(15));
    }
} /* Output:
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
   *///:~
```

Beachten Sie, daß die Klasse `FilledList` voraussetzt, daß jeder Parametertyp einen Standardkonstruktor besitzt (einen parameterlosen Konstruktor) und andernfalls eine Ausnahme ausgeworfen wird. Der Compiler gibt beim Übersetzen dieses Programms keine Warnung aus.

[39] Die generische Syntax bei Referenzvariablen vom Typ `Class` hat eine interessante Auswirkung: Die `newInstance()`-Methode gibt den den exakten Objekttyp zurück, nicht `Object` wie im Beispiel *ToyTest.java* auf Seite 440. This is somewhat limited:

```
//: typeinfo/toys/GenericToyTest.java
// Testing class Class.
package typeinfo.toys;

public class GenericToyTest {
```

```
    public static void main(String[] args) throws Exception {
        Class<FancyToy> ftClass = FancyToy.class;
        // Produces exact type:
        FancyToy fancyToy = ftClass.newInstance();
        Class<? super FancyToy> up = ftClass.getSuperclass();
        // This won't compile:
        // Class<Toy> up2 = ftClass.getSuperclass();
        // Only produces Object:
        Object obj = up.newInstance();
    }
} ///:~
```

Beim Anfordern der Referenz auf das Klassenobjekt der Basisklasse gestattet der Compiler nur die Deklaration `Class<? super FancyToy>`, das heißt daß die Basisklasse eine Oberklasse von `FancyToy` ist, nicht aber `Class<Toy>`. Letzteres ist ein wenig sonderbar, da die `getSuperClass()`-Methode das Klassenobjekt der Basis*klasse* (nicht Interface) zurückgibt und dem Compiler die Klasse bereits zur Übersetzungszeit bekannt ist (hier `Toy.class` und keine beliebige Basisklasse von `FancyToy`). Die über die Referenzvariable `up` aufgerufene `newInstance()`-Methode gibt aufgrund der Unsicherheit jedenfalls keinen exakten Typ zurück, sondern `Object`.

15.2.3 Neue Typumwandlungssyntax: Die Class-Methode cast()

[40] Die SE 5 stellt eine neue Syntax für Typumwandlungen zur Verfügung, die beim Bewerten von Referenzvariablen des Typs `Class` vorkommt, nämlich die `Class`-Methode `cast()`:

```
//: typeinfo/ClassCasts.java
class Building {}
class House extends Building {}

public class ClassCasts {
    public static void main(String[] args) {
        Building b = new House();
        Class<House> houseType = House.class;
        House h = houseType.cast(b);
        h = (House)b; // ... or just do this.
    }
} ///:~
```

Die `cast()`-Methode erwartet ein Objekt als Argument und wandelt es in den Parametertyp der `Class`-Referenzvariablen um. In diesem kleinen Beispiel wirkt die neue Syntax, verglichen mit der letzten Zeile in der `main()`-Methode, welche dieselbe Wirkung hat, wie zusätzliche Arbeit.

[41] Die neue Typumwandlungssyntax ist in Situationen nützlich, in denen Sie keine gewöhnliche Typumwandlung ausführen können. Dieser Fall tritt im allgemeinen ein, wenn Sie generischen Quelltext entwickeln (siehe Kapitel 16) und die Referenz auf ein Klassenobjekt speichern, um damit später eine Typumwandlung zu bewerkstelligen. Die `cast()`-Methode wird nur selten angewendet. Ich habe in der gesamten Bibliothek der SE 5 nur ein einziges Vorkommen gefunden (in der Klasse `com.sun.mirror.util.DeclarationFilter`) Eine weitere Neuerung hat überhaupt keine Anwendung in der Bibliothek der SE 5: Die `Class`-Methode `asSubclass()`. Die Methode ermöglicht die Umwandlung des Klassenobjektes in einen spezialisierteren Typ.

15.3 Typprüfung vor Typumwandlung

[42] Sie haben bereits zwei Möglichkeiten kennengelernt, um Typinformationen zur Laufzeit auszuwerten:

- Die klassische Typumwandlung, zum Beispiel `(Shape)`, nutzt RTTI, um sicherzustellen, daß die Typumwandlung korrekt ist.
- Das Klassenobjekt repräsentiert den Typ eines Objektes der entsprechenden Klasse und gestattet das Abfragen nützlicher Informationen zur Laufzeit.

Bei C++ bewirkt die klassische Typumwandlung `(Shape)` keine Auswertung von Typinformationen zur Laufzeit, sondern weist den Compiler an, das Objekt als dem angegebenen Typ zugehörig zu behandeln. Bei Java wird dagegen eine Typprüfung vorgeschaltet und eine solche Umwandlung als *typsichere abwärts gerichtete Typumwandlung* bezeichnet. Die Bezeichnung „abwärts gerichtete Typumwandlung“ (*downcast*) stammt von der „historisch gewachsenen“ Darstellungsweise von Klassenhierarchiediagrammen. Die Umwandlung von `Circle` nach `Shape` ist aufwärts gerichtet (*upcast*), die Umwandlung von `Shape` nach `Circle` abwärts. Der Compiler erlaubt die implizite aufwärtsgerichtete Typumwandlung bei Zuweisungen, ohne ausdrückliche Umwandlungssyntax, da bekannt ist, daß ein `Circle`-Objekt auch der Klasse `Shape` angehört. Andererseits *kann* der Compiler nicht „wissen“, welcher Typ sich „hinter“ einer Referenz vom Typ `Shape` verbirgt. Es könnte ein `Shape`-Objekt selbst sein, aber auch ein von `Shape` abgeleiteter Typ wie `Circle`, `Square` oder `Triangle`. Der Compiler „sieht“ zur Übersetzungszeit nur den Typ `Shape` und gestattet daher keine abwärtsgerichtete Typumwandlung ohne explizite Syntax, mit der Sie dem Compiler mitteilen, daß Sie über zusätzliche Informationen verfügen und wissen, daß das referenzierte Objekt einem bestimmten Typ angehört. (Der Compiler prüft, ob die anforderte Typumwandlung sinnvoll ist, erlaubt also keine abwärts gerichtete Typumwandlung in einen Typ, der kein abgeleiteter Typ ist.)

[43] Es gibt noch eine dritte Variante, um Typinformationen zur Laufzeit auszuwerten. Der Operator `instanceof` gibt an, ob ein Objekt einem bestimmten Typ angehört. Der Operator gibt einen `boolean`-Wert zurück, so daß er sich in einer Abfrage wie folgt einsetzen läßt:

```
if (x instanceof Dog)
    ((Dog) x).bark();
```

Die `if`-Anweisung prüft, ob das von `x` referenzierte Objekt der Klasse `Dog` angehört, *bevor* die Typumwandlung vollzogen wird. Wenn Sie keine Informationen über den Typ eines Objektes haben, ist es wichtig, vor einer abwärts gerichteten Typumwandlung die Prüfung mit dem `instanceof`-Operator durchzuführen. Andernfalls kann eine Ausnahme vom Typ `ClassCastException` hervorgerufen werden.

[44] In der Regel suchen Sie nach Objekten eines bestimmten Typs (beispielsweise um Dreiecke zu färben), können aber auch einfach alle Objekte mit Hilfe des `instanceof`-Operators durchgehen. Stellen Sie beispielsweise eine Familie von Klassen vor, die Haustiere beschreiben (und ihre Besitzer; diese Eigenschaft ist für Beispiel ~~///~~ auf Seite **??** wichtig). Jedes Individuum (`Individual`) in dieser Hierarchie hat einen Kennzahl (`id`-Feld) und einen Namen (`name`-Feld). Die folgenden Klassen sind von der Klasse `Individual` abgeleitet. Die Klasse `Individual` hat einige komplizierte Eigenschaften und wird daher erst in Unterabschnitt 18.9.3 vorgestellt und erläutert. Sie werden sehen, daß Sie den Quelltext von `Individual` an dieser Stelle noch nicht wirklich brauchen. Es genügt, zu wissen, daß Sie ein `Individual`-Objekt mit oder ohne Namen erzeugen können und das jedes solche Objekt eine `id()`-Methode besitzt, welche die eindeutige Kennzahl (Objektzähler) zurückgibt. Außerdem definiert die Klasse `Individual` eine `toString()`-Methode, die den einfachen (nicht voll qualifizierten) Namen des Typs zurückgibt, falls das `name`-Feld des Objektes nicht bewertet ist.

[45] Die folgenden Klassen sind von `Individual` abgeleitet:

```
//: typeinfo/pets/Person.java
package typeinfo.pets;

public class Person extends Individual {
    public Person(String name) { super(name); }
} ///:~

//: typeinfo/pets/Pet.java
package typeinfo.pets;

public class Pet extends Individual {
    public Pet(String name) { super(name); }
    public Pet() { super(); }
} ///:~

//: typeinfo/pets/Dog.java
package typeinfo.pets;

public class Dog extends Pet {
    public Dog(String name) { super(name); }
    public Dog() { super(); }
} ///:~

//: typeinfo/pets/Mutt.java
package typeinfo.pets;

public class Mutt extends Dog {
    public Mutt(String name) { super(name); }
    public Mutt() { super(); }
} ///:~

//: typeinfo/pets/Pug.java
package typeinfo.pets;

public class Pug extends Dog {
    public Pug(String name) { super(name); }
    public Pug() { super(); }
} ///:~

//: typeinfo/pets/Cat.java
package typeinfo.pets;

public class Cat extends Pet {
    public Cat(String name) { super(name); }
    public Cat() { super(); }
} ///:~

//: typeinfo/pets/EgyptianMau.java
package typeinfo.pets;

public class EgyptianMau extends Cat {
    public EgyptianMau(String name) { super(name); }
    public EgyptianMau() { super(); }
} ///:~

//: typeinfo/pets/Manx.java
package typeinfo.pets;

public class Manx extends Cat {
    public Manx(String name) { super(name); }
```

```
    public Manx() { super(); }
} ///:~

//: typeinfo/pets/Cymric.java
package typeinfo.pets;

public class Cymric extends Manx {
    public Cymric(String name) { super(name); }
    public Cymric() { super(); }
} ///:~

//: typeinfo/pets/Rodent.java
package typeinfo.pets;

public class Rodent extends Pet {
    public Rodent(String name) { super(name); }
    public Rodent() { super(); }
} ///:~

//: typeinfo/pets/Rat.java
package typeinfo.pets;

public class Rat extends Rodent {
    public Rat(String name) { super(name); }
    public Rat() { super(); }
} ///:~

//: typeinfo/pets/Mouse.java
package typeinfo.pets;

public class Mouse extends Rodent {
    public Mouse(String name) { super(name); }
    public Mouse() { super(); }
} ///:~

//: typeinfo/pets/Hamster.java
package typeinfo.pets;

public class Hamster extends Rodent {
    public Hamster(String name) { super(name); }
    public Hamster() { super(); }
} ///:~
```

[46] Als nächstes brauchen wir eine Möglichkeit, um verschiedene Haustiere von zufällig bestimmter Tierart zu erzeugen, insbesondere Arrays und *List*-Container mit Haustieren. Wir definieren diese Hilfsklasse als abstrakte Klasse, um die Weiterentwicklung in verschiedenen Implementierungen zu ermöglichen:

```
//: typeinfo/pets/PetCreator.java
// Creates random sequences of Pets.
package typeinfo.pets;
import java.util.*;

public abstract class PetCreator {
    private Random rand = new Random(47);
    // The List of the different types of Pet to create:
    public abstract List<Class<? extends Pet>> types();
    public Pet randomPet() { // Create one random Pet
        int n = rand.nextInt(types().size());
        try {
```

```
            return types().get(n).newInstance();
        } catch(InstantiationException e) {
            throw new RuntimeException(e);
        } catch(IllegalAccessException e) {
            throw new RuntimeException(e);
        }
    }
    public Pet[] createArray(int size) {
        Pet[] result = new Pet[size];
        for(int i = 0; i < size; i++)
            result[i] = randomPet();
        return result;
    }
    public ArrayList<Pet> arrayList(int size) {
        ArrayList<Pet> result = new ArrayList<Pet>();
        Collections.addAll(result, createArray(size));
        return result;
    }
} ///:~
```

Die abstrakte Methode `types()` verlegt das Erzeugen eines *`List`*-Containers mit Klassenobjekten in die von `PetCreator` abgeleiteten Klassen (eine Variante des Entwurfsmusters *Templatemethod*). Beachten Sie, daß der Parametertyp des Klassenobjektes definiert ist, als jede von `Pet` abgeleitete Klasse. Die `newInstance()`-Methode gibt daher den Typ `Pet` zurück, ohne eine Typumwandlung zu benötigen. Die `randomPet()`-Methode wählt ein zufällig bestimmtes Element aus dem *`List`*-Container und verwendet das gewählte Klassenobjekt, um per `newInstance()` ein neues Objekt der entsprechenden Klasse zu erzeugen. Die `createArray()`-Methode ruft `randomPet()` auf, um ein Array zu füllen und die `arrayList()`-Methode stützt sich wiederum auf `createArray()`.

[47] Die `newInstance()`-Methode kann zwei Typen von Ausnahmen hervorrufen, die Sie in den `catch`-Klauseln nach dem `try`-Block sehen. Die Namen der Ausnahmen sind auch hier eine relativ hilfreiche Erklärung der Ursache (`IllegalAccessException` bezieht sich auf eine Verletzung des Sicherheitsmechanismus von Java; hier auf einen als privat definierten Standardkonstruktor).

[48] Wenn Sie eine Klasse von `PetCreator` ableiten, müssen Sie lediglich den *`List`*-Container mit den Klassenobjekten der verschiedenen Tierarten erzeugen, der benötigt wird, um über `randomPet()` und die übrigen Methoden `Pet`-Objekte zu generieren. Die `types()`-Methode gibt in der Regel die Referenz auf einen statischen *`List`*-Container zurück. Die folgende Implementierung verwendet die `forName()`-Methode:

```
//: typeinfo/pets/ForNameCreator.java
package typeinfo.pets;
import java.util.*;

public class ForNameCreator extends PetCreator {
    private static List<Class<? extends Pet>> types =
        new ArrayList<Class<? extends Pet>>();
    // Types that you want to be randomly created:
    private static String[] typeNames = {
        ''typeinfo.pets.Mutt'',
        ''typeinfo.pets.Pug'',
        ''typeinfo.pets.EgyptianMau'',
        ''typeinfo.pets.Manx'',
        ''typeinfo.pets.Cymric'',
        ''typeinfo.pets.Rat'',
        ''typeinfo.pets.Mouse'',
        ''typeinfo.pets.Hamster''
```

```
    };
    @SuppressWarnings("unchecked")
    private static void loader() {
        try {
            for(String name : typeNames)
                types.add((Class<? extends Pet>) Class.forName(name));
        } catch(ClassNotFoundException e) {
            throw new RuntimeException(e);
        }
    }
    static { loader(); }
    public List<Class<? extends Pet>> types() {return types;}
} ///:~
```

Die `loader()`-Methode füllt den von `types` referenzierten *`List`*-Container mit Hilfe der `Class`-Methode `forName()` mit Klassenobjekten. Die `forName()`-Methode kann eine Ausnahme vom Typ `ClassNotFoundException` hervorrufen, da das als Argument übergebene `String`-Objekt zur Übersetzungszeit nicht validiert werden kann. Da die `Pet`-Objekte im Package `typeinfo.pets` liegen, müssen die Klassennamen mit Package angegeben werden.

[49] *In order to produce a typed List of Class objects, a cast is required, which produces a compile-time warning.* Die `loader()`-Methode wurde separat definiert und in einen statischen Initialisierungsblock eingesetzt, da die Annotation `@SuppressWarnings` nicht unmittelbar vor einem statischen Initialisierungsblock platziert werden kann.

[50] Wir brauchen noch ein Hilfsmittel, um die Anzahlen der Objekte der unterschiedlichen Tierarten zu erfassen. Ein *`Map`*-Container eignet sich zu diesem Zweck hervorragend. Wir verwenden die Tierart als Schlüssel und die Häufigkeit als Wert. Damit können Sie zum Beispiel abfragen, wieviele Hamster vorhanden sind. Der `instanceof`-Operator hilft beim Zählen der Haustiere:

```
//: typeinfo/PetCount.java
// Using instanceof.
import typeinfo.pets.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class PetCount {
    static class PetCounter extends HashMap<String,Integer> {
        public void count(String type) {
            Integer quantity = get(type);
            if(quantity == null)
                put(type, 1);
            else
                put(type, quantity + 1);
        }
    }
    public static void
        countPets(PetCreator creator) {
        PetCounter counter= new PetCounter();
        for(Pet pet : creator.createArray(20)) {
            // List each individual pet:
            printnb(pet.getClass().getSimpleName() + " ");
            if(pet instanceof Pet)
                counter.count("Pet");
            if(pet instanceof Dog)
                counter.count("Dog");
            if(pet instanceof Mutt)
                counter.count("Mutt");
```

```
            if(pet instanceof Pug)
                counter.count("Pug");
            if(pet instanceof Cat)
                counter.count("Cat");
            if(pet instanceof Manx)
                counter.count("EgyptianMau");
            if(pet instanceof Manx)
                counter.count("Manx");
            if(pet instanceof Manx)
                counter.count("Cymric");
            if(pet instanceof Rodent)
                counter.count("Rodent");
            if(pet instanceof Rat)
                counter.count("Rat");
            if(pet instanceof Mouse)
                counter.count("Mouse");
            if(pet instanceof Hamster)
                counter.count("Hamster");
        }
        // Show the counts:
        print();
        print(counter);
    }
    public static void main(String[] args) {
        countPets(new ForNameCreator());
    }
} /* Output:
      Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau
        Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug Mouse Cymric
      {Pug=3, Cat=9, Hamster=1, Cymric=7, Mouse=2, Mutt=3, Rodent=5,
        Pet=20, Manx=7, EgyptianMau=7, Dog=6, Rat=2}
   *///:~
```

Die `countPets()`-Methode füllt mit Hilfe eines `PetCreator`-Objektes ein Array mit Haustieren zufällig gewählten Typs. Danach wird jedes `Pet`-Objekt des Arrays per `instanceof` auf seinen Typ hin geprüft und gezählt.

[51] Der `instanceof`-Operator unterliegt einer kleinlichen Einschränkung: Sie können Referenzen nur mit einem benannten Typ vergleichen, nicht aber mit einem Klassenobjekt. Vielleicht empfinden Sie es im obigen Beispiel (aus meiner Sicht berechtigterweise) als lästig, alle `instanceof`-Ausdrücke ausschreiben zu müssen. Aber es gibt keine Möglichkeit, die `instanceof`-Vergleiche zu automatisieren, etwa indem Sie ein Array von Klassenobjekten anlegen und eine Referenz mit dessen Elementen vergleichen. (Sie lernen in Unterabschnitt 15.3.2 eine dynamische Alternative zu `instanceof` kennen.) Die Einschränkung ist aber nicht so groß, wie Sie eventuell annehmen. Letztendlich werden Sie verstehen, daß eine Vielzahl von `instanceof`-Ausdrücken wahrscheinlich auf Schwächen in Ihrem Design hinweist.

15.3.1 Anwendung von Klassenliteralen

[52] Die Neuimplementierung der Klasse `PetCreator` mit Klassenliteralen ist in vieler Hinsicht klarer:

```
//: typeinfo/pets/LiteralPetCreator.java
// Using class literals.
package typeinfo.pets;
```

```
import java.util.*;

public class LiteralPetCreator extends PetCreator {
    // No try block needed.
    @SuppressWarnings("unchecked")
    public static final List<Class<? extends Pet>> allTypes =
            Collections.unmodifiableList(Arrays.asList(
                Pet.class, Dog.class, Cat.class, Rodent.class,
                Mutt.class, Pug.class, EgyptianMau.class, Manx.class,
                Cymric.class, Rat.class, Mouse.class,Hamster.class));
    // Types for random creation:
    private static final List<Class<? extends Pet>> types =
        allTypes.subList(allTypes.indexOf(Mutt.class), allTypes.size());
    public List<Class<? extends Pet>> types() {
        return types;
    }
    public static void main(String[] args) {
        System.out.println(types);
    }
} /* Output:
      [class typeinfo.pets.Mutt, class typeinfo.pets.Pug,
        class typeinfo.pets.EgyptianMau, class typeinfo.pets.Manx,
        class typeinfo.pets.Cymric, class typeinfo.pets.Rat,
        class typeinfo.pets.Mouse, class typeinfo.pets.Hamster]
   *///:~
```

Für das bevorstehende Beispiel *PetCount3.java* auf Seite 455 brauchen wir einen vorinitialisierten *Map*-Container mit allen Tierarten (nicht nur den zufällig ausgewählten). Daher ist der von `allTypes` referenzierte *List*-Container notwendig. Der von `types` referenzierte *List*-Container enthält eine Teilmenge des Inhalts von `allTypes` (erzeugt mit der *List*-Methode `subList()`). Die Elemente in `types` repräsentieren die exakten Untertypen von `Pet` und werden zum Erzeugen der `Pet`-Objekte verwendet.

[53] Diesesmal ist beim Initialisieren von `types`, im Gegensatz zu der Lösung mit `forName()`, kein `try`-Block erforderlich, da die Elemente zur Übersetzungszeit geprüft werden und somit keine Ausnahmen hervorgerufen werden.

[54] Das Package `typeinfo.pets` enthält nun zwei Implementierungen der abstrakten Klasse `PetCreator`. Die folgende Klasse `Pets` implementiert das Entwurfsmuster *Façade*, um die Klasse `LiteralPetCreator` als Standardimplementierung zu verwenden:

```
//: typeinfo/pets/Pets.java
// Facade to produce a default PetCreator.
package typeinfo.pets;
import java.util.*;

public class Pets {
    public static final PetCreator creator = new LiteralPetCreator();
    public static Pet randomPet() {
        return creator.randomPet();
    }
    public static Pet[] createArray(int size) {
        return creator.createArray(size);
    }
    public static ArrayList<Pet> arrayList(int size) {
        return creator.arrayList(size);
    }
} ///:~
```

Die „Fassade“ liefert außerdem eine Indirektionsebene über den Methoden `randomPet()`, `create-Array()` und `arrayList()`.

[55] Die Klasse `LiteralPetCreator` läßt sich über die *Façade*-Klasse `Pets` mühelos testen, da die statische `PetCount`-Methode `countPets()` ein Argument vom Typ `PetCreator` erwartet:

```
//: typeinfo/PetCount2.java
import typeinfo.pets.*;

public class PetCount2 {
    public static void main(String[] args) {
        PetCount.countPets(Pets.creator);
    }
} /* (Execute to see output) *////:~
```

Die Ausgabe entspricht der Ausgabe des Beispiels *PetCount.java* auf Seite 452.

15.3.2 Ein „dynamischer instanceof-Operator“: Die Class-Methode isInstance()

[56] Die `Class`-Methode `isInstance()` gestattet das dynamische Testen des Objekttyps. Damit lassen sich die lästigen `instanceof`-Anweisungen aus dem Beispiel *PetCount.java* auf Seite 452 entfernen:

```
//: typeinfo/PetCount3.java
// Using isInstance()
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class PetCount3 {
    static class PetCounter extends LinkedHashMap<Class<? extends Pet>,Integer> {
        public PetCounter() {
            super(MapData.map(LiteralPetCreator.allTypes, 0));
        }
        public void count(Pet pet) {
            // Class.isInstance() eliminates instanceofs:
            for(Map.Entry<Class<? extends Pet>,Integer> pair: entrySet())
                if(pair.getKey().isInstance(pet))
                    put(pair.getKey(), pair.getValue() + 1);
        }
        public String toString() {
            StringBuilder result = new StringBuilder(''{'');
            for(Map.Entry<Class<? extends Pet>,Integer> pair: entrySet()) {
                result.append(pair.getKey().getSimpleName());
                result.append(''='');
                result.append(pair.getValue());
                result.append('', '');
            }
            result.delete(result.length()-2, result.length());
            result.append(''}'');
            return result.toString();
        }
    }
    public static void main(String[] args) {
        PetCounter petCount = new PetCounter();
        for(Pet pet : Pets.createArray(20)) {
```

```
                printnb(pet.getClass().getSimpleName() + " ");
                petCount.count(pet);
            }
            print();
            print(petCount);
        }
    } /* Output:
          Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau
            Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug Mouse Cymric
          {Pet=20, Dog=6, Cat=9, Rodent=5, Mutt=3, Pug=3, EgyptianMau=2,
            Manx=7, Cymric=5, Rat=2, Mouse=2, Hamster=1}
       *///:~
```

Die von *Map* abgeleitete innere Containerklasse `PetCounter` wird mit den Tierarten aus dem von `LiteralPetCreator.allTypes` referenzierten *List*-Container vorinitialisiert. Die Hilfsklasse `net.mindview.util.MapData` erwartet ein Argument vom Typ *Iterable* (den von `allTypes` referenzierten *List*-Container) sowie einen konstanten Wert (hier Null) und füllt den *Map*-Container mit Schlüssel/Wert-Paaren, wobei die Tierarten als Schlüssel und die Anzahl Null als Wert eingesetzt werden. Ohne Vorinitialisierung des *Map*-Containers würden nur die zufällig gewählten Tierarten gezählt werden, nicht aber die Basistypen wie `Pet` oder `Cat`.

[57] Durch die `isInstance()`-Methode sind die `instanceof`-Ausdrücke nicht mehr notwendig. Darüberhinaus können Sie neue Tierarten anlegen, indem Sie sie einfach in das von `types` referenzierte Array in der Klasse `LiteralPetCreator` eintragen. Das restliche Programm braucht nicht verändert zu werden (wie etwa bei Verwendung der `instanceof`-Ausdrücke).

[58] Die `toString()`-Methode wurde überschrieben, um die Ausgabe in gut lesbarer Weise zu Formatieren. Die Ausgabe ist an die typische Formatierung angelehnt, die Sie erhalten, wenn Sie einen *Map*-Container über `System.out.println()` ausgeben.

15.3.3 Rekursives Zahlen und die Class-Methode isAssignableFrom()

[59] Der *Map*-Container in der inneren Klasse `PetCount3.PetCounter` im vorigen Unterabschnitt war mit den verschiedenen Tierarten vorinitialisiert. Statt den Container vorzuinitialisieren, können wir mit Hilfe der `Class`-Methode `isAssignableFrom()` eine universelle Hilfsklasse/-methode schreiben, die nicht auf das Zählen von Haustieren beschränkt ist:

```
//: net/mindview/util/TypeCounter.java
// Counts instances of a type family.
package net.mindview.util;
import java.util.*;

public class TypeCounter extends HashMap<Class<?>,Integer>{
    private Class<?> baseType;
    public TypeCounter(Class<?> baseType) {
        this.baseType = baseType;
    }
    public void count(Object obj) {
        Class<?> type = obj.getClass();
        if(!baseType.isAssignableFrom(type))
            throw new RuntimeException(obj + " incorrect type: "
                + type + ", should be type or subtype of " + baseType);
        countClass(type);
    }
    private void countClass(Class<?> type) {
        Integer quantity = get(type);
```

```
            put(type, quantity == null ? 1 : quantity + 1);
            Class<?> superClass = type.getSuperclass();
            if(superClass != null && baseType.isAssignableFrom(superClass))
                countClass(superClass);
        }
        public String toString() {
            StringBuilder result = new StringBuilder("{");
            for(Map.Entry<Class<?>,Integer> pair : entrySet()) {
                result.append(pair.getKey().getSimpleName());
                result.append("=");
                result.append(pair.getValue());
                result.append(", ");
            }
            result.delete(result.length()-2, result.length());
            result.append("}");
            return result.toString();
        }
    } ///:~
```

[60] Die Methode `count()` bestimmt das Klassenobjekt ihres Argumentes und ruft `isAssignableFrom()` auf, um zur Laufzeit zu prüfen, ob das übergebene Objekt zu der betrachteten Hierarchie gehört. Die Methode `countClass()` zählt zuerst das Vorkommen des exakten Typs der Klasse. Then, if baseType is assignable from the superclass, ruft sich `countClass()` rekursiv auf der Basisklasse auf.

```
    //: typeinfo/PetCount4.java
    import typeinfo.pets.*;
    import net.mindview.util.*;
    import static net.mindview.util.Print.*;

    public class PetCount4 {
        public static void main(String[] args) {
            TypeCounter counter = new TypeCounter(Pet.class);
            for(Pet pet : Pets.createArray(20)) {
                printnb(pet.getClass().getSimpleName() + " ");
                counter.count(pet);
            }
            print();
            print(counter);
        }
    } /* Output: (Sample)
          Rat Manx Cymric Mutt Pug Cymric Pug Manx Cymric Rat EgyptianMau
            Hamster EgyptianMau Mutt Mutt Cymric Mouse Pug Mouse Cymric
          {Mouse=2, Dog=6, Manx=7, EgyptianMau=2, Rodent=5, Pug=3, Mutt=3,
            Cymric=5, Cat=9, Hamster=1, Pet=20, Rat=2}
       *///:~
```

Die Ausgabe zeigt, daß beide Basistypen und die exakten Typen gezählt werden.

Übungsaufgabe 11: (2) Legen Sie im Package `typeinfo.pets` eine neue Klasse `Gerbil` an und ändern Sie alle Beispiele in diesem Kapitel, so daß sie mit der neuen Klasse arbeiten. ■

Übungsaufgabe 12: (3) Wenden Sie die Klasse `TypeCounter` auf die Klasse `CoffeeGenerator` (Seite 493) in Abschnitt 16.3 an. ■

Übungsaufgabe 13: (3) Wenden Sie die Klasse `TypeCounter` auf die Klasse `RegisteredFactories` (Seite 458) im folgenden Abschnitt an. ■

15.4 Registrierte Fabrikobjekte

[61] Beim Erzeugen von Objekten aus der `Pet`-Hierarchie, besteht das Problem, daß Sie jedesmal, wenn Sie eine neue Klasse von `Pet` ableiten, daran denken müssen, die neue Klasse in den von `allTypes` referenzierten *`List`*-Container im Beispiel *LiteralPetCreator.java* auf Seite 453 einzusetzen. In einem Programm oder einer Anwendung, die Sie regelmäßig um neue Klassen erweitern, kann dieser Umstand problematisch werden.

[62] Eventuell haben Sie sich überlegt, in jeder Klasse einen statischen Initialisierungsblock anzulegen, der die Klasse irgendwo registriert. Bedauerlicherweise wird ein statischer Initialisierungsblock nur beim erstmaligen Laden der Klasse verarbeitet, so daß Sie vor einem Henne-Ei-Problem stehen: Der Generator hat die Klasse noch nicht in seiner Liste, kann also kein Objekt dieser Klasse erzeugen, so daß die Klasse nicht geladen und in die Liste eingetragen wird.

[63] Sie sind im Grunde genommen gezwungen, die Liste selbst und von Hand anzulegen (es sei denn, Sie möchten eine Hilfsklasse schreiben, die Ihren Quelltext untersucht, die Liste zusammenstellt und übersetzt). Am besten platzieren Sie die Liste an einer zentralen offensichtlichen Stelle. Die Basisklasse der jeweiligen Hierarchie ist wahrscheinlich die beste Stelle.

[64] Eine weitere Änderung, die wir in diesem Abschnitt vornehmen, besteht darin, die Objekterzeugung, in Anlehnung an das Entwurfsmuster *Factorymethod*, in die jeweilige Klasse zu verlegen. Eine Fabrikmethode kann polymorph aufgerufen werden und erzeugt für Sie ein Objekt des entsprechenden Typs. In dieser sehr einfachen Fassung, ist die Methode `create()` aus dem folgenden Interface *`Factory`* die Fabrikmethode:

```
//: typeinfo/factory/Factory.java
package typeinfo.factory;
public interface Factory<T> { T create(); } ///:~
```

Der generische Typparameter `T` gestattet der `create()`-Methode, bei jeder Implementierung des Interfaces einen anderen Typ zurückzugeben. Insbesondere sind kovariante Rückgabetypen erlaubt.

[65] Im folgenden Beispiel enthält die Basisklasse `Part` einen *`List`*-Container mit Fabrikobjekten (`partFactories`). Die Fabrikobjekte der Typen, zu denen die `createRandom()`-Methode Objekte liefern soll, werden in den *`List`*-Container eingetragen und in diesem Sinne in der Basisklasse „registriert“:

```
//: typeinfo/RegisteredFactories.java
// Registering Class Factories in the base class.
import typeinfo.factory.*;
import java.util.*;

class Part {
    public String toString() {
        return getClass().getSimpleName();
    }
    static List<Factory<? extends Part>> partFactories =
        new ArrayList<Factory<? extends Part>>();
    static {
        // Collections.addAll() gives an ''unchecked generic
        // array creation ... for varargs parameter'' warning.
        partFactories.add(new FuelFilter.Factory());
        partFactories.add(new AirFilter.Factory());
        partFactories.add(new CabinAirFilter.Factory());
        partFactories.add(new OilFilter.Factory());
        partFactories.add(new FanBelt.Factory());
```

```
        partFactories.add(new PowerSteeringBelt.Factory());
        partFactories.add(new GeneratorBelt.Factory());
    }
    private static Random rand = new Random(47);
    public static Part createRandom() {
        int n = rand.nextInt(partFactories.size());
        return partFactories.get(n).create();
    }
}

class Filter extends Part {}

class FuelFilter extends Filter {
    // Create a Class Factory for each specific type:
    public static class Factory
        implements typeinfo.factory.Factory<FuelFilter> {
        public FuelFilter create() { return new FuelFilter(); }
    }
}

class AirFilter extends Filter {
    public static class Factory
        implements typeinfo.factory.Factory<AirFilter> {
        public AirFilter create() { return new AirFilter(); }
    }
}

class CabinAirFilter extends Filter {
    public static class Factory
        implements typeinfo.factory.Factory<CabinAirFilter> {
        public CabinAirFilter create() {
            return new CabinAirFilter();
        }
    }
}

class OilFilter extends Filter {
    public static class Factory
        implements typeinfo.factory.Factory<OilFilter> {
        public OilFilter create() { return new OilFilter(); }
    }
}

class Belt extends Part {}

class FanBelt extends Belt {
    public static class Factory
        implements typeinfo.factory.Factory<FanBelt> {
        public FanBelt create() { return new FanBelt(); }
    }
}

class GeneratorBelt extends Belt {
    public static class Factory
        implements typeinfo.factory.Factory<GeneratorBelt> {
        public GeneratorBelt create() {
            return new GeneratorBelt();
        }
    }
}

class PowerSteeringBelt extends Belt {
```

```
        public static class Factory
            implements typeinfo.factory.Factory<PowerSteeringBelt> {
            public PowerSteeringBelt create() {
                return new PowerSteeringBelt();
            }
        }
    }

    public class RegisteredFactories {
        public static void main(String[] args) {
            for(int i = 0; i < 10; i++)
                System.out.println(Part.createRandom());
        }
    } /* Output:
          GeneratorBelt
          CabinAirFilter
          GeneratorBelt
          AirFilter
          PowerSteeringBelt
          CabinAirFilter
          FuelFilter
          PowerSteeringBelt
          PowerSteeringBelt
          FuelFilter
       *///:~
```

Nicht zu jeder Klasse in einer Hierarchie sollen Objekte erzeugt werden. In diesem Fall sind die Klassen `Filter` und `Belt` nur Klassifizierungsknoten innerhalb der Hierarchie, von denen keine Objekt erzeugt werden, wohl aber von ihren Unterklassen. Jede Klasse, von der Objekte erzeugt werden sollen, enthält eine innere Klasse namens `Factory`. Die einzige Möglichkeit sich in der `implements`-Klausel auf das Interface *`Factory`* zu beziehen ist der voll qualifizierte Name *`typeinfo.factory.Factory`*.

[66] Sie können die Fabrikobjekte auch mit Hilfe der statischen `Collections`-Methode `addAll()` in den *`List`*-Container einsetzen, aber der Compiler gibt in diesem Fall eine Warnung mit dem Inhalt „generic array creation" aus (anscheinend unmöglich, siehe Abschnitt 16.8), so daß ich auf die `add()`-Methode des Container zurückgegriffen habe. Die `createRandom()`-Methode wählt ein zufälliges Fabrikobjekt aus dem von `partFactories` referenzierten *`List`*-Container und ruft seine `create()`-Methode auf, um ein neues `Part`-Objekt zurückzugeben.

Übungsaufgabe 14: (4) Ein Konstruktor ist eine Art Fabrikmethode. Ändern Sie das Beispiel *RegisteredFactories.java* (Seite 458), so daß das Klassenobjekt in einem *`List`*-Container gespeichert und die `newInstance()`-Methode anstelle der Fabrikmethode aufgerufen wird, um die Objekte zu erzeugen. ■

Übungsaufgabe 15: (4) Leiten Sie eine neue Klasse von `PetCreator` ab using/Registered/Factories und ändern Sie die *Façade*-Klasse `Pets`, so daß sie Ihre neue Klasse anstelle der beiden vorigen verwendet. Sorgen Sie dafür, daß die übrigen Beispiele, die sich auf *Pets.java* (Seite 454) beziehen korrekt funktioniert. ■

Übungsaufgabe 16: (4) Passen Sie die `Coffee`-Hierachie aus Abschnitt 16.3 (Seite 492) an das Beispiel *RegisteredFactories.java* (Seite 458) an. ■

15.5 Vergleichen von Klassen und Vergleichen von Klassenobjekten

[67] Beim Auswerten von Typinformationen besteht ein wesentlicher Unterschied zwischen dem `instanceof`-Operator und der `isInstance()`-Methode, die äquivalente Ergebnisse liefern und dem direkten Vergleich der Klassenobjekte. Das folgende Beispiel demonstriert diesen Unterschied:

```
//: typeinfo/FamilyVsExactType.java
// The difference between instanceof and class
package typeinfo;
import static net.mindview.util.Print.*;

class Base {}
class Derived extends Base {}

public class FamilyVsExactType {
    static void test(Object x) {
        print("Testing x of type " + x.getClass());
        print("x instanceof Base " + (x instanceof Base));
        print("x instanceof Derived "+ (x instanceof Derived));
        print("Base.isInstance(x) "+ Base.class.isInstance(x));
        print("Derived.isInstance(x) " + Derived.class.isInstance(x));
        print("x.getClass() == Base.class " + (x.getClass() == Base.class));
        print("x.getClass() == Derived.class " + (x.getClass() == Derived.class));
        print("x.getClass().equals(Base.class)) " +
              (x.getClass().equals(Base.class)));
        print("x.getClass().equals(Derived.class)) " +
              (x.getClass().equals(Derived.class)));
    }
    public static void main(String[] args) {
        test(new Base());
        test(new Derived());
    }
} /* Output:
      Testing x of type class typeinfo.Base
      x instanceof Base true
      x instanceof Derived false
      Base.isInstance(x) true
      Derived.isInstance(x) false
      x.getClass() == Base.class true
      x.getClass() == Derived.class false
      x.getClass().equals(Base.class)) true
      x.getClass().equals(Derived.class)) false
      Testing x of type class typeinfo.Derived
      x instanceof Base true
      x instanceof Derived true
      Base.isInstance(x) true
      Derived.isInstance(x) true
      x.getClass() == Base.class false
      x.getClass() == Derived.class true
      x.getClass().equals(Base.class)) false
      x.getClass().equals(Derived.class)) true
   *///:~
```

Die `test()`-Methode führt Typprüfungen mit ihrem Argument aus, wobei sowohl der `instanceof`-Operator als auch die `isInstance()`-Methode verwendet werden. Anschließend fordert `test()` die Referenz auf das Klassenobjekt ihres Argumentes an und vergleicht die Klassenobjekte sowohl per `==`-Operator als auch über die `equals()`-Methode. Auf beruhigende Weise liefern `instance-`

`of` und `isInstance()` exakt dasselbe Ergebnis, ebenso `equals()` und `==`. Aus den Tests ergeben sich aber verschiedene Schlußfolgerungen: Der `instanceof`-Operator ermittelt in Überstimmung mit dem Typkonzept, ob sei linkes Argument dem rechten Argument oder einem davon abgeleiteten Typ entspricht. Andererseits liefert der Vergleich der Klassenobjekte mit dem `==`-Operator kein Anzeichen auf eine Ableitungsbeziehung. Der Typ stimmt entweder exakt überein oder nicht.

15.6 Der Reflexionsmechanismus: Informationen über Klassen zur Laufzeit

[68] Der RTTI-Mechanismus ermittelt den genauen Typ eines Objektes, unterliegt allerdings einer Beschränkung: Damit Sie den Typ per RTTI auswerten und etwas nützliches mit dieser Information anfangen können, muß der Typ zur Übersetzungszeit bekannt sein. Mit anderen Worten, der Compiler muß über jede Klasse Bescheid wissen, mit der Sie arbeiten.

[69] Diese Bedingung wirkt auf den ersten Blick nicht wie eine Einschränkung. Stellen Sie sich aber vor, Ihr Programm arbeitet mit der Referenz auf ein Objekt, das nicht im Einflußbereich Ihres Programmes liegt. Vielmehr ist die Klasse dieses Objekt zur Übersetzungszeit Ihres Programmes nicht verfügbar. Stellen Sie beispielsweise eine Folge von Bytes vor, die Sie von der Festplatte oder über eine Netzwerkverbindung einlesen und wissen, daß diese Bytes eine Klasse repräsentieren. Wie können Sie sich auf eine solche Klasse beziehen, die erst nach der Übersetzung Ihres Programms erscheint?

[70] Aus der Perspektive der traditionellen Programmierung wirkt diese Situation weit hergeholt. Bei der Entwicklung moderner Anwendungen gibt es dagegen wichtige Beispiele für diese Situation. In der komponentenbasierten Softwareentwicklung werden Projekte nach dem Konzept des *Rapid Application Development* (RAD) in der *integrierten Entwicklungsumgebung* (IDE) eines ~~Application Builders~~ programmiert. In einer solchen visuellen Programmierumgebung entwickeln Sie ein Programm, indem Sie Symbole, die Komponenten darstellen, auf einem Formular anordnen. Diese Komponenten werden konfiguriert, indem Sie während der Entwicklungsphase einige ihrer Eigenschaften bewerten. Die Konfiguration während der Entwicklungsphase setzt voraus, daß von jeder Komponente Objekte erzeugt werden können, die einen Teil ihrer Eigenschaften und Fähigkeiten exponieren und das Abfragen und Ändern dieser Eigenschaft zulassen. Komponenten, die Ereignisse aus einer graphischen Benutzeroberfläche behandeln, müssen außerdem Informationen über die entsprechenden Methoden exponieren, damit die IDE den Programmier beim Überschreiben der Ereignisbehandler unterstützen kann. Der Reflexionsmechanismus ermittelt die verfügbaren Methoden und liefert die Methodennamen. Java unterstützt komponentenbasierte Softwareentwicklung durch JavaBeans (siehe Abschnitt 23.11).

[71–72] Die Befähigung Objekte auf einer entfernten Plattform über ein Netzwerk hinweg erzeugen und bedienen zu können, ist eine weitere Motivation, um Klasseninformationen zur Laufzeit zu ermitteln. Das entsprechende Protokoll heißt *Remote Method Invocation* (RMI) und gestattet einem Java-Programm, Objekte auf viele Rechner zu verteilen. Es gibt viele Gründe für eine solche Verteilung. Beispielsweise können Sie eine rechenintensive Aufgabe verarbeiten, die Sie in mehrere Teile zerlegen und auf unbenutzte Rechner verteilen, um die Verarbeitung zu beschleunigen. Oder Sie wollen Programmteile, die verschiedene Aufgaben erfüllen (zum Beispiel „Geschäftsregeln“ (*business rules*) in einer mehrschichtigen Client/Server-Architektur) auf einem bestimmten Rechner verarbeiten, der dadurch zum Aufbewahrungsort für solche Funktionalität wird. Die Software kann einfach geändert werden und die Änderung ist überall im System sichtbar. (Dies ist eine interessante Entwicklung, da dieser Rechner nur existiert, um Softwareänderungen zu erleichtern.) Verteilte Systeme unterstützen insbesondere spezialisierte Hardware, die für bestimmte Aufgaben entwickelt

wurde (zum Beispiel das Invertieren von Matrizen), für universelle Programmierung aber zu teuer ist.

[73] Die Klasse `Class` unterstützt, zusammen mit dem Package `java.lang.reflect`, welches die Klassen `Field`, `Method` und `Constructor` enthält (die jeweils das Interface *`Member`* implementieren), das Konzept der Reflexion („Reflexionsmechanismus"). Objekte dieser Klassen werden von der Laufzeitumgebung zur Laufzeit erzeugt und repräsentieren die entsprechenden Komponenten der unbekannten Klasse. Sie können das `Constructor`-Objekt verwenden, um neue Objekte zu erzeugen, die `get()`- und `set()`-Methoden eines `Field`-Objektes aufrufen, um Feldwerte abzufragen oder zu ändern, oder die `invoke()`-Methode eines `Method`-Objektes aufrufen, um eine Methode aufzurufen. Die `Class`-Methoden `getFields()`, `getMethods()` und `getConstructors()` geben Arrays vom Typ `Field`, `Method` beziehungsweise `Constructor` zurück, welche die entsprechenden Felder, Methoden und Konstruktoren darstellen. (In der API-Dokumentation der Klasse `Class` finden Sie weitere Methoden.) Die Informationen über die Klasse eines anonymen Objektes können also zur Laufzeit vollständig ermittelt werden und nichts muß zur Laufzeit bekannt sein.

[74] Es ist wichtig, zu verstehen, daß sich hinter dem Reflexionsmechanismus keinerlei Magie verbirgt. Wenn Sie den Reflexionsmechanismus anwenden, um mit einem Objekt unbekannten Typs zu kommunizieren, stellt die Laufzeitumgebung (wie bei RTTI) nur fest, daß es zu einer bestimmten Klasse gehört. Vor der Interaktion mit dem Objekt muß das entsprechende Klassenobjekt geladen werden. Die *.class* Datei des jeweiligen Typs muß also für die Laufzeitumgebung erreichbar sein, entweder lokal oder über das Netzwerk. Der Unterschied zwischen dem RTTI- und dem Reflexionsmechanismus besteht darin, daß die *.class* Datei bei RTTI zur Übersetzungszeit vom Compiler ausgewertet wird. Sie können also alle Methoden eines Objekt „normal" aufrufen. Beim Reflexionsmechanismus wird die *.class* Datei dagegen nicht zur zur Übersetzungs- sondern zur Laufzeit ausgewertet.

15.6.1 Ein Hilfsprogramm zum Anzeigen aller Methoden einer Klasse

[75] Im allgemeinen werden Sie die Hilfsmittel des Reflexionsmechanismus' nicht direkt einsetzen. Sie sind aber hilfreich, wenn Sie dynamische Funktionalität entwickeln. Der Reflexionsmechanismus gehört zum Sprachumfang, um Spracheigenschaften und -fähigkeiten wie die Serialisierung von Objekten und JavaBeans zu unterstützen (siehe Abschnitte 23.11 und 19.12). Es ist dennoch hin und wieder nützlich, zur Laufzeit Informationen über eine Klasse auswerten zu können.

[76] Das folgende Programm zeigt *alle* Methoden einer Klasse an. Beim Durchsehen einer Klassendefinition im Quelltext oder Nachlesen in der API-Dokumentation finden Sie nur die in der Klasse definierten beziehungsweise die von dieser Klasse überschriebenen Methoden. Die Klasse kann aber viele weitere Methoden von ihre Basisklassen geerbt haben. Es ist lästig und zeitaufwändig, alle diese Methoden zusammenzusuchen.[1] Der Reflexionsmechanismus bietet eine Möglichkeit, um die gesamte Schnittstelle einer Klasse mittels einer einfachen Hilfsklasse anzuzeigen:

```
//: typeinfo/ShowMethods.java
// Using reflection to show all the methods of a class,
// even if the methods are defined in the base class.
// {Args: ShowMethods}
import java.lang.reflect.*;
import java.util.regex.*;
import static net.mindview.util.Print.*;

public class ShowMethods {
```

[1] Vor allem in der Vergangenheit. Sun Microsystems hat die HTML-Dokumentation zu Java erheblich verbessert, so daß sich de Methoden aus den Basisklassen leichter ablesen lassen.

```
    private static String usage =
        ''usage:\n'' +
        ''ShowMethods qualified.class.name\n'' +
        ''To show all methods in class or:\n'' +
        ''ShowMethods qualified.class.name word\n'' +
        ''To search for methods involving 'word''';
    private static Pattern p = Pattern.compile(''\\w+\\.'');
    public static void main(String[] args) {
        if(args.length < 1) {
            print(usage);
            System.exit(0);
        }
        int lines = 0;
        try {
            Class<?> c = Class.forName(args[0]);
            Method[] methods = c.getMethods();
            Constructor[] ctors = c.getConstructors();
            if(args.length == 1) {
                for(Method method : methods)
                    print(p.matcher(method.toString()).replaceAll(''''));
                for(Constructor ctor : ctors)
                    print(p.matcher(ctor.toString()).replaceAll(''''));
                lines = methods.length + ctors.length;
            } else {
                for(Method method : methods)
                    if(method.toString().indexOf(args[1]) != -1) {
                        print(p.matcher(method.toString()).replaceAll(''''));
                        lines++;
                    }
                for(Constructor ctor : ctors)
                    if(ctor.toString().indexOf(args[1]) != -1) {
                        print(p.matcher(ctor.toString()).replaceAll(''''));
                        lines++;
                    }
            }
        } catch(ClassNotFoundException e) {
            print(''No such class: '' + e);
        }
    }
} /* Output:
      public static void main(String[])
      public native int hashCode()
      public final native Class getClass()
      public final void wait(long,int) throws InterruptedException
      public final void wait() throws InterruptedException
      public final native void wait(long) throws InterruptedException
      public boolean equals(Object)
      public String toString()
      public final native void notify()
      public final native void notifyAll()
      public ShowMethods()
   *///:~
```

[77] Die `Class`-Methoden `getMethods()` und `getConstructors()` geben ein Array von `Method`- beziehungsweise `Constructor`-Objekten zurück. Sowohl die Klasse `Method` als auch die Klasse `Constructor` verfügen über Methoden, welche den Namen, die Parametertypen und den Rückgabetyp der repräsentierten Methoden liefern. Alternativ gibt die `toString()`-Methode ein `String`-Objekt

zurück, das die gesamte Signatur der Methode darstellt. Die übrigen Anweisungen werten die Kommandozeilenargumente aus, ermitteln (per `indexOf()`), ob eine der Signaturen zu Ihrem Suchwort paßt und entfernen mit Hilfe regulärer Ausdrücke (siehe Abschnitt 14.6) den Packagepfad von allen Typnamen.

[78] Der Rückgabewert der `forName()`-Methode hängt vom ersten Kommandozeilenargument ab, kann zur Übersetzungszeit also nicht bekannt sein. Somit werden die Methodensignaturen zur Laufzeit ausgewertet. Wenn Sie in der API-Dokumentation des `java.lang.reflect`-Packages nachlesen, werden Sie feststellen, daß die Unterstützung weit genug reicht, um eine Methode eines Objektes aufrufen, das zur Übersetzungszeit völlig unbekannt ist (Sie finden Beispiele dafür ~~später in diesem Buch~~). Auch wenn Sie auf den ersten Blick nicht glauben, daß Sie den Reflexionsmechanismus jemals brauchen werden, sind seine Auswirkungen durchaus überraschend.

[79] Die obige Ausgabe stammt von dem folgenden Aufruf:

```
java ShowMethods ShowMethods
```

Die Ausgabe zeigt einen öffentlichen Standardkonstruktor an, obwohl die Klasse keinen Konstruktor definiert. Es handelt sich um den vom Compiler automatisch erzeugten Konstruktor. Wenn Sie `ShowMethods` in eine nicht-öffentliche Klasse ändern (zum Beispiel indem Sie das Schlüsselwort `public` entfernen und den Zugriff somit auf das Package einschränken), zeigt sich der automatisch erzeugte Standardkonstruktor nicht mehr in der Ausgabe. Der automatisch erzeugte Standardkonstruktor erhält automatisch die Zugriffseinstellung seiner Klasse.

[80] Eine weitere interessante Ausgabe liefert der Aufruf `java ShowMethods java.lang.String` mit einem der Zusatzargumente `char`, `int` oder `String`.

[81] Die Hilfsklasse `ShowMethods` kann tatsächlich Zeit sparen, wenn Sie sich beim Programmieren nicht erinnern können, ob eine Klasse eine bestimmte Methode hat oder nicht wissen, ob die Klasse beispielsweise `Color`-Objekte verarbeiten kann und nicht den Index oder die Klassenhierarchie in der API-Dokumentation durchsuchen wollen.

[82] Abschnitt 23.7.1 enthält eine GUI-Version von *ShowMethods.java* (angepaßt, um Informationen über Swing-Komponenten auszugeben). Lassen Sie das Programm nebenher laufen, um schnell etwas nachzuschlagen.

Übungsaufgabe 17: (2) Ändern Sie den regulären Ausdruck im Beispiel *ShowMethods.java*, so daß auch die Modifikatoren `native` und `final` entfernt werden (Tip: Verwenden Sie den logischen Oder-Operator `|`). ■

Übungsaufgabe 18: (1) Ändern Sie `ShowMethods` in eine nicht-öffentliche Klasse und verifizieren Sie, daß der automatisch erzeugte Standardkonstruktor nicht mehr in der Ausgabe erscheint. ■

Übungsaufgabe 19: (4) Wenden Sie Beispiel *ToyTest.java* (Seite 440) den Reflexionsmechanismus an, um mit Hilfe des parameterbehafteten Konstruktors (nicht Standardkonstruktor) ein `Toy`-Objekt zu erzeugen. ■

Übungsaufgabe 20: (5) Lesen Sie die Schnittstelle der Klasse `java.lang.Class` in der API-Dokumentation nach (*http://java.sun.com*). Schreiben Sie ein Programm, das ein Kommandozeilenargument erwartet. Verwenden Sie die Methoden der Klasse `Class`, um sämtliche über die Klasse verfügbaren Informantionen auszugeben. Testen Sie Ihr Programm mit einer Klasse aus der Standardbibliothek und einer Ihrer eigenen Klassen. ■

15.7 Dynamische Stellvertreter

[83] *Proxy* ist eines der grundlegenden Entwurfsmuster. Das Stellvertreterobjekt wird anstelle des eigentlichen Objektes eingesetzt, um zusätzliche oder andere Funktionalität zur Verfügung zu stellen. In der Regel ist dabei die Kommunikation mit dem eigentlichen Objekt erforderlich, so daß das Stellvertreterobjekt typischerweise die Rolle eines Vermittlers spielt. Das folgende triviale Beispiel zeigt die Struktur eines Stellvertreterobjektes:

```
//: typeinfo/SimpleProxyDemo.java
import static net.mindview.util.Print.*;

interface Interface {
    void doSomething();
    void somethingElse(String arg);
}

class RealObject implements Interface {
    public void doSomething() { print("doSomething"); }
    public void somethingElse(String arg) {
        print("somethingElse " + arg);
    }
}

class SimpleProxy implements Interface {
    private Interface proxied;
    public SimpleProxy(Interface proxied) {
        this.proxied = proxied;
    }
    public void doSomething() {
        print("SimpleProxy doSomething");
        proxied.doSomething();
    }
    public void somethingElse(String arg) {
        print("SimpleProxy somethingElse " + arg);
        proxied.somethingElse(arg);
    }
}

class SimpleProxyDemo {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }
    public static void main(String[] args) {
        consumer(new RealObject());
        consumer(new SimpleProxy(new RealObject()));
    }
} /* Output:
      doSomething
      somethingElse bonobo
      SimpleProxy doSomething
      doSomething
      SimpleProxy somethingElse bonobo
      somethingElse bonobo
   *///:~
```

Die Methode `consumer()` erwartet ein Argument vom Typ `Interface` und kann nicht unterscheiden, ob sie mit einem `RealObject`- oder einem `SimpleProxy`-Objekt aufgerufen wurde, da beide Klassen

das Interface `Interface` implementieren. Das zwischengeschaltete `SimpleProxy`-Objekt führt einige Operationen aus und ruft anschließend die gleichnamigen Methoden seines `RealObject`-Objektes auf.

[84] Ein Stellvertreterobjekt ist stets nützlich, wenn Sie zusätzliche Funktionalität an einer anderen Stelle als dem eigentlichen Objekt implementieren wollen und vor allem dann, wenn Sie einfach zwischen Anwendung und Nichtanwendung der zusätzliche Funktionalität hin- und herschalten müssen. (Entwurfsmuster sind dazu da, um Änderungen zu kapseln. Sie brauchen sich ändernde Eigenschaften oder Fähigkeiten, um den Einsatz des Entwurfsmusters zu rechtfertigen.) Stellen Sie sich beispielsweise vor, Sie wollten die Methodenaufrufe auf einem `RealObject`-Objekt verfolgen oder den Aufwand der Methodenaufrufe messen. Derartige Funktionalität gehört nicht in die eigentliche Anwendung und Sie können sie per Stellvertreterobjekt mühelos aktivieren und deaktivieren.

[85] Das Java-Konzept des dynamischen Stellvertreters entwickelt die Idee des Stellvertreters einen Schritt weiter, indem sowohl das Stellvertreterobjekt dynamisch erzeugt als auch die darauf aufgerufenen Methode dynamisch gehandhabt werden. Alle an ein dynamisches Stellvertreterobjekt gerichteten Methodenaufrufe werden an einen einzelnen *Aufrufbehandler* umgeleitet, der untersucht, was es mit dem Aufruf auf sich hat und entscheidet, wie damit zu verfahren ist. Das nächste Beispiel ist eine umgeschriebene Version von *SimpleProxyDemo.java* mit dynamischen Stellvertretern:

```
//: typeinfo/SimpleDynamicProxy.java
import java.lang.reflect.*;

class DynamicProxyHandler implements InvocationHandler {
    private Object proxied;
    public DynamicProxyHandler(Object proxied) {
        this.proxied = proxied;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        System.out.println("**** proxy: " + proxy.getClass() +
                           ", method: " + method + ", args: " + args);
        if(args != null)
            for(Object arg : args)
                System.out.println(" " + arg);
        return method.invoke(proxied, args);
    }
}

class SimpleDynamicProxy {
    public static void consumer(Interface iface) {
        iface.doSomething();
        iface.somethingElse("bonobo");
    }
    public static void main(String[] args) {
        RealObject real = new RealObject();
        consumer(real);
        // Insert a proxy and call again:
        Interface proxy =
            (Interface) Proxy.newProxyInstance(
                Interface.class.getClassLoader(),
                new Class[] { Interface.class },
                new DynamicProxyHandler(real));
        consumer(proxy);
    }
} /* Output: (95% match)
      doSomething
```

```
        somethingElse bonobo
        **** proxy: class $Proxy0, method: public abstract void
          Interface.doSomething(), args: null
        doSomething
        **** proxy: class $Proxy0, method: public abstract void
          Interface.somethingElse(java.lang.String),
          args: [Ljava.lang.Object;@42e816
        bonobo
        somethingElse bonobo
    *///:~
```

[86] Sie erzeugen einen dynamischen Stellvertreter, indem Sie die statische `Proxy`-Methode `newProxyInstance()` aufrufen, die einen Klassenlader (Sie können im allgemeinen den Klassenlader einer bereits geladenen Klasse übergeben), eine Liste von Interfaces (keine konkreten oder abstrakten Klassen), die das Stellvertreterobjekt implementieren soll und eine Implementierung des Interfaces *`InvocationHandler`* erwartet. Der dynamische Stellvertreter leitet alle Methodenaufrufe an den Aufrufbehandler um, weshalb der Konstruktor des Aufrufbehandlers üblicherweise die Referenz auf das eigentliche Objekt übergeben bekommt, um Methodenaufrufe während der Verarbeitung seiner intermediären Aufgabe weitergeben zu können.

[87] Für den Fall, daß der Aufrufbehandler unterscheiden muß, woher eine Anfrage stammt (in vielen Fällen ist dies unerheblich), wird der `invoke()`-Methode das Stellvertreterobjekt übergeben. Lassen Sie allerdings Sorgfalt walten, wenn Sie im Körper der `invoke()`-Methode Methoden des Stellvertreterobjektes aufrufen, da Aufrufe durch das Interface an das Stellvertreterobjekt umgeleitet werden.

[88] Im allgemeinen rufen Sie zuerst die Methode des Stellvertreterobjektes auf und anschließend die `Method`-Methode `invoke()`, um den Aufruf mit den erforderlichen Argumenten an das eigentliche Objekt weiterzugeben. Dies wirkt auf den ersten Blick so, als könnten Sie nur allgemeine Operationen ausführen. Sie können aber bestimmte Methodenaufrufe herausfiltern, während andere „durchgelassen“ werden:

```
//: typeinfo/SelectingMethods.java
// Looking for particular methods in a dynamic proxy.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

class MethodSelector implements InvocationHandler {
    private Object proxied;
    public MethodSelector(Object proxied) {
        this.proxied = proxied;
    }
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable {
        if(method.getName().equals("interesting"))
            print("Proxy detected the interesting method");
        return method.invoke(proxied, args);
    }
}

interface SomeMethods {
    void boring1();
    void boring2();
    void interesting(String arg);
    void boring3();
}

class Implementation implements SomeMethods {
```

```
    public void boring1() { print("boring1"); }
    public void boring2() { print("boring2"); }
    public void interesting(String arg) {
        print("interesting " + arg);
    }
    public void boring3() { print("boring3"); }
}

class SelectingMethods {
    public static void main(String[] args) {
        SomeMethods proxy = (SomeMethods)
            Proxy.newProxyInstance(SomeMethods.class.getClassLoader(),
            new Class[] { SomeMethods.class },
            new MethodSelector(new Implementation()));
        proxy.boring1();
        proxy.boring2();
        proxy.interesting("bonobo");
        proxy.boring3();
    }
} /* Output:
      boring1
      boring2
      Proxy detected the interesting method
      interesting bonobo
      boring3
   *///:~
```

[89] Wir betrachten in diesem Beispiel nur die Methodennamen. Sie können aber auch andere Teile der Signatur untersuchen, bis hin zu bestimmten Argumentwerten.

[90] Dynamische Stellvertreter sind kein alltägliches Werkzeug, liefern aber schöne Lösungen für bestimmte Arten von Problemen. Weiterführende Informationen über *Proxy* und andere Entwurfsmuster finden Sie in *Thinking in Patterns, Problem-Solving Techniques using Java* sowie in Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

Übungsaufgabe 21: (3) Ändern Sie das Beispiel *SimpleProxyDemo.java* (Seite 466), so daß es die für einen Methodenaufruf benötigte Zeit mißt. ■

Übungsaufgabe 22: (3) Ändern Sie das Beispiel *SimpleDynamicProxyDemo.java* (Seite 467), so daß es die für einen Methodenaufruf benötigte Zeit mißt. ■

Übungsaufgabe 23: (3) Versuchen Sie in der `invoke()`-Methode im Beispiel *SimpleDynamicProxyDemo.java* (Seite 467) den Parameter `proxy` auszugeben und erklären Sie was geschieht. ■

Projekt[2]**:** Verwenden Sie das Java-Konzept des dynamischen Stellvertreterobjektes, um ein Programm zu schreiben, das Transaktionen implementiert. Das Stellvertreterobjekt führt eine Transaktion aus, wenn der vertretene Methodenaufruf (auf dem eigentlichen Objekt) erfolgreich verarbeitet wurde (keine Ausnahme hervorruft) und bewirkt ein Rollback, wenn die Transaktion scheitert. Ausführung und Rollback der Transaktionen finden in einer externen Textdatei statt, die außerhalb der Kontrolle durch Java-Ausnahmen liegt. Achten Sie auf die Atomizität der Operationen (siehe Unterabschnitt 22.3.3). ■

[2] Projekte sind Vorschläge für Semester- oder Halbjahresarbeiten. Der *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können, enthält *keine* Lösungsvorschläge für Projekte.

15.8 Nullobjekte

[91] Wenn Sie die eingebaute `null`-Referenz verwenden, um die Abwesenheit eines Objektes anzuzeigen, müssen Sie eine Referenz vor Gebrauch stets auf `null` testen. Das kann sehr lästig werden und zu schwerfälligem Quelltext führen. Das Problem besteht darin, daß die `null`-Referenz, abgesehen vom Hervorrufen einer Ausnahme vom Typ `NullPointerException` beim Versuch etwas mit ihr zu tun, kein eigenes Verhalten hat. Gelegentlich ist es nützlich, das Konzept des „Nullobjektes“[3] zu implementieren. Ein Nullobjekt akzeptiert Methodenaufrufe auf dem Objekt, dessen Stelle es einnimmt und gibt Werte zurück, aus denen ersichtlich ist, daß eigentlich kein „richtiges“ Objekt existiert. Auf diese Weise können Sie davon ausgehen, daß alle Referenzen auf Objekte verweisen und brauchen keine Zeit zu verschwenden, um Referenzen auf `null` zu prüfen (und den resultierenden Quelltext zu lesen).

[92] Obwohl es interessant ist, sich eine Programmiersprache vorzustellen, die automatisch Nullobjekte erzeugt, ist das Konzept in der Praxis nicht überall sinnvoll anwendbar. Hin und wieder ist die `null`-Prüfung eine gute Lösung. Es gibt Situationen in denen Sie in vernünftiger Weise davon ausgehen können, daß es keine `null`-Referenzen geben wird und manchmal ist auch das Feststellen von Anomalien per `NullPointerException` eine akzeptable Lösung. Die Stelle an der Nullobjekte am nützlichsten sind, liegt „näher bei Daten“, genauer, bei den Objekten, welche Entitäten im Problemraum darstellen. Stellen Sie sich als Beispiel eine Anwendung mit einer Klasse `Person` vor. Es gibt Stellen im Quelltext, an denen Sie keine Person zur Verfügung haben (oder es ist bereits eine Person vorhanden, aber Sie haben nicht alle Informationen über diese Person), so daß Sie herkömmlicherweise `null`-Referenzen verwenden und auf `null` testen. Statt dessen können Sie ein Nullobjekt einsetzen. Obwohl das Nullobjekt auf alle Methoden reagiert, auf die das eigentliche Objekt reagieren würde, brauchen Sie noch immer einen `null`-Test. Der einfachste Weg, um diese Prüfung zu bewerkstelligen, besteht darin, ein Markierungsinterface anzulegen:

```
//: net/mindview/util/Null.java
package net.mindview.util;
public interface Null {} ///:~
```

[93] Das Interface *`Null`* gestattet die Erkennung von Nullobjekten mit Hilfe des `instanceof`-Operators und verlangt nicht, daß jede Ihrer Klassen eine `isNull()`-Methode besitzen muß. (Letzteres wäre schließlich nur ein weiterer Ansatz, der sich auf den RTTI-Mechanismus bezieht. Warum sollten wir nicht statt dessen die eingebaute Funktionalität nutzen?).

```
//: typeinfo/Person.java
// A class with a Null Object.
import net.mindview.util.*;

class Person {
    public final String first;
    public final String last;
    public final String address;
    // etc.
    public Person(String first, String last, String address){
        this.first = first;
        this.last = last;
        this.address = address;
    }
    public String toString() {
```

[3] Entdeckt von Bobby Woolf und Bruce Anderson. Das Konzept „Nullobjekt“ kann als Spezialfall des *Strategy*-Entwurfsmusters betrachtet werden. Eine Variante des Nullobjektes ist das Entwurfsmuster *Nulliterator*, welches die Iteration über die Knoten einer zusammengesetzten Hierarchie für den Client transparent beschreibt (der Client kann diese Logik dann nutzen, um über die Knoten der zusammengesetzten Hierarchie und die Blattknoten zu iterieren).

```
        return ''Person: '' + first + '' '' + last + '' '' + address;
    }
    public static class NullPerson extends Person implements Null {
        private NullPerson() { super(''None'', ''None'', ''None''); }
        public String toString() { return ''NullPerson''; }
    }
    public static final Person NULL = new NullPerson();
} ///:~
```

[94] Im allgemeinen ist das Nullobjekt ein Einzelexemplar (das heißt seine Klasse implementiert das *Singleton*-Entwurfsmuster) und wird daher in diesem Beispiel von einem statischen finalen Feld referenziert. Das funktioniert, weil die Klasse `Person` unveränderlich ist. Sie können die Eigenschaften nur per Konstruktor bewerten und abfragen, aber nicht ändern (da `String`-Objekte von Natur aus unveränderlich sind). Wenn Sie ein `NullPerson`-Objekt ändern wollen, können Sie es nur durch ein neues `Person`-Objekt ersetzen. Beachten Sie, daß Sie mit dem `instanceof`-Operator nur gegen den allgemeinen Typ *`Null`* und `NullPerson` vergleichen können, während Sie mit einem Einzelexemplar von `NullPerson` auch die `equals()`-Methode oder den `==`-Operator zur Verfügung haben, um gegen das `Person.NULL`-Feld zu vergleichen.

[95] Stellen Sie sich nun vor, Sie seien in die ehrgeizigen Tage der Internet-Startupunternehmen zurückversetzt worden und hätten Risikokapital, um Ihre faszinierende Idee zu realisieren. Sie sind bereit, um Personal einzustellen. Während Sie auf Bewerbungen warten, um die offenen Positionen zu besetzen, können Sie schon einmal Nullobjekte vom Typ `NullPerson` als Platzhalter für die verschiedenen Positionen einsetzen:

```
//: typeinfo/Position.java
class Position {
    private String title;
    private Person person;
    public Position(String jobTitle, Person employee) {
        title = jobTitle;
        person = employee;
        if(person == null)
            person = Person.NULL;
    }
    public Position(String jobTitle) {
        title = jobTitle;
        person = Person.NULL;
    }
    public String getTitle() { return title; }
    public void setTitle(String newTitle) {
        title = newTitle;
    }
    public Person getPerson() { return person; }
    public void setPerson(Person newPerson) {
        person = newPerson;
        if(person == null)
            person = Person.NULL;
    }
    public String toString() {
        return ''Position: '' + title + '' '' + person;
    }
} ///:~
```

Für die Klasse `Position` ist kein Nullobjekt erforderlich, da eine Referenz auf das Feld `Person.NULL` eine offene Position anzeigt. (Es ist zwar nicht auszuschließen, daß sich die Notwendigkeit eines

Nullobjektes für die Klasse `Position` später herausstellt, aber das YAGNI-Prinzip[4] („You Aren't Going to Need It") sagt aus, daß Sie beim ersten Entwurf die einfachste funktionstüchtigte Lösung versuchen und abwarten sollten, bis sich der Bedarf der Zusatzfunktionalität zeigt, statt ihre Notwendigkeit vorauszusetzen.)

[96] Die Klasse `Staff` kann nun auf Nullobjekte achten, wenn Sie Positionen besetzen:

```
//: typeinfo/Staff.java
import java.util.*;

public class Staff extends ArrayList<Position> {
    public void add(String title, Person person) {
        add(new Position(title, person));
    }
    public void add(String... titles) {
        for(String title : titles)
            add(new Position(title));
    }
    public Staff(String... titles) { add(titles); }
    public boolean positionAvailable(String title) {
        for(Position position : this)
            if(position.getTitle().equals(title) &&
                position.getPerson() == Person.NULL)
                return true;
        return false;
    }
    public void fillPosition(String title, Person hire) {
        for(Position position : this)
            if(position.getTitle().equals(title) &&
                position.getPerson() == Person.NULL) {
                position.setPerson(hire);
                return;
            }
        throw new RuntimeException("Position " + title + " not available");
    }
    public static void main(String[] args) {
        Staff staff =
            new Staff("President", "CTO", "Marketing Manager",
                      "Product Manager", "Project Lead",
                      "Software Engineer", "Software Engineer",
                      "Software Engineer", "Software Engineer",
                      "Test Engineer", "Technical Writer");
        staff.fillPosition(
            "President", new Person("Me", "Last", "The Top, Lonely At"));
        staff.fillPosition(
            "Project Lead", new Person("Janet", "Planner", "The Burbs"));
        if(staff.positionAvailable("Software Engineer"))
            staff.fillPosition(
                "Software Engineer",
                new Person("Bob", "Coder", "Bright Light City"));
        System.out.println(staff);
    }
} /* Output:
      [Position: President Person: Me Last The Top, Lonely At,
       Position: CTO NullPerson,
       Position: Marketing Manager NullPerson,
```

[4] Ein Grundsatz des Extreme Programming (XP): „Do the simplest thing that could possibly work."

```
        Position: Product Manager NullPerson,
        Position: Project Lead Person: Janet Planner The Burbs,
        Position: Software Engineer Person: Bob Coder Bright Light City,
        Position: Software Engineer NullPerson,
        Position: Software Engineer NullPerson,
        Position: Software Engineer NullPerson,
        Position: Test Engineer NullPerson,
        Position: Technical Writer NullPerson]
    *///:~
```

Beachten Sie, daß Sie stellenweise noch immer testen müssen, ob sich hinter einer Referenz ein Nullobjekt verbirgt, der Ansatz sich also nicht all zu sehr vom Testen auf `null` unterscheidet. Andernorts dagegen (in diesem Beispiel etwa bei der Umwandlung von Objekten in ihre `String`-Darstellung) sind keine zusätzlichen Tests erforderlich, da Sie voraussetzen können, daß alle Referenzen von `null` verschieden sind.

[97] Wenn Sie mit Interfaces anstelle konkreter Klassen arbeiten, können Sie mit Hilfe eines dynamischen Stellvertreters automatisch Nullobjekte erzeugen. Angenommen, ein Interface *`Robot`* deklariert drei Methoden, die Name (`name()`) und Modell (`model()`) eines Roboters sowie einen *`List<Operation>`*-Container (`operations()`) zurückgeben, dessen Elemente die einzelnen Funktionen des Roboters beschreiben. Das Interface *`Operation`* deklariert zwei Methoden: `description()` gibt einen Beschreibungstext zurück, während `command()` die durch das Interface beschriebene Funktion ausführt (eine Variante des *Command*-Entwurfsmusters):

```
//: typeinfo/Operation.java
public interface Operation {
    String description();
    void command();
} ///:~
```

Sie erhalten Zugriff auf die Funktionen des Roboters, indem Sie die `operations()`-Methode aufrufen:

```
//: typeinfo/Robot.java
import java.util.*;
import net.mindview.util.*;

public interface Robot {
    String name();
    String model();
    List<Operation> operations();
    class Test {
        public static void test(Robot r) {
            if(r instanceof Null)
                System.out.println("[Null Robot]");
            System.out.println("Robot name: " + r.name());
            System.out.println("Robot model: " + r.model());
            for(Operation operation : r.operations()) {
                System.out.println(operation.description());
                operation.command();
            }
        }
    }
} ///:~
```

Das Interface enthält eine innerere Klasse zum Testen der Funktionen des Roboters.

[98] Die folgende Klasse `SnowRemovalRobot` implementiert einen Roboter zum Schnee räumen:

```
//: typeinfo/SnowRemovalRobot.java
```

```
import java.util.*;

public class SnowRemovalRobot implements Robot {
    private String name;
    public SnowRemovalRobot(String name) {this.name = name;}
    public String name() { return name; }
    public String model() { return ''SnowBot Series 11''; }
    public List<Operation> operations() {
        return Arrays.asList(
            new Operation() {
                    public String description() {
                        return name + '' can shovel snow'';
                    }
                    public void command() {
                        System.out.println(name + '' shoveling snow'');
                    }
            },
            new Operation() {
                    public String description() {
                        return name + '' can chip ice'';
                    }
                    public void command() {
                        System.out.println(name + '' chipping ice'');
                    }
            },
            new Operation() {
                    public String description() {
                        return name + '' can clear the roof'';
                    }
                    public void command() {
                        System.out.println(name + '' clearing roof'');
                    }
            });     }
    public static void main(String[] args) {
        Robot.Test.test(new SnowRemovalRobot(''Slusher''));
    }
} /* Output:
      Robot name: Slusher
      Robot model: SnowBot Series 11
      Slusher can shovel snow
      Slusher shoveling snow
      Slusher can chip ice
      Slusher chipping ice
      Slusher can clear the roof
      Slusher clearing roof
   *///:~
```

[99] Da es voraussichtlich eine Reihe verschiedener Roboter geben wird, möchten wir das Verhalten der Nullobjekte an den Robotertyp anpassen, beispielsweise könnte das Nullobjekt den Robotertyp angeben, dessen Stelle es einnimmt. Der dynamische Stellvertreter erfaßt diese Information:

```
//: typeinfo/NullRobot.java
// Using a dynamic proxy to create a Null Object.
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;

class NullRobotProxyHandler implements InvocationHandler {
    private String nullName;
```

```
        private Robot proxied = new NRobot();
        NullRobotProxyHandler(Class<? extends Robot> type) {
            nullName = type.getSimpleName() + '' NullRobot'';
        }
        private class NRobot implements Null, Robot {
            public String name() { return nullName; }
            public String model() { return nullName; }
            public List<Operation> operations() {
                return Collections.emptyList();
            }
        }
        public Object invoke(Object proxy, Method method, Object[] args)
            throws Throwable {
            return method.invoke(proxied, args);
        }
    }

    public class NullRobot {
        public static Robot newNullRobot(Class<? extends Robot> type) {
            return (Robot) Proxy.newProxyInstance(
                NullRobot.class.getClassLoader(),
                new Class[] { Null.class, Robot.class },
                new NullRobotProxyHandler(type));
        }
        public static void main(String[] args) {
            Robot[] bots = {
                new SnowRemovalRobot(''SnowBee''),
                newNullRobot(SnowRemovalRobot.class)
            };
            for(Robot bot : bots)
                Robot.Test.test(bot);
        }
    } /* Output:
          Robot name: SnowBee
          Robot model: SnowBot Series 11
          SnowBee can shovel snow
          SnowBee shoveling snow
          SnowBee can chip ice
          SnowBee chipping ice
          SnowBee can clear the roof
          SnowBee clearing roof
          [Null Robot]
          Robot name: SnowRemovalRobot NullRobot
          Robot model: SnowRemovalRobot NullRobot
       *///:~
```

Wenn Sie ein Nullobjekt für einen Roboter brauchen, rufen Sie die Methode `newNullRobot()` auf und übergeben den Typ des zu ersetzenden Roboters. Das Stellvertreterobjekt erfüllt die in den Interfaces *`Robot`* und *`Null`* beschriebenen Anforderungen und gibt den Namen des Robotertyps an, dessen Stelle es einnimmt.

15.8.1 Mock- und Stubobjekte

[100] Mock- und Stubobjekte sind logische Variationen von Nullobjekten und spielen die Rolle des eigentlichen Objektes im fertigen Programm. Im Gegensatz zu Nullobjekten, die nur etwas intelligentere Platzhalter für die `null`-Referenz sind, geben Mock- und Stubobjekte aber vor, „lebendige“

Objekte zu sein, die echte Informationen liefern.

[101] Mock- und Stubobjekte unterscheiden sich durch den Grad ihrer Platzhalterfunktion. Mockobjekte sind tendentiell leichtgewichtige und selbsttestende Objekte und werden in der Regel in großen Anzahlen erzeugt, um in verschiedenen Testsituationen eingesetzt zu werden. Stubobjekte geben lediglich stubbed data zurück, sind typischerweise schwergewichtig und werden häufig von einem zum anderen Test wiederverwendet. Stubobjekte können so konfiguriert werden, daß sie sich in Abhängigkeit von der Art in der sie aufgerufen werden ändern. Ein Stubobjekt ist somit ein anspruchsvolles Objekt mit umfangreicher Funktionalität, während Sie im allgemeinen viele kleine einfache Mockobjekte erzeugen, um viele Dinge zu erledigen.

Übungsaufgabe 24: (4) Ergänzen Sie das Beispiel *RegisteredFactories.java* (Seite 458) um Nullobjekte. ■

15.9 Interfaces, Kopplung und vollständige Typinformation durch Reflexion

[102] Ein wichtige Funktion des Schlüsselwortes `interface` besteht darin, dem Programmierer das Isolieren von Komponenten zu erlauben und somit die Kopplung zu verringern. Wenn Sie ihre Referenzvariablen auf ein Interface beziehen, bleibt diese Eigenschaft erhalten. Der RTTI-Mechanismus gestattet allerdings, diesen Vorteil auszuhebeln. Interfaces sind keine wasserdichte Garantie für Entkopplung. Das folgende Beispiel beginnt mit der Definition eines Interfaces:

```
//: typeinfo/interfacea/A.java
package typeinfo.interfacea;

public interface A {
    void f();
} ///:~
```

[103] Die folgende Klasse `B` implementiert das Interface *`A`* und zeigt, wie sich der Typ der Implementierung einschleicht:

```
//: typeinfo/InterfaceViolation.java
// Sneaking around an interface.
import typeinfo.interfacea.*;

class B implements A {
    public void f() {}
    public void g() {}
}

public class InterfaceViolation {
    public static void main(String[] args) {
        A a = new B();
        a.f();
        // a.g(); // Compile error
        System.out.println(a.getClass().getName());
        if(a instanceof B) {
            B b = (B) a;
            b.g();
        }
    }
} /* Output:
      B
   *///:~
```

Der RTTI-Mechanismus liefert die Information, daß das von `a` referenzierte Objekt vom Typ `B` ist. Durch Umwandlung nach `B` kann eine Methode aufgerufen werden, die nicht in *`A`* deklariert ist.

[104] Dies ist zwar völlig rechtmäßig und akzeptabel, aber Sie möchten unter Umständen nicht, daß die Clientprogrammierer von dieser Möglichkeit Gebrauch machen, weil sie damit eine Gelegenheit hätten, ihre Programme enger an Ihre Klasse zu koppeln, als von Ihnen beabsichtigt. Sprich, Sie glauben vielleicht, daß Sie das Schlüsselwort `interface` schützt, aber dieser Schutz ist nicht gegeben und die Tatsache, daß die Klasse `B` das Interface *`A`* implementiert, ist effektiv eine Art „staatliche Urkunde“.[5]

[105] Eine Lösung besteht darin, die Programmierer darüber aufzuklären, daß sie auf sich alleine gestellt sind, wenn sie sich auf eine Klasse statt eines Interfaces beziehen. Diese Vorgehensweise ist wahrscheinlich in vielen Fällen das beste. Wenn aber „wahrscheinlich“ nicht ausreicht, wünschen Sie Sie vielleicht eine strengere Kontrollmöglichkeit.

[106] Das einfachste ist, die Implementierung unter Packagezugriff zu stellen, so daß die Klasse außerhalb ihres Packages unsichtbar ist:

```
//: typeinfo/packageaccess/HiddenC.java
package typeinfo.packageaccess;
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class C implements A {
    public void f() { print("public C.f()"); }
    public void g() { print("public C.g()"); }
    void u() { print("package C.u()"); }
    protected void v() { print("protected C.v()"); }
    private void w() { print("private C.w()"); }
}

public class HiddenC {
    public static A makeA() { return new C(); }
} ///:~
```

Die einzige öffentliche Komponente in diesem Package ist die Klasse `HiddenC`, deren `makeA()`-Methode die Referenz auf ein Objekt vom Interfacetyp *`A`* zurückgibt. An diesem Beispiel ist interessant, daß Sie die erhaltene Referenz außerhalb des Packages, selbst wenn `makeA()` den Typ `C` zurückgeben würde, nur als Typ *`A`* „ansprechen“ können, da die Klasse `C` dort nicht sichtbar ist.

[107] Wenn Sie nun eine abwärts gerichtete Typumwandlung nach `C` versuchen, meldet der Compiler, daß kein Typ `C` auffindbar ist:

```
//: typeinfo/HiddenImplementation.java
// Sneaking around package access.
import typeinfo.interfacea.*;
import typeinfo.packageaccess.*;
import java.lang.reflect.*;

public class HiddenImplementation {
    public static void main(String[] args) throws Exception {
        A a = HiddenC.makeA();
        a.f();
        System.out.println(a.getClass().getName());
```

[5] Das berühmteste Beispiel hierfür ist das Betriebssystem Windows. Es hatte eine öffentliche Programmierschnittstelle, auf die sich die Entwickler beziehen sollten und eine nicht öffentliche aber sichtbare Menge von Funktionen, die man entdecken und aufrufen konnte. Die Entwickler nutzten aber auch diese verborgenen Funktionen, um Probleme zu lösen und zwangen Microsoft dadurch, diese Funktionen wie Bestandteile der öffentlichen Schnittstelle zu pflegen. Dem Unternehmen erwuchsen daraus erhebliche Kosten und Mühen.

```
        // Compile error: cannot find symbol 'C':
        /* if(a instanceof C) {
            C c = (C)a;
            c.g();
            } */
        // Oops! Reflection still allows us to call g():
        callHiddenMethod(a, "g");
        // And even methods that are less accessible!
        callHiddenMethod(a, "u");
        callHiddenMethod(a, "v");
        callHiddenMethod(a, "w");
    }
    static void callHiddenMethod(Object a, String methodName)
        throws Exception {
        Method g = a.getClass().getDeclaredMethod(methodName);
        g.setAccessible(true);
        g.invoke(a);
    }
} /* Output:
      public C.f()
      typeinfo.packageaccess.C
      public C.g()
      package C.u()
      protected C.v()
      private C.w()
   *///:~
```

Wie Sie sehen, ist es mit Hilfe des Reflexionsmechanismus noch immer möglich, den eigentlichen Typ eines Objektes zu erreichen und *sämtliche* Methoden aufrufen, insbesondere *private* Methoden! Ist der Name der Methode bekannt, so können Sie die Methode aufrufbar machen, indem Sie die `setAccessible()`-Methode des entsprechenden `Method`-Objektes mit dem Argument `true` aufrufen (siehe Methode `callHiddenMethod()`).

[108] Vielleicht denken Sie, dies ließe sich dadurch verhindern, daß Sie nur übersetzten Code herausgeben, aber das ist keine Lösung. Es genügt, das Hilfsprogramm `javap` aufzurufen, den Decompiler der zum Java Development Kit gehört. Der Decompiler wird wie folgt aufgerufen:

```
javap -private C
```

Der Schalter `-private` weist den Decompiler an, alle Komponenten anzuzeigen, auch die privaten. Die Ausgabe lautet:

```
class typeinfo.packageaccess.C extends java.lang.Object
    implements typeinfo.interfacea.A {
    typeinfo.packageaccess.C();
    public void f();
    public void g();
    void u();
    protected void v();
    private void w();
}
```

Somit kann jeder die Namen und Signaturen Ihrer privaten Methoden abfragen und sie aufrufen.

[109] Hilft es vielleicht, das Interface als private innere Klase zu implementieren? Beispielsweise so:

```
//: typeinfo/InnerImplementation.java
// Private inner classes can't hide from reflection.
import typeinfo.interfacea.*;
```

```
import static net.mindview.util.Print.*;

class InnerA {
    private static class C implements A {
        public void f() { print("public C.f()"); }
        public void g() { print("public C.g()"); }
        void u() { print("package C.u()"); }
        protected void v() { print("protected C.v()"); }
        private void w() { print("private C.w()"); }
    }
    public static A makeA() { return new C(); }
}

public class InnerImplementation {
    public static void main(String[] args) throws Exception {
        A a = InnerA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Reflection still gets into the private class:
        HiddenImplementation.callHiddenMethod(a, "g");
        HiddenImplementation.callHiddenMethod(a, "u");
        HiddenImplementation.callHiddenMethod(a, "v");
        HiddenImplementation.callHiddenMethod(a, "w");
    }
} /* Output:
      public C.f()
      InnerA$C
      public C.g()
      package C.u()
      protected C.v()
      private C.w()
   *///:~
```

[110] Auch so läßt sich nichts vor dem Reflexionsmechanismus verbergen. Vielleicht eine anonyme innere Klasse?

```
//: typeinfo/AnonymousImplementation.java
// Anonymous inner classes can't hide from reflection.
import typeinfo.interfacea.*;
import static net.mindview.util.Print.*;

class AnonymousA {
    public static A makeA() {
        return new A() {
                public void f() { print("public C.f()"); }
                public void g() { print("public C.g()"); }
                void u() { print("package C.u()"); }
                protected void v() { print("protected C.v()"); }
                private void w() { print("private C.w()"); }
            };
    }
}

public class AnonymousImplementation {
    public static void main(String[] args) throws Exception {
        A a = AnonymousA.makeA();
        a.f();
        System.out.println(a.getClass().getName());
        // Reflection still gets into the anonymous class:
        HiddenImplementation.callHiddenMethod(a, "g");
```

```
        HiddenImplementation.callHiddenMethod(a, ''u'');
        HiddenImplementation.callHiddenMethod(a, ''v'');
        HiddenImplementation.callHiddenMethod(a, ''w'');
    }
} /* Output:
      public C.f()
      AnonymousA$1
      public C.g()
      package C.u()
      protected C.v()
      private C.w()
   *///:~
```

Es scheint keine Möglichkeit zu geben, den Reflexionsmechanismus am Zugang zu und Aufrufen von Methoden zu hindern, die nicht unter öffentlichem Zugriff stehen. Dies gilt auch für Felder, insbesondere private Felder:

```
//: typeinfo/ModifyingPrivateFields.java
import java.lang.reflect.*;

class WithPrivateFinalField {
    private int i = 1;
    private final String s = ''I'm totally safe'';
    private String s2 = ''Am I safe?'';
    public String toString() {
        return ''i = '' + i + '', '' + s + '', '' + s2;
    }
}

public class ModifyingPrivateFields {
    public static void main(String[] args) throws Exception {
        WithPrivateFinalField pf = new WithPrivateFinalField();
        System.out.println(pf);
        Field f = pf.getClass().getDeclaredField(''i'');
        f.setAccessible(true);
        System.out.println(''f.getInt(pf): '' + f.getInt(pf));
        f.setInt(pf, 47);
        System.out.println(pf);
        f = pf.getClass().getDeclaredField(''s'');
        f.setAccessible(true);
        System.out.println(''f.get(pf): '' + f.get(pf));
        f.set(pf, ''No, you're not!'');
        System.out.println(pf);
        f = pf.getClass().getDeclaredField(''s2'');
        f.setAccessible(true);
        System.out.println(''f.get(pf): '' + f.get(pf));
        f.set(pf, ''No, you're not!'');
        System.out.println(pf);
    }
} /* Output:
      i = 1, I'm totally safe, Am I safe?
      f.getInt(pf): 1
      i = 47, I'm totally safe, Am I safe?
      f.get(pf): I'm totally safe
      i = 47, I'm totally safe, Am I safe?
      f.get(pf): Am I safe?
      i = 47, I'm totally safe, No, you're not!
   *///:~
```

Finale Felder sind aber vor Änderungen geschützt. Die Laufzeitumgebung akzeptiert zwar Änderungsversuche ohne sich zu beschweren, läßt finale Felder aber tatsächlich unverändert.

[111] Diese Zugriffsverletzungen sind allerdings weniger schlimm, als es auf den ersten Blick scheint. Wenn ein Programmierer per Reflexionsmechanismus eine Methode aufruft, die Sie als privat markiert oder unter Packagezugriff gestellt und damit deutlich angezeigt haben, daß sie nicht aufgerufen werden soll, kann sich dieser Programmierer kaum bei Ihnen beschweren, wenn Sie die Funktionsweise einer solchen Methode verändern. Auf der anderen Seite kann Ihnen eine solche Hintertür das Lösen von Problemen ermöglichen, die sonst schwierig oder überhaupt nicht gelöst werden können. Die Vorteile des Reflexionsmechanismus sind darüber hinaus in der Regel nicht von der Hand zu weisen.

Übungsaufgabe 25: (2) Schreiben Sie eine Klasse mit einer privaten, einer geschützten (`protected`) und einer Methode unter Packagezugriff. Rufen Sie diese Methoden außerhalb des Packages auf, in dem die Klasse liegt. ■

15.10 Zusammenfassung

[112] Der RTTI-Mechanismus gestattet Ihnen, den eigentlichen Typ eines Objektes zu ermitteln, das von einer anonymen Referenzvariablen vom Typ einer Basisklasse referenziert wird. Der RTTI-Mechanismus bietet sich damit zum Mißbrauch durch unerfahrene Programmierer an, weil seine Funktionsweise einleuchtet, bevor der Programmierer den Sinn des polymorphen Methodenaufrufs verstanden hat. Programmierer mit prozeduralem Hintergrund haben Schwierigkeiten damit, ein Programm nicht mit Hilfe von `switch`-Anweisungen aufzubauen. Sie können polymorphe Methodenaufrufe durch die Kombination von `switch` und RTTI ersetzen, opfern damit aber die wertvollen Auswirkungen der Polymophie bei der Entwicklung und Pflege des Quelltextes. Eines der Ziele der objektorientierten Programmierung besteht darin, so oft wie möglich polymorphe Methodenaufrufe auszuführen und den RTTI-Mechanismus nur dann zu gebrauchen, wenn Sie müssen.

[113] Der bestimmungsgemäße Einsatz polymorpher Methodenaufrufe setzt allerdings voraus, daß Sie die Definition der Basisklasse beeinflussen können, da Sie beim Erweitern Ihres Programms irgendwann den Punkt erreichen, an dem Sie erkennen, daß die Basisklasse eine erforderliche Methode nicht enthält. Stammt die Bibliothek von einem Fremdanbieter, so bietet der RTTI-Mechanismus eine Lösung: Sie können eine neue Klassen ableiten und die benötigten Methoden hinzufügen. Sie können den neuen Typ an jeder Stelle Ihres Programms auswerten und die spezielle Methode aufrufen. Polymorphie und Erweiterbarkeit des Programmes bleiben erhalten, da Sie nach dem Anlegen eines neuen Typs keine `switch`-Anweisungen in Ihrem Quelltext suchen müssen. Wenn Sie die neue Funktionalität nutzen wollen, müssen Sie den RTTI-Mechanismus aufrufen, um Ihre zusätzliche Klasse zu erkennen.

[114] Das Anlegen einer Methode in der Basisklasse zum Nutzen einer bestimmten Unterklasse bedeutet, daß alle von dieser Basisklasse abgeleiteten Klassen eine bedeutungslose Methode enthalten. Einerseits leidet darunter die Klarheit der Schnittstelle der Basisklasse und andererseits belästigen Sie diejenigen, welche die abstrakten Methoden überschreiben müssen, wenn Sie eine Klasse von dieser Basisklasse ableiten. Wir betrachten eine Hierarchie von Musikinstrumenten als Beispiel. Angenommen, Sie wollen die Wasserklappen aller entsprechenden Instrumente öffnen. Eine Lösung besteht darin, eine `clearSpitValve()`-Methode in die Basisklasse `Instrument` einzusetzen, ist aber verwirrend, da in diesem Fall auch Schlag-, Seiten- und elektronische Instrumente ebenfalls eine Wasserklappe bekommen. RTTI gestattet eine viel sinnvollere Lösung, da Sie die `clearSpitValve()`-Methode in einer speziellen Klasse anlegen können, zu der die Methode paßt (hier `Wind`). Eventuell entdecken Sie gleichzeitig eine noch sinnvollere Lösung, nämlich eine `prepareInstru-`

`ment()`-Methode in der Basisklasse. Beim ersten Lösungsansatz eines Problems erkennen Sie aber eine solche Lösung vielleicht nicht und gehen fälschlicherweise davon aus, daß Sie auf den RTTI-Mechanismus zurückgreifen müssen.

[115] Schließlich hilft RTTI manchmal dabei, Effizienzprobleme zu lösen. Stellen Sie sich vor, daß Ihr Programm schöne von Polymorphie Gebrauch macht, Sie aber feststellen, daß eines Ihrer Objekte auf äußerst ineffiziente Weise auf Ihre universelle Funktionalität reagiert. Dann können Sie diesen Typ per RTTI herausfiltern und durch fallspezifisch angepaßtes Verhalten die Effizienz verbessern. Hüten Sie aber davor, Effizienzbetrachtungen zu früh in die Programmierung einzubeziehen. Es ist eine verlockende Falle. Am besten bringen Sie das Programm zuerst in einen funktionstüchtigen Zustand und entscheiden anschließend ob es schnell genug läuft. Nur wenn das Programm nicht schnell genug verarbeitet wird, sollten Sie sich um Effizienzbetrachtungen kümmern und zwar mit einem Profiler (siehe Anhang von *http://www.mindview.net/Books/BetterJava*).

[116] Sie haben gesehen, daß der Reflexionsmechanismus das Tor zu einer erheblichen Bandbreite von Programmiermöglichkeiten aufstößt, indem er einen viel dynamischeren Programmierstil ermöglicht. Einige Kollegen empfinden die dynamische Natur des Reflexionsmechanismus' als beunruhigend. Die Tatsache, daß Sie Dinge tun können, die nur zur Laufzeit geprüft und durch Ausnahmen angezeigt werden können, deutet aus der Perspektive derjenigen Kollegen, welche sich an die Sicherheit durch statische Typprüfung gewöhnt haben, in die falsche Richtung. Manche Kollegen vertreten sogar die Auffassung, daß die Möglichkeit, zur Laufzeit Ausnahmen auswerfen zu können, ein klares Zeichen dafür ist, solchen Code zu vermeiden. Ich halte dieses Verständnis von Sicherheit für eine Illusion, da zur Laufzeit stets Situationen eintreten können, in deren Folge eine Ausnahme hervorgerufen wird, selbst bei einem Programm, das weder `try`-Blöcke noch `throws`-Klauseln enthält. Ich bin stattdessen der Meinung, daß uns die Existenz eines konsistenten Fehlermeldungsmodells dazu bevollmächtigt, mit Hilfe des Reflexionsmechanismus dynamisch zu programmieren. Es ist selbstverständlich die Mühe wert, den Quelltext so zu schreiben, daß statische Typprüfung ausreicht, ... sofern Sie können. Ich glaube, daß dynamischer Code eine der wesentlichen Eigenschaften ist, die Java von Sprachen wie C++ unterscheidet.

Übungsaufgabe 26: (3) Implementieren Sie Methode `clearSpitValve()` wie in der Zusammenfassung beschrieben. ■

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 16

Generische Typen

Inhaltsübersicht

[0] Gewöhnliche Klassen und Methoden arbeiten mit bestimmten Typen: Entweder primitiven Typen oder Klassen beziehungsweise Interfaces. Beim Implementieren einer Funktionalität, die über mehr als einen Typ hinweg verwendet werden kann, kann diese Starrheit eine zu starke Einschränkung sein.[1]

[1] Ein Weg auf dem objektorientierte Programmiersprachen Generalisierung zulassen ist Polymorphie. Sie schreiben beispielsweise eine Methode, die ein Objekt einer Basisklasse als Argument erwartet und können diese Methode anschließend mit Objekten einer beliebigen von dieser Basisklasse abgeleiteten Klasse aufrufen. Somit ist Ihre Methode etwas allgemeiner und kann an mehreren Stellen verwendet werden. Dasselbe gilt für Klassen: An jeder Stelle, an der Sie eine bestimmte Klasse einsetzen, bewirkt eine Basisklasse oder ein Interface mehr Flexibilität. Natürlich läßt sich jeder Typ außer einer finalen Klasse oder einer Klasse, die nur private Konstruktoren besitzt, erweitern, das heißt diese Art von Flexibilität ist fast immer automatisch gegeben.

[2] In manchen Fällen ist die Bindung an eine einzelne Ableitungslinie eine zu starke Einschränkung. Wenn Sie bei der Typdeklaration des obigen Methodenargumentes die Klasse durch ein Interface ersetzen, wird diese Beschränkung gelockert, da nun alle Klassen erfaßt werden, die dieses Interface implementieren, insbesondere auch Klassen, die noch nicht existieren. Ein Programmierer, der Ihre Klasse oder Methode verwenden möchte, hat nun die Möglichkeit, ein Interface zu implementieren, um seine Klasse an den erforderlichen Typ anzupassen. Interfaces gestatten dem Programmierer also,

[1] Angelika Langers „Java Generics FAQ“ (siehe *http://www.angelikalanger.com/GenericsFAQ/JavaGenericsFAQ-.html*) und ihre weiteren Publikationen (zusammen mit Klaus Kreft) waren während der Arbeit an diesem Kapitel von unschätzbarem Wert.

die Grenze zwischen Klassenhierarchien zu überschreiten, sofern es möglich ist, eine entsprechende neue Klasse anzulegen.

[3] Es gibt aber Situationen, in denen sogar ein Interface noch eine zu starke Einschränkung bedeutet. Ein Interface erzwingt noch immer, daß sich Ihr Quelltext auf dieses Interface bezieht. Sie könnten Ihre Klasse oder Methode noch allgemeiner anlegen, wenn Sie festlegen könnten, daß Ihre Anweisungen mit „irgend einem nicht genau angegeben Typ“ arbeiten, statt mit einem spezifischen Interface oder eine spezifischen Klasse.

[4] Dies ist das Konzept der generischen Typen, einer der bedeutsameren Änderungen in Version 5 der Java Standard Edition (SE 5). Generische Typen implementieren das Konzept der *parametrisierten Typen*, which allow multiple types. Das Attribut „generisch“ bedeutet „in Bezug auf oder passend für eine große Anzahl von Klassen“. Die ursprüngliche Absicht hinter der Aufnahme generischer Typen in Programmiersprachen bestand darin, dem Entwickler beim Schreiben von Klassen und Methoden die größtmögliche Ausdrucksfreiheit zu verleihen, in dem die Beschränkungen hinsichtlich der von diesen Klassen und Methoden verwendeten Typen wegfallen. Wie Sie in diesem Kapitel sehen werden, wird die Java-Implementierung der generischen Typen diesem Anspruch nicht in vollem Umfang gerecht. Tatsächlich werden Sie sich vielleicht fragen, ob die Bezeichnung „generisch“ der in diesem Kapitel beschriebenen Eigenschaft angemessen ist.

[5] Wenn Sie noch nie mit einer anderen Implementierung parametrisierter Typen in Berührung gekommen sind, werden Sie die generischen Typen von Java wahrscheinlich für eine bequeme Spracherweiterung halten. Wenn Sie ein Objekt eines parameterisierten Typs erzeugen, müssen Sie sich nicht um Typumwandlungen kümmern und die Typsicherheit wird bereits zur Übersetzungszeit gewährleistet. Das scheint eine Verbesserung zu sein.

[6] Falls Sie aber bereits Erfahrung mit parameterisierten Typen haben, beispielsweise aus C++, dann werden Sie feststellen, daß Sie mit den generischen Typen von Java nicht alles tun können, was Sie eventuell erwartet haben. Während die Verwendung eines von einem anderen Entwickler geschriebenen generischen Typs ziemlich einfach ist, werden Sie auf einige Überraschungen stoßen, wenn Sie selbst einen solchen Typ entwickeln wollen. Zu den Dingen, die ich in diesem Kapitel erklären möchte gehört, wie sich die generischen Typen zu ihrer jetzigen Form entwickelt haben.

[7] Die generischen Typen von Java sind keineswegs nutzlos, sondern machen den Quelltext in vielen Fällen einfacher und sogar elegant. Sofern Sie von einer Sprache kommen, die eine reinere Version dieses Konzepts implementiert haben, werden Sie eventuell enttäuscht werden. In diesem Kapitel werden wir sowohl die Stärken als auch die Schwächen der generischen Typen von Java untersuchen, so daß Sie diese neuen Spracheigenschaft effektiver einsetzen können.

16.1 Vergleich mit C++

[8] Die Designer der Programmiersprache Java haben einmal gesagt, der schöpferische Gedanke sei zu einem großen Teil von der Motivation getragen worden, eine Reaktion auf C++ zu schaffen. Dennoch ist es möglich, Java zu lehren, ohne sich auf C++ zu beziehen. Ich habe mich bemüht diesem Weg treu zu bleiben, mit Ausnahme der Fälle, in denen der Vergleich mit C++ dem tieferen Verständnis dient.

[9] Generische Typen verlangen aus zwei Gründen stärker nach dem Vergleich mit C++. Erstens hilft Ihnen das Verständnis bestimmter Eigenschaften der C++ Templates (die hauptsächliche Anregung für die generischen Typen von Java, inklusive der Syntax) dabei, die Grundlagen des Konzepts und wichtiger noch, die Grenzen sowie die Hintergründe dessen zu verstehen, was Sie mit den generischen Typen von Java tun können. Letztendlich besteht das Ziel darin, Ihnen ein klares

Verständnis hinsichtlich des Verlaufs der Grenzen zu vermitteln, da Ihnen die Kenntnis der Grenzen nach meiner Erfahrung hilft, ein leistungsstärkerer Entwickler zu werden. Wenn Sie wissen, was Sie nicht tun können oder dürfen, können Sie den Rahmen Ihrer Möglichkeiten besser nutzen (nicht zuletzt deshalb, weil Sie keine Zeit damit verlieren, sich den Kopf an der Wand einzurennen).

[10–11] Der zweite Grund ist die Tatsache, daß in der Java Entwicklergemeinde ein wesentliches Mißverständnis über C++ Templates kursiert, welches Sie im Hinblick auf das Ziel der generische Typen eventuell zusätzlich verwirrt. Ich habe C++ Templates trotz einiger eingeflochtener Beispiele auf ein Minimum beschränkt.

16.2 Einfache generische Typen

[12] Die Containerklassen aus Kapitel 12 sind einer der Hauptgründe für die Existenz der generischen Typen (mehr über die Containerklassen in Kapitel 18). Ein Container dient zur Aufbewahrung von Objekten, während Sie mit diesen Objekten arbeiten. Obwohl dies auch für Arrays gilt, sind Container (auch „Kollektionen“) in der Regel flexibler und zeichnen sich durch andere Eigenschaften aus als Arrays. Nahezu jedes Programm erfordert die Speicherung einer Anzahl von Objekten, während Sie mit ihnen arbeiten. Somit gehören Container zu den Bibliotheksklassen mit dem höchsten Wiederverwendungswert.

[13] Wir betrachten zunächst eine Klasse, die ein einzelnes Objekt enthält. Die Klasse kann selbstverständlich den Typ dieses Objekte exakt angeben:

```
//: generics/Holder1.java
class Automobile {}

public class Holder1 {
    private Automobile a;
    public Holder1(Automobile a) { this.a = a; }
    Automobile get() { return a; }
} ///:~
```

Die Klasse `Holder1` ist im Hinblick auf ihre Wiederwendbarkeit nicht besonders nützlich, da das enthaltene Objekt ausschließlich von Typ `Automobile` sein kann. Wir würden es vorziehen, nicht für jeden Objekttyp eine neue solche Klasse schreiben zu müssen.

[14] Vor der SE 5 hätten wir dazu statt `Automobile` einfach den Typ `Object` gewählt:

```
//: generics/Holder2.java
public class Holder2 {
    private Object a;
    public Holder2(Object a) { this.a = a; }
    public void set(Object a) { this.a = a; }
    public Object get() { return a; }
    public static void main(String[] args) {
        Holder2 h2 = new Holder2(new Automobile());
        Automobile a = (Automobile) h2.get();
        h2.set("Not an Automobile");
        String s = (String) h2.get();
        h2.set(1); // Autoboxes to Integer
        Integer x = (Integer) h2.get();
    }
} ///:~
```

Nun kann ein Objekt der Klasse `Holder2` ein Objekt beliebigen Typs beinhalten, in diesem Beispiel nacheinander Objekte drei verschiedener Typen.

[15] Nun gibt es Situationen, in denen wir einen Container zwar *grundsätzlich* für Objekte verschiedener Typen vorsehen möchten, im Anwendungsfall aber typischerweise nur Objekte *eines* bestimmten Typs in einem solchen Container speichern wollen. Eine wesentliche Motivation für die Erweiterung des Sprachumfangs um generische Typen bestand darin, den Typ der in einem Container gespeicherten Objekte sowohl festlegen, als auch durch eine compilerseitige Überprüfung sicherstellen zu können.

[16] Wir wollen also anstelle von `Object` einen nicht genau angegeben Typ verwenden, der erst zu einem späteren Zeitpunkt gewählt wird. Zu diesem Zweck notieren wir nach dem Klassennamen einen *Typparameter* in spitzen Klammern, der beim späteren Gebrauch der Klasse durch den tatsächlichen Typ ersetzt wird. Mit dem Typparameter `T` nimmt unser Beispiel die folgende Gestalt an:

```
//: generics/Holder3.java
public class Holder3<T> {
    private T a;
    public Holder3(T a) { this.a = a; }
    public void set(T a) { this.a = a; }
    public T get() { return a; }
    public static void main(String[] args) {
        Holder3<Automobile> h3 =
            new Holder3<Automobile>(new Automobile());
        Automobile a = h3.get(); // No cast needed
        // h3.set("Not an Automobile"); // Error
        // h3.set(1); // Error
    }
} ///:~
```

[17] Beim Erzeugen eines `Holder3`-Objektes müssen wir den Typ der erwarteten Elemente in einer spitzen Klammer angeben (siehe `main()`). Wir dürfen nur Objekte dieses Typs oder eines von diesem abgeleiteten Typs speichern (das Substitutionsprinzip von Seite 33 gilt auch bei generischen Typen). Ein aus einem solchen Container entnommenes Objekt hat automatisch den richtigen Typ (im Gegensatz zu `Object`).

[18] Der Grundgedanke der generischen Typen von Java besteht darin, den Typ der in einem Container gespeicherten Objekte zu definieren und die Einzelheiten dem Compiler zu überlassen.

[19] Im allgemeinen behandeln Sie generische Typen wie alle anderen Typen auch, wobei generische Typen zufälligerweise Typparameter haben. Wie Sie lernen werden, müssen Sie bei der Verwendung eines generischen Typs lediglich die Liste der Parametertypen angeben.

Übungsaufgabe 1: (1) Wenden Sie ein Objekt der Klasse `Holder3` auf die Typen im Package `typeinfo.pets` an, um zu zeigen, daß ein für die Aufnahme von Objekten des Basistyps definiertes `Holder3`-Objekt auch Objekte eines vom Basistyp abgeleiteten Typs aufnimmt. ■

Übungsaufgabe 2: (1) Schreiben Sie eine Klasse, die drei Objekte ein und desselben Typs beinhaltet und legen Sie sowohl die Abfrage- und Änderungsmethoden dieser Objekte als auch einen Konstruktor an, der alle drei Felder bewertet. ■

16.2.1 Eine Bibliothek von Tupel-Klassen

[20] Häufig möchten Sie mehr als ein Objekt aus einer Methode zurückgeben. Da die `return`-Anweisung nur die Rückgabe eines einzigen Wertes erlaubt, besteht die Lösung darin, ein Objekt zurückzugeben, welches wiederum die Objekte beinhaltet, die Sie eigentlich zurückgeben wollen. Sie können selbstverständlich jedesmal eine Klasse schreiben, wenn Sie vor dieser Situation stehen, aber

mit generischen Typen können Sie das Problem einmal lösen und sich zukünftig die Mühe ersparen. Gleichzeitig gewährleisten Sie Typsicherheit zur Übersetzungszeit.

[21] Eine solche Klasse implementiert ein sogenanntes „Tupel“, das heißt eine Anzahl von Objekten (Komponenten), die wiederum in einem einzigen Objekt (dem Tupel) verpackt sind. Der Empfänger eines solchen Objektes darf die Komponenten abfragen aber nicht ändern. (Das Entwurfsmuster *Data-Transfer-Object* beschreibt dieses Konzept.)

[22] Tupel können beliebig lang sein, wobei jede Komponente des Tupels einem anderen Typ angehören kann. Wir wollen den Typ jeder Komponente angeben und sicherstellen, daß der Empfänger beim Abfragen der Komponenten jeweils den richtigen Typ erhält. Wir lösen das Problem der unterschiedlichen Längen, indem wir mehrere Tupelklassen anlegen. Die erste Klasse hat zwei Komponenten:

```
//: net/mindview/util/TwoTuple.java
package net.mindview.util;

public class TwoTuple<A,B> {
    public final A first;
    public final B second;
    public TwoTuple(A a, B b) { first = a; second = b; }
    public String toString() {
        return ”(” + first + ”, ” + second + ”)”;
    }
} ///:~
```

Der Konstruktor erwartet die zu speichernden Objekte. Die `toString()`-Methode ist der Bequemlichkeit halber vorhanden und gestattet, die Komponenten des Tupels in Form einer Liste anzuzeigen. Beachten Sie, daß ein Tupel implizit die Reihenfolge seiner Komponenten aufrecht erhält.

[23] Beim ersten Lesen fällt auf, daß das Beispiel *TwoTuple.java* einige gängige Sicherheitsregeln der Java-Programmierung zu mißachten scheint. Sollten die Felder `first` und `second` nicht als `private` deklariert und nur über die Abfragemethoden `getFirst()` und `getSecond()` erreichbar sein? Betrachten Sie, welche Sicherheit Sie in diesem Fall erhalten würden: Der Zugriff auf die von `first` und `second` referenzierten Objekte wäre möglich und keinem der beiden Felder könnte ein neues Objekt zugewiesen werden (da keine Änderungsmethoden `setFirst()` und `setSecond()` verlangt werden). Die Kennzeichnung von `first` und `second` als finale Felder liefert dieselbe Sicherheit, aber die obige Klasse ist kleiner und leichter verständlich.

[24] Ein anderer Einwand gegen den obigen Entwurf ist, daß Sie einem Programmierer, der die Klasse `TwoTuple` verwendet, vielleicht erlauben möchten, die von den Feldern `first` und `second` referenzierten Objekt auszutauschen. Es ist aber sicherer, die Klasse in der obigen Form zu belassen und den Programmierer zu zwingen, ein neues `TwoTuple`-Objekt zu erzeugen, falls ein Tupel mit neuen Komponenten benötigt wird.

[25] Die Tupel mit mehr als zwei Komponenten können per Ableitung erzeugt werden, wobei weitere Typparameter einfach hinzugefügt werden:

```
//: net/mindview/util/ThreeTuple.java
package net.mindview.util;

public class ThreeTuple<A,B,C> extends TwoTuple<A,B> {
    public final C third;
    public ThreeTuple(A a, B b, C c) {
        super(a, b);
        third = c;
    }
    public String toString() {
```

```
        return ''('' + first + '', '' + second + '', '' + third +'')'';
    }
} ///:~

//: net/mindview/util/FourTuple.java
package net.mindview.util;

public class FourTuple<A,B,C,D> extends ThreeTuple<A,B,C> {
    public final D fourth;
    public FourTuple(A a, B b, C c, D d) {
        super(a, b, c);
        fourth = d;
    }
    public String toString() {
        return ''('' + first + '', '' + second + '', '' +
            third + '', '' + fourth + '')'';
    }
} ///:~

//: net/mindview/util/FiveTuple.java
package net.mindview.util;

public class FiveTuple<A,B,C,D,E>
    extends FourTuple<A,B,C,D> {
    public final E fifth;
    public FiveTuple(A a, B b, C c, D d, E e) {
        super(a, b, c, d);
        fifth = e;
    }
    public String toString() {
        return ''('' + first + '', '' + second + '', '' +
            third + '', '' + fourth + '', '' + fifth + '')'';
    }
} ///:~
```

[26] Im folgenden Beispiel definieren wir ein Tupel der gewünschten Länge als Rückgabetyp einer Methode, erzeugen ein entsprechendes Objekt und geben es per `return`-Anweisung zurück:

```
//: generics/TupleTest.java
import net.mindview.util.*;

class Amphibian {}
class Vehicle {}

public class TupleTest {
    static TwoTuple<String, Integer> f() {
        // Autoboxing converts the int to Integer:
        return new TwoTuple<String, Integer>(''hi'', 47);
    }
    static ThreeTuple<Amphibian, String, Integer> g() {
        return new ThreeTuple<Amphibian, String, Integer>(
            new Amphibian(), ''hi'', 47);
    }
    static FourTuple<Vehicle, Amphibian, String, Integer> h() {
        return new FourTuple<Vehicle, Amphibian, String, Integer>(
            new Vehicle(), new Amphibian(), ''hi'', 47);
    }
    static FiveTuple<Vehicle, Amphibian, String, Integer, Double> k() {
        return new FiveTuple<Vehicle, Amphibian, String, Integer, Double>(
            new Vehicle(), new Amphibian(), ''hi'', 47, 11.1);
```

```
        }
        public static void main(String[] args) {
            TwoTuple<String, Integer> ttsi = f();
            System.out.println(ttsi);
            // ttsi.first = ''there''; // Compile error: final
            System.out.println(g());
            System.out.println(h());
            System.out.println(k());
        }
    } /* Output: (80% match)
          (hi, 47)
          (Amphibian@1f6a7b9, hi, 47)
          (Vehicle@35ce36, Amphibian@757aef, hi, 47)
          (Vehicle@9cab16, Amphibian@1a46e30, hi, 47, 11.1)
       *///:~
```

Generische Typen gestatten Ihnen, mühelos ein Tupel von Komponenten beliebigen Typs zu erzeugen, just by writing the expression.

[27] Die Fehlermeldung des Compilers bei der Anweisung `ttsi.first = ''there''` zeigt, wie die `final`-Deklaration der öffentlichen Felder Neuzuweisungen nach der Objekterzeugung verhindert.

[28] Die `new`-Ausdrück sind etwas länglich. Sie sehen später in diesem Kapitel, wie sich die Schreibweise mit Hilfe generischer Methoden vereinfachen läßt.

Übungsaufgabe 3: (1) Schreiben und testen Sie den generischen Typ `SixTuple`. ■

Übungsaufgabe 4: (1) Schreiben Sie das Beispiel *innerclasses/Sequence.java* so um, daß es generische Typen verwendet. ■

16.2.2 Eine Stapelspeicher-Klasse

[29] Wir betrachten nun ein etwas komplizierteres Beispiel, nämlich den traditionellen Stapelspeicher (*stack*). In Abschnitt 12.8 haben wir die Implementierung der Klasse `net.mindview.util.Stack` mit Hilfe der Klasse `LinkedList` besprochen (Beispiel *Stack.java*). Aus diesem Beispiel war ersichtlich, daß die Klasse `LinkedList` bereits alle erforderlichen Methoden besitzt, um einen Stapelspeicher zu implementieren. Die Implementierung der Stapelspeicherklasse wurde dort durch Komposition bewerkstelligt: Die generische Klasse `Stack<T>` beinhaltet ein Objekt einer anderen generischen Klasse (`LinkedList<T>`). Beachten Sie, daß in diesem Beispiel ein generischer Typ is just another type (mit wenigen Ausnahmen, die wir später betrachten).

[30] Im folgenden Beispiel stützen wir uns nicht auf die Klasse `LinkedList`, sondern implementieren die interne Verknüpfung der Listenelemente selbst:

```
//: generics/LinkedStack.java
// A stack implemented with an internal linked structure.

public class LinkedStack<T> {
    private static class Node<U> {
        U item;
        Node<U> next;
        Node() { item = null; next = null; }
        Node(U item, Node<U> next) {
            this.item = item;
            this.next = next;
        }
        boolean end() { return item == null && next == null; }
```

```
    }
    private Node<T> top = new Node<T>(); // End sentinel
    public void push(T item) {
        top = new Node<T>(item, top);
    }
    public T pop() {
        T result = top.item;
        if(!top.end())
            top = top.next;
        return result;
    }
    public static void main(String[] args) {
        LinkedStack<String> lss = new LinkedStack<String>();
        for(String s : ''Phasers on stun!''.split('' ''))
            lss.push(s);
        String s;
        while((s = lss.pop()) != null)
            System.out.println(s);
    }
} /* Output:
      stun!
      on
      Phasers
   *///:~
```

Auch die innere Klasse `Node` ist ein generischer Typ und verfügt über einen eigenen Typparameter.

[31] Im obigen Beispiel wird eine Endmarkierung verwendet, um anzeigen zu können, daß der Stapelspeicher leer ist. Die Endmarkierung wird beim Erzeugen eines `LinkedStack`-Objektes bewertet. Bei jedem Aufruf der `push()`-Methode wird ein neues `Node<T>`-Objekt erzeugt und mit seinem Vorgänger verknüpft. Bei jedem Aufruf der `pop()`-Methode wird das „oberste“ `Node<T>`-Objekt (`top.item`) zurückgegeben, intern verworfen und der Zeiger auf das nächste `Node<T>`-Objekt weitergestellt. Beim Erreichen der Endmarkierung entfällt das Weiterstellen des Zeigers. Die `pop()`-Methode gibt in diesem Fall `null` zurück, um anzuzeigen, daß der Stapelspeicher leer ist.

Übungsaufgabe 5: (2) Entfernen Sie den Typparameter der Klasse `Node` und ändern Sie die übrigen Anweisungen im Beispiel *LinkedStack.java*, um zu zeigen, daß eine innere Klasse Zugriff auf den generischen Typparameter ihrer äußeren Klasse hat. ■

16.2.3 Eine Liste mit zufällig ausgewähltem Rückgabeelement

[32] Im folgenden Beispiel für einen weiteren Behälter implementieren wir einen speziellen Listentyp, der bei jedem Aufruf der `select()`-Methode ein willkürlich gewähltes Element zurückgibt. Die Klasse hat einen generischen Typparameter, um den Listentyp mit einem beliebigen Elementtyp verwenden zu können:

```
//: generics/RandomList.java
import java.util.*;

public class RandomList<T> {
    private ArrayList<T> storage = new ArrayList<T>();
    private Random rand = new Random(47);
    public void add(T item) { storage.add(item); }
    public T select() {
        return storage.get(rand.nextInt(storage.size()));
    }
```

```
    public static void main(String[] args) {
        RandomList<String> rs = new RandomList<String>();
        for(String s: (''The quick brown fox jumped over '' +
                       ''the lazy brown dog'').split('' ''))
            rs.add(s);
        for(int i = 0; i < 11; i++)
            System.out.print(rs.select() + '' '');
    }
} /* Output:
     brown over fox quick quick dog brown The brown lazy brown
   *///:~
```

Übungsaufgabe 6: (1) Wenden Sie die Klasse `RandomList` mit zwei anderen Elementtypen an. ■

16.3 Generische Interfaces

[33] Auch Interfaces können generische Typparameter haben. Wir betrachten einen Generator als Beispiel, das heißt eine Klasse, die Objekte erzeugt. Generatoren sind eigentlich ein Spezialfall des *Factorymethod*-Entwurfsmusters. Während Sie einer Fabrikmethode typischerweise Argumente übergeben, um ein neues Objekt zu erzeugen, hat die entsprechende Generatormethode keine Parameter. Ein Generator „weiß“ ohne zusätzliche Informationen, wie ein Objekt erzeugt wird.

[34] Ein Generatorinterface deklariert typischerweise nur eine Methode, nämlich die Methode, welche neue Objekte erzeugt. Wir nennen diese Methode `next()` und ordnen das Generatorinterface *`Generator<T>`* unserer Standardbibliothek für Hilfsklassen zu:

```
//: net/mindview/util/Generator.java
// A generic interface.
package net.mindview.util;
public interface Generator<T> { T next(); } ///:~
```

Der Rückgabetyp der `next()`-Methode ist parametrisiert (`T`). Generische Typen werden bei Interfaces analog wie bei Klassen verwendet.

[35] Wir benötigen ein paar Klassen, um eine Implementierung von *`Generator<T>`* vorzuführen:

```
//: generics/coffee/Coffee.java
package generics.coffee;

public class Coffee {
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getSimpleName() + '' '' + id;
    }
} ///:~

//: generics/coffee/Latte.java
package generics.coffee;
public class Latte extends Coffee {} ///:~

//: generics/coffee/Mocha.java
package generics.coffee;
public class Mocha extends Coffee {} ///:~

//: generics/coffee/Cappuccino.java
package generics.coffee;
```

```
public class Cappuccino extends Coffee {} ///:~

//: generics/coffee/Americano.java
package generics.coffee;
public class Americano extends Coffee {} ///:~

//: generics/coffee/Breve.java
package generics.coffee;
public class Breve extends Coffee {} ///:~
```

Wir implementieren nun eine Klasse, die das Interface *`Generator<Coffee>`* implementiert und zufällig gewählte `Coffee`-Objekte zurückgibt:

```
//: generics/coffee/CoffeeGenerator.java
// Generate different types of Coffee:
package generics.coffee;
import java.util.*;
import net.mindview.util.*;

public class CoffeeGenerator
    implements Generator<Coffee>, Iterable<Coffee> {
    private Class[] types = { Latte.class, Mocha.class,
                              Cappuccino.class, Americano.class, Breve.class, };
    private static Random rand = new Random(47);
    public CoffeeGenerator() {}
    // For iteration:
    private int size = 0;
    public CoffeeGenerator(int sz) { size = sz; }
    public Coffee next() {
        try {
            return (Coffee) types[rand.nextInt(types.length)].newInstance();
            // Report programmer errors at run time:
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    class CoffeeIterator implements Iterator<Coffee> {
        int count = size;
        public boolean hasNext() { return count > 0; }
        public Coffee next() {
            count--;
            return CoffeeGenerator.this.next();
        }
        public void remove() { // Not implemented
            throw new UnsupportedOperationException();
        }
    };
    public Iterator<Coffee> iterator() {
        return new CoffeeIterator();
    }
    public static void main(String[] args) {
        CoffeeGenerator gen = new CoffeeGenerator();
        for(int i = 0; i < 5; i++)
            System.out.println(gen.next());
        for(Coffee c : new CoffeeGenerator(5))
            System.out.println(c);
    }
} /* Output:
```

```
        Americano 0
        Latte 1
        Americano 2
        Mocha 3
        Mocha 4
        Breve 5
        Americano 6
        Latte 7
        Cappuccino 8
        Cappuccino 9
    *///:~
```

Das parametrisierte *Generator<T>*-Interface garantiert, daß `next()` stets den Parametertyp zurückgibt. Die Klasse `CoffeeGenerator` implementiert außerdem das Interface *Iterable*, kann also in die erweitere `for`-Schleife eingesetzt werden. Im „Iteratormodus“ ist allerdings ein Abbruchkriterium erforderlich. Der zweite Konstruktor initialisiert das `size`-Feld, welches wiederum zur Bewertung der lokalen Variable `count` in der innere Klasse `CoffeeIterator` verwendet wird. Der Iterator bricht ab, wenn der Ausdruck `count > 0` nicht mehr `true` liefert.

[36] Als zweites Beispiel für eine Implementierung des *Generator<T>*-Interfaces liefert die folgende Klasse die Fibonaccizahlen:

```
//: generics/Fibonacci.java
// Generate a Fibonacci sequence.
import net.mindview.util.*;

public class Fibonacci implements Generator<Integer> {
    private int count = 0;
    public Integer next() { return fib(count++); }
    private int fib(int n) {
        if (n < 2) return 1;
        return fib(n-2) + fib(n-1);
    }
    public static void main(String[] args) {
        Fibonacci gen = new Fibonacci();
        for(int i = 0; i < 18; i++)
            System.out.print(gen.next() + " ");
    }
} /* Output:
      1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
   *///:~
```

Obwohl wir innerhalb und außerhalb (das heißt in `main()`) der Klasse mit `int`-Werten arbeiten wurde der Typparameter durch `Integer` ersetzt. Hier zeigt sich eine Beschränkung der generischen Typen in Java: Sie können keinen primitiven Typ anstelle des Typparameters einsetzen. Andererseits konvertiert das seit der SE 5 vorhandene Autoboxing bidirektional zwischen primitiven Typen und ihren Wrappertypen. Sie können die Wirkung des Autoboxings daran erkennen, daß in die `int`-Wert nahtlos von der Klasse angenommen beziehungsweise erzeugt werden.

[37] Wir wollen noch einen Schritt weitergehen und einen Fibonacci-Generator angeben, der auch das Interface *Iterable* implementiert. Sie könnten die Klasse neu zu schreiben und dabei *Iterable* implementieren, aber einerseits haben Sie nicht immer den Quelltext der Ausgangsklasse zur Verfügung und andererseits ist es nicht sinnvoll eine Klasse neu zu schreiben, wenn es sich verhindern läßt. Stattdessen schreiben wir eine Adapterklasse, die die gewünschte Schnittstelle hat. (Das *Adapter*-Entwurfsmuster wurde früher in diesem Buch eingeführt).

[38] Es gibt verschiedene Möglichkeiten eine Adapterklasse zu implementieren, beispielsweise durch

Ableiten der Ausgangsklasse:

```
//: generics/IterableFibonacci.java
// Adapt the Fibonacci class to make it Iterable.
import java.util.*;

public class IterableFibonacci
    extends Fibonacci implements Iterable<Integer> {
    private int n;
    public IterableFibonacci(int count) { n = count; }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            public boolean hasNext() { return n > 0; }
            public Integer next() {
                n--;
                return IterableFibonacci.this.next();
            }
            public void remove() { // Not implemented
                throw new UnsupportedOperationException();
            }
        };
    }
    public static void main(String[] args) {
        for(int i : new IterableFibonacci(18))
            System.out.print(i + " ");
    }
} /* Output:
      1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597 2584
   *///:~
```

Der Konstruktor erwartet eine Schranke, ab der die `hasNext()`-Methode `false` zurückgibt, damit die Klasse `IterableFibonacci` in die erweiterte `for`-Schleife eingesetzt werden kann.

Übungsaufgabe 7: (2) Verwenden Sie Komposition anstelle von Ableitung („Vererbung“), um eine Adapterklasse für `Fibonacci` zu schreiben, die das Interface *`Iterable`* implementiert. ■

Übungsaufgabe 8: (2) Entwickeln Sie eine Ableitungshierarchie von `StoryCharacter`s aus Ihren Lieblingsfilmen, aufgeteilt in `GoodGuys` und `BadGuys`. Schreiben Sie eine Generatorklasse für `StoryCharacter`. Orientieren Sie sich dabei an der Klasse `CoffeeGenerator`. ■

16.4 Generische Methoden

[39] Bis jetzt haben wir uns mit der Parametrisierung ganzer Klassen beschäftigt. Es ist ebenfalls möglich, Methoden einer Klasse zu parametrisieren. Ob die Klasse generisch ist oder nicht, ist für die Generizität ihrer Methoden unerheblich.

[40] Ein generischer Typparameter gestattet der Methode eine von ihrer Klasse unabhängige Variabilität. Als Richtlinie sollten Sie generische Methoden verwenden, sooft Sie können, das heißt wenn es möglich ist, eine Methode statt der gesamten Klasse generisch anzulegen, it's probably going to be clearer to do so. Ist eine Methode darüber hinaus statisch, so hat sie keinen Zugriff auf die generischen Typparameter ihrer Klasse, muß also als generische Methode entwickelt werden, wenn ihre Funktionalität Generizität verlangt.

[41] Bei der Definition einer generischen Methode geben Sie einfach vor dem Rückgabetyp eine Liste der generischen Typparameter an:

```
//: generics/GenericMethods.java
public class GenericMethods {
    public <T> void f(T x) {
        System.out.println(x.getClass().getName());
    }
    public static void main(String[] args) {
        GenericMethods gm = new GenericMethods();
        gm.f("");
        gm.f(1);
        gm.f(1.0);
        gm.f(1.0F);
        gm.f('c');
        gm.f(gm);
    }
} /* Output:
      java.lang.String
      java.lang.Integer
      java.lang.Double
      java.lang.Float
      java.lang.Character
      GenericMethods
   *///:~
```

Die Klasse `GenericMethods` ist nicht parametrisiert, obwohl eine Klasse und ihre Methoden zugleich parametrisiert sein können. In diesem Fall hat nur die Methode `f()` einen Typparameter, wie Sie an der Parameterliste vor dem Rückgabetyp der Methode erkennen können.

[42] Beachten Sie, daß Sie beim Erzeugen eines Objektes einer generischen Klasse den für den Typparameter eingesetzten Argumenttyp angeben müssen. Bei einer generischen Methode ist die Angabe dieses Typs in der Regel nicht erforderlich, da der Compiler diese Information selbst ermitteln kann. Diese Fähigkeit des Compilers wird als **Typinferenz** oder **Typherleitung** (*type argument inference*) bezeichnet. Die Aufrufe von `f()` sehen daher wie gewöhnliche Methodenaufrufe aus und `f()` scheint unendlich oft überladen zu sein. Die Methode akzeptiert sogar ein Argument vom Typ `GenericMethods`.

[43] Bei den Aufrufen von `f()` mit Argument primitiven Typs kommt Autoboxing ins Spiel und verpackt automatisch den primitiven Wert in einem Objekt des entsprechenden Wrappertyps. Tatsächlich können generische Methoden und Autoboxing Anweisungen überflüssig machen, bei denen früher die Umwandlung von Hand erforderlich war.

Übungsaufgabe 9: (1) Ändern Sie das Beispiel *GenericMethods.java* so, daß `f()` drei Argumente erwartet, wobei jedes Argument einem anderen generischen Typ angehört. ■

Übungsaufgabe 10: (1) Ändern Sie Übungsaufgabe 9 so, daß der Typ eines Argument nicht parametrisiert ist. ■

16.4.1 Wirksamer Einsatz der compilerseitigen Typherleitung

[44] Eine der über die generischen Typen bei Java geäußerten Beschwerden lautet, daß durch die Syntax noch mehr Quelltext hinzukommt als ohnehin schon vorhanden. Sehen Sie sich etwa das Beispiel *holding/MapOfList.java* aus Abschnitt 12.10 an, speziell die Zeile in der das `Map`-Objekt erzeugt wird, welches `Person`-Objekte auf `List`-Objekte abbildet:

```
Map<Person, List<? extends Pet>> petPeople =
    new HashMap<Person, List<? extends Pet>>();
```

(Dieser Gebrauch des `extends`-Schlüsselwortes sowie des Fragezeichens wird später in diesem Kapitel erklärt.) Es sieht aus, als ob Sie sich wiederholen und der Compiler eine Liste von generischen Argumenten aus der anderen herleiten können sollte. Leider ist der Compiler dazu nicht in der Lage, aber dennoch gestattet die Typherleitung bei generischen Methoden eine Vereinfachung. Beispielsweise können wir eine Hilfsklasse anlegen, deren statische Methoden Objekte der am häufigsten verwendeten Implementierungen der verschiedenen Containerinterfaces liefern:

```
//: net/mindview/util/New.java
// Utilities to simplify generic container creation
// by using type argument inference.
package net.mindview.util;
import java.util.*;

public class New {
    public static <K,V> Map<K,V> map() {
        return new HashMap<K,V>();
    }
    public static <T> List<T> list() {
        return new ArrayList<T>();
    }
    public static <T> LinkedList<T> lList() {
        return new LinkedList<T>();
    }
    public static <T> Set<T> set() {
        return new HashSet<T>();
    }
    public static <T> Queue<T> queue() {
        return new LinkedList<T>();
    }
    // Examples:
    public static void main(String[] args) {
        Map<String, List<String>> sls = New.map();
        List<String> ls = New.list();
        LinkedList<String> lls = New.lList();
        Set<String> ss = New.set();
        Queue<String> qs = New.queue();
    }
} ///:~
```

[45] Die `main()`-Methode zeigt einige Anwendungsbeispiele. Durch die Typherleitung ist die Wiederholung der generischen Parameterliste nicht mehr erforderlich. Die neue Hilfsklasse kann auf das Beispiel *holding/MapOfList.java* angewendet werden:

```
//: generics/SimplerPets.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class SimplerPets {
    public static void main(String[] args) {
        Map<Person, List<? extends Pet>> petPeople = New.map();
        // Rest of the code is the same...
    }
} ///:~
```

Obwohl ein interessantes Beispiel für Typherleitung, ist es schwierig, zu sagen, welchen Nutzen Sie tatsächlich davon haben. Wer diesen Quelltext liest, muß auch die obige Hilfsklasse durchsehen und ihre Auswirkungen verstehen, so daß die hier vorgestellte Lösung eventuell genauso produktiv

ist, als die ursprüngliche (zugegebenermaßen sich wiederholende) Definition, ironischerwise um ein Einfachheit willen. Enthielte die Java-Bibliothek allerdings eine Hilfsklasse wie das obige Beispiel *New.java*, so wäre es sinnvoll, sie zu gebrauchen.

[46] Der Mechanismus der Typherleitung funktioniert ausschließlich bei Zuweisungen. Wenn Sie beispielsweise das Ergebnis eines Aufrufs der Methode `New.map()` als Argument in einen zweiten Methodenaufruf einsetzen, versucht der Compiler *keine* Typherleitung, sondern behandelt den ersten Methodenaufruf, als sei der Rückgabewert einer Variablen vom Typ `Object` zugewiesen worden. Das folgende Beispiel scheitert:

```
//: generics/LimitsOfInference.java
import typeinfo.pets.*;
import java.util.*;

public class LimitsOfInference {
    static void
        f(Map<Person, List<? extends Pet>> petPeople) {}
    public static void main(String[] args) {
        // f(New.map()); // Does not compile
    }
} ///:~
```

Übungsaufgabe 11: (1) Testen Sie die Hilfsklasse `New` aus dem Beispiel *New.java*, by creating your own classes and ensuring that New will work properly with them. ■

16.4.1.1 Explizite Typangabe

[47] Es ist möglich, in einer generischen Methode den Typ explizit anzugeben, obwohl diese Syntax nur selten benötigt wird. Sie notieren dazu den Typ in spitzen Klammern nach dem Punkt und unmittelbar vor dem Methodennamen. Wenn Sie die Methoden innerhalb ihrer eigenen Klasse aufrufen, geht dem Punkt die Selbstreferenz `this` voraus. Bei statischen Methoden steht vor dem Punkt der Klassennamen. Das Problem aus dem vorigen Beispiel (*LimitsOfInference.java*) läßt sich folgendermaßen lösen:

```
//: generics/ExplicitTypeSpecification.java
import typeinfo.pets.*;
import java.util.*;
import net.mindview.util.*;

public class ExplicitTypeSpecification {
    static void f(Map<Person, List<Pet>> petPeople) {}
    public static void main(String[] args) {
        f(New.<Person, List<Pet>>map());
    }
} ///:~
```

Der Vorteil der Hilfsklasse `New`, weniger schreiben zu müssen, ist natürlich dahin. Andererseits ist die zusätzliche Syntax nur dann erforderlich, wenn Sie *keine* Zuweisung definieren.

Übungsaufgabe 12: (1) Wiederholen Sie Übungsaufgabe 11 mit expliziter Typangabe. ■

16.4.2 Argumentlisten variabler Länge und generische Methoden

[48] Generische Methoden und Argumentlisten variabler Länge koexistieren friedlich miteinander:

```
//: generics/GenericVarargs.java
import java.util.*;

public class GenericVarargs {
    public static <T> List<T> makeList(T... args) {
        List<T> result = new ArrayList<T>();
        for(T item : args)
            result.add(item);
        return result;
    }
    public static void main(String[] args) {
        List<String> ls = makeList("A");
        System.out.println(ls);
        ls = makeList("A", "B", "C");
        System.out.println(ls);
        ls = makeList("ABCDEFFHIJKLMNOPQRSTUVWXYZ".split(""));
        System.out.println(ls);
    }
} /* Output:
      [A]
      [A, B, C]
      [, A, B, C, D, E, F, F, H, I, J, K, L, M, N, O, P, Q, R, S, T, U, V,
      W, X, Y, Z]
   *///:~
```

Die Funktionalität der `makeList()`-Methode ist identisch mit der Funktionalität der statischen `Arrays`-Methode `asList()`.

16.4.3 A Generic Method To Use With Generators

[49] Es ist praktisch, einen Container (Kollektion) mit Hilfe eines Generators zu füllen und sinnvoll, diese Operation generisch anzulegen:

```
//: generics/Generators.java
// A utility to use with Generators.
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;

public class Generators {
    public static <T> Collection<T>
        fill(Collection<T> coll, Generator<T> gen, int n) {
        for(int i = 0; i < n; i++)
            coll.add(gen.next());
        return coll;
    }
    public static void main(String[] args) {
        Collection<Coffee> coffee = fill(
            new ArrayList<Coffee>(), new CoffeeGenerator(), 4);
        for(Coffee c : coffee)
            System.out.println(c);
        Collection<Integer> fnumbers = fill(
            new ArrayList<Integer>(), new Fibonacci(), 12);
        for(int i : fnumbers)
            System.out.print(i + ", ");
    }
} /* Output:
```

```
        Americano 0
        Latte 1
        Americano 2
        Mocha 3
        1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144,
    *///:~
```

Beachten Sie, wie sich die generische `fill()`-Methode transparent auf die `Coffee`- beziehungsweise `Integer`-Kollektionen und Generatoren anwenden läßt.

Übungsaufgabe 10: (4) Überladen Sie die Methode `fill()`, so daß die Argumente und Rückgabetypen die *`Collection`*-Untertypen *`List`*, *`Queue`* und *`Set`* haben. Dadurch bleibt der Containertyp erhalten. Können Sie die Methode so überladen, daß zwischen *`List`* und `LinkedList` unterschieden wird? ■

16.4.4 Ein universeller Generator

[50] Die folgende Generatorklasse kann Objekte jeder Klasse erzeugen, die einen Standardkonstruktor besitzt. Die Klasse enthält eine generische Methode, die ein Objekt von sich selbst erzeugt:

```
//: net/mindview/util/BasicGenerator.java
// Automatically create a Generator, given a class
// with a default (no-arg) constructor.
package net.mindview.util;

public class BasicGenerator<T> implements Generator<T> {
    private Class<T> type;
    public BasicGenerator(Class<T> type){ this.type = type; }
    public T next() {
        try {
            // Assumes type is a public class:
            return type.newInstance();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    // Produce a Default generator given a type token:
    public static <T> Generator<T> create(Class<T> type) {
        return new BasicGenerator<T>(type);
    }
} ///:~
```

Die Klasse `BasicGenerator` ist eine elementare Implementierung des *`Generator`*-Interfaces und liefert Objekte einer beliebigen Klasse, welche die beiden folgenden Eigenschaften hat: (1) Die Klasse ist öffentlich (da `BasicGenerator` in einem separaten Package liegt, muß die fragliche Klasse öffentlich sein und darf nicht nur unter Packagezugriff stehen). (2) Die Klasse hat einen Standardkonstruktor (einen argumentlosen Konstruktor). Um ein Objekt der Klasse `BasicGenerator` zu erzeugen, rufen Sie die `create()`-Methode auf und übergeben dabei das Klassenobjekt des erwünschten Typs. Die generische Methode `create()` gestattet die Schreibweise `BasicGenerator.create(MyType.class)` anstelle der ungelenkeren Syntax `new BasicGenerator<MyType>(MyType.class)`.

[51] Die folgende Klasse hat beispielsweise einen Standardkonstruktor:

```
//: generics/CountedObject.java
public class CountedObject {
    private static long counter = 0;
    private final long id = counter++;
```

```
    public long id() { return id; }
    public String toString() { return ''CountedObject '' + id;}
} ///:~
```

Die Klasse `CountedObject` überwacht die Anzahl ihrer Objekte und gibt diese Information in ihrer `toString()`-Methode aus.

[52] Mit Hilfe der Klasse `BasicGenerator` können Sie mühelos einen Generator für `CountedObject`-Objekte erzeugen:

```
//: generics/BasicGeneratorDemo.java
import net.mindview.util.*;

public class BasicGeneratorDemo {
    public static void main(String[] args) {
        Generator<CountedObject> gen = BasicGenerator.create(CountedObject.class);
        for(int i = 0; i < 5; i++)
            System.out.println(gen.next());
    }
} /* Output:
      CountedObject 0
      CountedObject 1
      CountedObject 2
      CountedObject 3
      CountedObject 4
   *////:~
```

Sie können sehen, wie die generische Methode die zum Erzeugen des *`Generator`*-Objektes benötigte Tipparbeit reduziert. Java generics force you to pass in the Class object anyway, so daß Sie auch zur Typherleitung in der `create()`-Methode heranziehen können.

Übungsaufgabe 14: (1) Ändern Sie das Beispiel *BasicGeneratorDemo.java* so, daß Sie das das *`Generator`*-Objekt explizit erzeugen (das heißt Sie sollen den expliziten Konstruktor anstelle der generischen `create()`-Methode verwenden). ■

16.4.5 Verbesserung der Bibliothek von Tupel-Klassen

[53] Typherleitung und statisches Importieren gestatten, die Tupel aus Unterabschnitt 16.2.1 in eine universelle Bibliothek umzuschreiben. Die folgende Hilfsklasse ermöglicht das Erzeugen von Tupeln mit Hilfe überladener statischer Methoden:

```
//: net/mindview/util/Tuple.java
// Tuple library using type argument inference.
package net.mindview.util;

public class Tuple {
    public static <A,B> TwoTuple<A,B> tuple(A a, B b) {
        return new TwoTuple<A,B>(a, b);
    }
    public static <A,B,C> ThreeTuple<A,B,C> tuple(A a, B b, C c) {
        return new ThreeTuple<A,B,C>(a, b, c);
    }
    public static <A,B,C,D> FourTuple<A,B,C,D> tuple(A a, B b, C c, D d) {
        return new FourTuple<A,B,C,D>(a, b, c, d);
    }
    public static <A,B,C,D,E> FiveTuple<A,B,C,D,E> tuple(A a, B b, C c, D d, E e) {
        return new FiveTuple<A,B,C,D,E>(a, b, c, d, e);
```

```
    }
} ///:~
```

Das folgende Testprogramm *TupleTest2.java* ist eine modifizierte Version von *TupleTest.java*, um die Methoden aus der obigen `Tuple`-Klasse zu testen:

```
//: generics/TupleTest2.java
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

public class TupleTest2 {
    static TwoTuple<String,Integer> f() {
        return tuple("hi", 47);
    }
    static TwoTuple f2() { return tuple("hi", 47); }
    static ThreeTuple<Amphibian,String,Integer> g() {
        return tuple(new Amphibian(), "hi", 47);
    }
    static FourTuple<Vehicle,Amphibian,String,Integer> h() {
        return tuple(new Vehicle(), new Amphibian(), "hi", 47);
    }
    static FiveTuple<Vehicle,Amphibian,String,Integer,Double> k() {
        return tuple(new Vehicle(), new Amphibian(), "hi", 47, 11.1);
    }
    public static void main(String[] args) {
        TwoTuple<String,Integer> ttsi = f();
        System.out.println(ttsi);
        System.out.println(f2());
        System.out.println(g());
        System.out.println(h());
        System.out.println(k());
    }
} /* Output: (80% match)
      (hi, 47)
      (hi, 47)
      (Amphibian@7d772e, hi, 47)
      (Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
      (Vehicle@1a46e30, Amphibian@3e25a5, hi, 47, 11.1)
   *///:~
```

Beachten Sie, daß `f()` ein parametrisiertes, `f2()` dagegen ein unparametrisiertes `TwoTuple`-Objekt zurückgibt. Der Compiler gibt keine Warnung zu `f2()` aus, da der Rückgabewert nicht in einem „parametrisierten Kontext“ verwendet wird. In gewisser Hinsicht wird der Rückgabewert von `f2()` aufwärts in den unparametrisierten Typ `TwoTuple` „umgewandelt“. Wenn Sie den Rückgabewert von `f2()` aber einem parametrisierten Feld oder einer parametrisierten Variablen zuweisen, warnt der Compiler.

Übungsaufgabe 15: (1) Verifizieren Sie die letzte Aussage im letzten Absatz. ■

Übungsaufgabe 16: (2) Legen Sie in der Datei *Tuple.java* eine Klasse `SixTuple` an und testen Sie sie im Programm *TupleTest2.java*. ■

16.4.6 Eine Hilfsklasse für Mengenoperationen

[54] Als nächstes Beispiel für die Verwendung generischer Methoden betrachten wir einige mathematische Mengenoperationen. Diese lassen sich komfortabel als generische Methoden definieren, um auf Elemente beliebigen Typs anwendbar zu sein:

```
//: net/mindview/util/Sets.java
package net.mindview.util;
import java.util.*;

public class Sets {
    public static <T> Set<T> union(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<T>(a);
        result.addAll(b);
        return result;
    }
    public static <T> Set<T> intersection(Set<T> a, Set<T> b) {
        Set<T> result = new HashSet<T>(a);
        result.retainAll(b);
        return result;
    }
    // Subtract subset from superset:
    public static <T> Set<T> difference(Set<T> superset, Set<T> subset) {
        Set<T> result = new HashSet<T>(superset);
        result.removeAll(subset);
        return result;
    }
    // Reflexive-everything not in the intersection:
    public static <T> Set<T> complement(Set<T> a, Set<T> b) {
        return difference(union(a, b), intersection(a, b));
    }
} ///:~
```

Die ersten drei Methoden duplizieren ihr erstes Argument, indem sie die Elemente in ein neues `HashSet`-Objekt kopieren, so daß die als Argument übergebenen Mengen nicht direkt verändert werden. Der Rückgabewert ist ein neues *`Set`*-Objekt.

[55] Die vier Methoden repräsentieren mathematische Operationen auf Mengen: Die `union()`-Methode gibt ein *`Set`*-Objekt zurück, welches die Vereinigung der beiden Argumente enthält. Die `intersection()`-Methode gibt ein *`Set`*-Objekt zurück, welches den Durchschnitt der beiden Argumente enthält. Die `difference()`-Methode gibt ein *`Set`* zurück, welches die Menge `superset` *ohne* die Elemente der Menge `subset` enthält. Die `complement()`-Methode gibt ein *`Set`*-Objekt zurück, welches die Vereinigung ohne den Durchschnitt enthält. Der folgende Aufzählungstyp `Watercolors` dient als Grundlage für ein einfaches Beispiel, um die Wirkung der vier `Sets`-Methoden vorzuführen:

```
//: generics/watercolors/Watercolors.java
package generics.watercolors;

public enum Watercolors {
    ZINC, LEMON_YELLOW, MEDIUM_YELLOW, DEEP_YELLOW, ORANGE,
    BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
    CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE,
    COBALT_BLUE_HUE, PERMANENT_GREEN, VIRIDIAN_HUE,
    SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA, RAW_UMBER,
    BURNT_UMBER, PAYNES_GRAY, IVORY_BLACK
} ///:~
```

[56] Der Aufzählungstyp wird statisch in das folgende Beispiel importiert, um die Qualifizierung der Namen fortlassen zu können. Das Beispiel verwendet die Klasse `EnumSet`, eine seit der SE 5 vorhandene Hilfsklasse, mit der Sie mühelos Mengen aus den Elementen eines Aufzählungstyps erzeugen können. (In Kapitel 20 lernen Sie mehr über die Klasse `EnumSet`.) Die statische `EnumSet`-Methode `range()` erhält das erste und das letzte Element des Bereichs von Elementen, der in der zu erzeugenden Menge liegen soll:

```
//: generics/WatercolorSets.java
import generics.watercolors.*;
import java.util.*;
import static net.mindview.util.Print.*;
import static net.mindview.util.Sets.*;
import static generics.watercolors.Watercolors.*;

public class WatercolorSets {
    public static void main(String[] args) {
        Set<Watercolors> set1 = EnumSet.range(BRILLIANT_RED, VIRIDIAN_HUE);
        Set<Watercolors> set2 = EnumSet.range(CERULEAN_BLUE_HUE, BURNT_UMBER);
        print("set1: " + set1);
        print("set2: " + set2);
        print("union(set1, set2): " + union(set1, set2));
        Set<Watercolors> subset = intersection(set1, set2);
        print("intersection(set1, set2): " + subset);
        print("difference(set1, subset): " + difference(set1, subset));
        print("difference(set2, subset): " + difference(set2, subset));
        print("complement(set1, set2): " + complement(set1, set2));
    }
} /* Output: (Sample)
      set1: [BRILLIANT_RED, CRIMSON, MAGENTA, ROSE_MADDER, VIOLET,
        CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
        PERMANENT_GREEN, VIRIDIAN_HUE]
      set2: [CERULEAN_BLUE_HUE, PHTHALO_BLUE, ULTRAMARINE, COBALT_BLUE_HUE,
        PERMANENT_GREEN, VIRIDIAN_HUE, SAP_GREEN, YELLOW_OCHRE, BURNT_SIENNA,
        RAW_UMBER, BURNT_UMBER]
      union(set1, set2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE, PERMANENT_GREEN,
        BURNT_UMBER, COBALT_BLUE_HUE, VIOLET, BRILLIANT_RED, RAW_UMBER,
        ULTRAMARINE, BURNT_SIENNA, CRIMSON, CERULEAN_BLUE_HUE, PHTHALO_BLUE,
        MAGENTA, VIRIDIAN_HUE]
      intersection(set1, set2): [ULTRAMARINE, PERMANENT_GREEN, COBALT_BLUE_HUE,
        PHTHALO_BLUE, CERULEAN_BLUE_HUE, VIRIDIAN_HUE]
      difference(set1, subset): [ROSE_MADDER, CRIMSON, VIOLET, MAGENTA,
        BRILLIANT_RED]
      difference(set2, subset): [RAW_UMBER, SAP_GREEN, YELLOW_OCHRE,
        BURNT_SIENNA, BURNT_UMBER]
      complement(set1, set2): [SAP_GREEN, ROSE_MADDER, YELLOW_OCHRE, BURNT_UMBER,
        VIOLET, BRILLIANT_RED, RAW_UMBER, BURNT_SIENNA, CRIMSON, MAGENTA]
   *///:~
```

Die sehen das Ergebnis der einzelnen Operationen an der Ausgabe.

[57] Das folgende Beispiel wendet die statische `Sets`-Methode `difference()` an, um die Unterschiede in den Methodenlisten verschiedener *`Collection`*- und *`Map`*-Klassen im Package `java.util` zu zeigen:

```
//: net/mindview/util/ContainerMethodDifferences.java
package net.mindview.util;
import java.lang.reflect.*;
import java.util.*;

public class ContainerMethodDifferences {
    static Set<String> methodSet(Class<?> type) {
        Set<String> result = new TreeSet<String>();
        for(Method m : type.getMethods())
            result.add(m.getName());
        return result;
    }
```

```
        static void interfaces(Class<?> type) {
            System.out.print("Interfaces in " + type.getSimpleName() + ": ");
            List<String> result = new ArrayList<String>();
            for(Class<?> c : type.getInterfaces())
                result.add(c.getSimpleName());
            System.out.println(result);
        }
        static Set<String> object = methodSet(Object.class);
        static { object.add("clone"); }
        static void
            difference(Class<?> superset, Class<?> subset) {
            System.out.print(superset.getSimpleName() +
                             " extends " + subset.getSimpleName() + ", adds: ");
            Set<String> comp = Sets.difference(methodSet(superset), methodSet(subset));
            comp.removeAll(object); // Don't show 'Object' methods
            System.out.println(comp);
            interfaces(superset);
        }
        public static void main(String[] args) {
            System.out.println("Collection: " + methodSet(Collection.class));
            interfaces(Collection.class);
            difference(Set.class, Collection.class);
            difference(HashSet.class, Set.class);
            difference(LinkedHashSet.class, HashSet.class);
            difference(TreeSet.class, Set.class);
            difference(List.class, Collection.class);
            difference(ArrayList.class, List.class);
            difference(LinkedList.class, List.class);
            difference(Queue.class, Collection.class);
            difference(PriorityQueue.class, Queue.class);
            System.out.println("Map: " + methodSet(Map.class));
            difference(HashMap.class, Map.class);
            difference(LinkedHashMap.class, HashMap.class);
            difference(SortedMap.class, Map.class);
            difference(TreeMap.class, Map.class);
        }
    } ///:~
```

Die Ausgabe dieses Programms wurde in Abschnitt „Zusammenfassung“ eingesetzt.

Übungsaufgabe 17: (4) Lesen sie die API-Dokumentation der Klasse `EnumSet`. Die Klasse definiert eine `clone()`-Methode. Aufgrund des Interfacetyps *`Set`* können die Rückgabewerte der Methoden in *Sets.java* nicht geklont werden. Können Sie das Beispiel *Sets.java* so ändern, daß die Methoden sowohl den allgemeinen Fall mit dem Rückgabetyp *`Set`*, als auch den Speziallfall *of an EnumSet, using clone() instead of creating a new HashSet* bedienen können? ■

16.5 Anonyme innere Klassen

[58] Generische Typen können auch bei inneren Klassen und anonymen inneren Klassen verwendet werden. Das folgende Beispiel implementiert das Interface *`Generator`* mittels einer anonymen inneren Klasse:

```
//: generics/BankTeller.java
// A very simple bank teller simulation.
import java.util.*;
```

```
import net.mindview.util.*;

class Customer {
    private static long counter = 1;
    private final long id = counter++;
    private Customer() {}
    public String toString() { return "Customer " + id; }
    // A method to produce Generator objects:
    public static Generator<Customer> generator() {
        return new Generator<Customer>() {
            public Customer next() { return new Customer(); }
        };
    }
}

class Teller {
    private static long counter = 1;
    private final long id = counter++;
    private Teller() {}
    public String toString() { return "Teller " + id; }
    // A single Generator object:
    public static Generator<Teller> generator =
        new Generator<Teller>() {
            public Teller next() { return new Teller(); }
        };
}

public class BankTeller {
    public static void serve(Teller t, Customer c) {
        System.out.println(t + " serves " + c);
    }
    public static void main(String[] args) {
        Random rand = new Random(47);
        Queue<Customer> line = new LinkedList<Customer>();
        Generators.fill(line, Customer.generator(), 15);
        List<Teller> tellers = new ArrayList<Teller>();
        Generators.fill(tellers, Teller.generator, 4);
        for(Customer c : line)
            serve(tellers.get(rand.nextInt(tellers.size())), c);
    }
} /* Output:
      Teller 3 serves Customer 1
      Teller 2 serves Customer 2
      Teller 3 serves Customer 3
      Teller 1 serves Customer 4
      Teller 1 serves Customer 5
      Teller 3 serves Customer 6
      Teller 1 serves Customer 7
      Teller 2 serves Customer 8
      Teller 3 serves Customer 9
      Teller 3 serves Customer 10
      Teller 2 serves Customer 11
      Teller 4 serves Customer 12
      Teller 2 serves Customer 13
      Teller 1 serves Customer 14
      Teller 1 serves Customer 15
   *///:~
```

Beide Klassen, `Customer` und`Teller`, haben private Konstruktoren, zwingen Sie also die Generatorobjekte beziehungsweise -methoden zu verwenden. Die `generator()`-Methode der Klasse `Customer` erzeugt bei jedem Aufruf ein neues *`Generator<Customer>`*-Objekt. Da Sie eventuell nicht mehr als ein Generatorobjekt benötigen folgt `Teller` einem anderen Ansatz und erzeugt nur ein einziges Generatorobjekt (`generator`). Die `fill()`-Methode in `main()` führt beide Ansätze in Aktion vor.

[59] Da sowohl die `generator()`-Methode der Klasse `Customer` als auch das *`Generator`*-Objekt in der Klasse `Teller` statisch sind, scheiden beide als Komponenten eines Interfaces aus, das heißt diese Notation kann nicht „generifiziert“ werden. Abgesehen davon, funktioniert sie recht gut in der `fill()`-Methode.

[60] In Unterabschnitt 22.8.1 behandeln wir andere Lösungen für das Warteschlangenproblem am Bankschalter.

Übungsaufgabe 18: (3) Entwickeln Sie ein Beispiel, in dem große Fische (`BigFish`) kleine Fische (`LittleFish`) im Ozean (`Ocean`) fressen. Orientieren Sie sich am Beispiel *BankTeller.java*. ■

16.6 Aufbau komplexer Datenstrukturen (Modelle)

[61] Ein entscheidender Vorteil der generischen Typen ist die Möglichkeit, einfach und sicher komplexe Datenstrukturen (Modelle) konstruieren zu können. Das erste Beispiel erzeugt eine Liste (*`List`*-Objekt) von Tupeln:

```
//: generics/TupleList.java
// Combining generic types to make complex generic types.
import java.util.*;
import net.mindview.util.*;

public class TupleList<A,B,C,D>
    extends ArrayList<FourTuple<A,B,C,D>> {
    public static void main(String[] args) {
        TupleList<Vehicle, Amphibian, String, Integer> tl =
            new TupleList<Vehicle, Amphibian, String, Integer>();
        tl.add(TupleTest.h());
        tl.add(TupleTest.h());
        for(FourTuple<Vehicle,Amphibian,String,Integer> i: tl)
            System.out.println(i);
    }
} /* Output: (75% match)
      (Vehicle@11b86e7, Amphibian@35ce36, hi, 47)
      (Vehicle@757aef, Amphibian@d9f9c3, hi, 47)
   *///:~
```

Wenn auch etwas langatmig (insbesondere beim Erzeugen des Iterators in der `for`-Schleife), so erhalten Sie doch ohne übertriebenen Aufwand eine relativ komplizierte Datenstruktur.

[62] Das zweite Beispiel für die einfache Konstruktion einer komplexen Datenstruktur mit Hilfe generischer Typen beschreibt ein Einzelhandelsgeschäft (*retail store*) mit Korridoren (*aisles*), Regalen (*shelves*) und Waren. Jede Klasse implementiert einen separaten Baustein und das Ganze besteht aus vielen Teilen:

```
//: generics/Store.java
// Building up a complex model using generic containers.
import java.util.*;
import net.mindview.util.*;
```

```
class Product {
    private final int id;
    private String description;
    private double price;
    public Product(int IDnumber, String descr, double price) {
        id = IDnumber;
        description = descr;
        this.price = price;
        System.out.println(toString());
    }
    public String toString() {
        return id + ": " + description + ", price: $" + price;
    }
    public void priceChange(double change) {
        price += change;
    }
    public static Generator<Product> generator =
        new Generator<Product>() {
            private Random rand = new Random(47);
            public Product next() {
                return new Product(rand.nextInt(1000), "Test",
                                   Math.round(rand.nextDouble() * 1000.0) + 0.99);
            }
        };
}

class Shelf extends ArrayList<Product> {
    public Shelf(int nProducts) {
        Generators.fill(this, Product.generator, nProducts);
    }
}

class Aisle extends ArrayList<Shelf> {
    public Aisle(int nShelves, int nProducts) {
        for(int i = 0; i < nShelves; i++)
            add(new Shelf(nProducts));
    }
}

class CheckoutStand {}
class Office {}

public class Store extends ArrayList<Aisle> {
    private ArrayList<CheckoutStand> checkouts = new ArrayList<CheckoutStand>();
    private Office office = new Office();
    public Store(int nAisles, int nShelves, int nProducts) {
        for(int i = 0; i < nAisles; i++)
            add(new Aisle(nShelves, nProducts));
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(Aisle a : this)
            for(Shelf s : a)
                for(Product p : s) {
                    result.append(p);
                    result.append("\n");
                }
        return result.toString();
    }
```

```
    public static void main(String[] args) {
        System.out.println(new Store(14, 5, 10));
    }
} /* Output:
      258: Test, price: $400.99
      861: Test, price: $160.99
      868: Test, price: $417.99
      207: Test, price: $268.99
      551: Test, price: $114.99
      278: Test, price: $804.99
      520: Test, price: $554.99
      140: Test, price: $530.99
      ...
   *///:~
```

Wie die `toString()`-Methode der Klasse `Store` dokumentiert, besteht das Ergebnis aus einer mehrfachen Schachtelung von Containern, die trotzdem typsicher und handhabbar sind. Das Beeindruckende ist, daß die Konstruktion eines solchen Modelles intellektuell nicht erschöpft.

Übungsaufgabe 19: (2) Konstruieren Sie ein Modell für ein Containerschiff. Orientieren Sie sich an der Struktur des Beispiels *Store.java*. ■

16.7 Typauslöschung

[63] Wenn Sie beginnen, sich tiefer mit den generischen Typen von Java auseinanderzusetzen, kristallisieren sich einige Dinge heraus, deren Sinnhaftigkeit sich auf den ersten Blick nicht erschließt. Beispielsweise ist die Notation `ArrayList.class` erlaubt, nicht aber `ArrayList<Integer>.class`. Betrachten Sie auch das folgende Beispiel:

```
//: generics/ErasedTypeEquivalence.java
import java.util.*;

public class ErasedTypeEquivalence {
    public static void main(String[] args) {
        Class c1 = new ArrayList<String>().getClass();
        Class c2 = new ArrayList<Integer>().getClass();
        System.out.println(c1 == c2);
    }
} /* Output:
      true
   *///:~
```

`ArrayList<String>` und `ArrayList<Integer>` können durchaus als voneinander verschiedene Typen betrachtet werden, da sie sich unterschiedlich verhalten. Beispielsweise scheitert der Versuch, ein `Integer`-Objekt in einem `ArrayList<String>`-Objekt zu speichern, während es sich einem `ArrayList<Integer>`-Objekt anstandslos übergeben läßt. Dennoch läßt das obige Programm den Eindruck entstehen, die beiden Typen seien gleich.

[64] Auch das folgende Beispiel trägt zu diesem Rätsel bei:

```
//: generics/LostInformation.java
import java.util.*;

class Frob {}
class Fnorkle {}
class Quark<Q> {}
class Particle<POSITION,MOMENTUM> {}
```

```
public class LostInformation {
    public static void main(String[] args) {
        List<Frob> list = new ArrayList<Frob>();
        Map<Frob,Fnorkle> map = new HashMap<Frob,Fnorkle>();
        Quark<Fnorkle> quark = new Quark<Fnorkle>();
        Particle<Long,Double> p = new Particle<Long,Double>();
        System.out.println(Arrays.toString(list.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(map.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(quark.getClass().getTypeParameters()));
        System.out.println(Arrays.toString(p.getClass().getTypeParameters()));
    }
} /* Output:
      [E]
      [K, V]
      [Q]
      [POSITION, MOMENTUM]
   *///:~
```

Gemäß der API-Dokumentation der Klasse `Class` gibt die Methode `getTypeParameters()` „ein Array von *`TypeVariable`*-Objekten zurück, welche die in der generische Deklaration befindlichen "type variables" und so weiter„ Diese Beschreibung erweckt den Eindruck, daß Sie mit Hilfe der `getTypeParameters()`-Methode die anstelle der generischen Typparameter eingesetzten Typen abfragen können. Wie Sie sehen, gibt die Methode aber nur die Bezeichner der Typparameter aus der Definition des generischen Typs zurück, also keine übermäßig interessante Information.

[65] Die Wahrheit ist:

> „There's no information about generic parameter types available inside generic code.“
>
> Übersetzt etwa: „Der aus einem generischen Quelltext übersetzte Bytecode enthält keinerlei Information über die gewählten Parametertypen.“

Sie können also beispielsweise den Bezeichner oder die Beschränkung eines Typparameters (siehe Abschnitt 16.9) abfragen, nicht aber den beim Erzeugen eines Objektes generischen Typs tatsächlich gewählten Parametertyp. Dies ist besonders ärgerlich, wenn Sie von C++ herkommen und die wichtigste fundamentale Eigenschaft, mit der Sie sich bei der Arbeit mit den generischen Typen von Java auseinandersetzen müssen.

[66] Die Implementierung der generischen Typen in Java beruht auf *Typauslöschung* (*type erasure*). Typauslöschung bedeutet, daß die Information, durch welchen Typ ein Typparameter eines generischen Typs ersetzt wurde, verloren geht. „Im Inneren“ eines generischen Typs wissen Sie lediglich, als daß Sie ein Objekt verwenden. So gesehen sind `ArrayList<String>` und `ArrayList<Integer>` tatsächlich vom selben Typ. Beide Varianten werden auf ihren ~~raw type~~ `ArrayList` reduziert. Das Verstehen der Typauslöschung und wie Sie damit umgehen müssen, das Thema dieses Abschnitts, ist eines der größten Hindernisse, welches Sie bei Ihrer Auseinandersetzung mit den generischen Typen bei Java zu überwinden haben.

16.7.1 Der Ansatz von C++

[67] Das folgende Beispiel ist in C++ geschrieben und verwendet ein Template. Die Syntax für parametrisierte Typen ist ähnlich, da Java viele Anregungen von C++ erhalten hat:

```
//: generics/Templates.cpp
#include <iostream>
using namespace std;
```

```
template<class T> class Manipulator {
  T obj;
public:
  Manipulator(T x) { obj = x; }
  void manipulate() { obj.f(); }
};

class HasF {
public:
  void f() { cout << ''HasF::f()'' << endl; }
};

int main() {
  HasF hf;
  Manipulator<HasF> manipulator(hf);
  manipulator.manipulate();
} /* Output:
HasF::f()
///:~
```

Die Klasse `Manipulator` speichert ein Objekt vom Typ `T`. Die interessante Stelle ist in der Methode `manipulate()`, welche *die Methode `f()` auf der Referenz `obj` aufruft*. Woher „weiß“ die Klasse `Manipulator`, daß die `f()`-Methode beim generischen Typ `T` vorhanden ist? Der Compiler von C++ prüft die Existenz von `f()`, wenn ein Objekt des Templates erzeugt wird und erkennt somit beim Erzeugen des `Manipulator<HasF>`-Objektes, daß die Klasse `HasF` eine Methode `f()` definiert. Andernfalls würde zur Übersetzungszeit ein Fehler gemeldet werden, so daß Typsicherheit gewährleistet ist.

[68] Ein derartiger Quelltext ist in C++ mühelos zu entwickeln, weil der aus einem Template übersetzte ausführbare Code beim Erzeugen eines Objektes aus dem Template den Typ der Templateparameter kennt. Für die generischen Typen bei Java gilt dies *nicht*. Wir übertragen das obige Programm von C++ nach Java. Zunächst die Klasse `HasF`:

```
//: generics/HasF.java
public class HasF {
    public void f() { System.out.println(''HasF.f()''); }
} ///:~
```

Das restliche Programm läßt sich nach der Übertragung in Java nicht übersetzen:

```
//: generics/Manipulation.java
// {CompileTimeError} (Won't compile)

class Manipulator<T> {
    private T obj;
    public Manipulator(T x) { obj = x; }
    // Error: cannot find symbol: method f():
    public void manipulate() { obj.f(); }
}

public class Manipulation {
    public static void main(String[] args) {
        HasF hf = new HasF();
        Manipulator<HasF> manipulator = new Manipulator<HasF>(hf);
        manipulator.manipulate();
    }
} ///:~
```

Aufgrund der Typauslöschung kann der Java-Compiler die Voraussetzung, daß `manipulate()` die Methode `f()` des von `ojb` referenzierten Objektes aufrufen können muß nicht auf die Tatsache abbilden, daß die Klasse `HasF` eine `f()`-Methode definiert. Um `f()` aufrufen zu können, müssen wir die

generische Klasse `Manipulator<T>` durch *Beschränkung ihres Typparameters* (*bound*) präparieren, die den Compiler anweist, nur solche Parametertypen zu akzeptieren, die unterhalb dieser Beschränkung liegen. Derartige Beschränkungen werden mit Hilfe des `extends`-Schlüsselworts ausgedrückt. Die Beschränkung bewirkt, daß sich das folgende Beispiel übersetzen läßt:

```
//: generics/Manipulator2.java
class Manipulator2<T extends HasF> {
    private T obj;
    public Manipulator2(T x) { obj = x; }
    public void manipulate() { obj.f(); }
} ///:~
```

Die Beschränkung `<T extends HasF>` besagt, daß `T` vom Typ `HasF` oder einem von `HasF` abgeleiteten Typ sein muß. Ist diese Bedingung erfüllt, so kann `f()` auf dem von `obj` referenzierten Objekt sicher aufgerufen werden.

[69] Sprechweisen: Bei `<T extends HasF>` „wird der generische Typparameter bis zu einer ersten Schranke hin ausgelöscht“ (*a generic type parameter erases to its first bound*). Ein Typparameter kann mehr als eine Schranke haben; siehe Abschnitt 16.9. Das „Auslöschungskomplement“ (im Englischen unzutreffenderweise als *erasure of the type parameter* bezeichnet) ist die Schranke, bei `<T extends HasF>` also `HasF`. Der Compiler ersetzt den Typparameter durch sein Auslöschungskomplement, hat also dieselbe Wirkung wie die Ersetzung von `T` durch `HasF` im Körper der Klasse `Manipulator2`.

[70] Eventuell haben Sie beim Beispiel *Manipulations.java* richtigerweise beobachtet, daß der generische Typ nichts zur Funktionalität der Klasse beiträgt. Sie könnten die Ersetzung ebensogut selbst vornehmen und die Klasse ohne generischen Typparameter umschreiben:

```
//: generics/Manipulator3.java
class Manipulator3 {
    private HasF obj;
    public Manipulator3(HasF x) { obj = x; }
    public void manipulate() { obj.f(); }
} ///:~
```

Dies führt zu einem wichtigen Aspekt: Die Verwendung eines generischen Typs lohnt sich nur, wenn Ihr Typparameter, „generischer“ als nur mit einem bestimmten Typ (und allen von diesem abgeleiteten Typen) bewertet wird, das heißt wenn Sie die Funktionalität über mehrere Ableitungslinien hinweg anwenden möchten. Die Deklaration und Anwendung eines Typparameters in einem brauchenbaren generischen Quelltext ist in der Regel komplizierter als das schlichte Ersetzen durch einen Klassennamen. Dennoch deutet nicht jedes Vorkommen der Notation `<T extends HasF>` auf einen Fehler hin. Hat eine Klasse beispielsweise eine Methode mit dem Rückgabetyp `T`, so besteht der Nutzen der Generizität darin, daß die Methode stets den exakten Typ zurückgibt:

```
//: generics/ReturnGenericType.java
class ReturnGenericType<T extends HasF> {
    private T obj;
    public ReturnGenericType(T x) { obj = x; }
    public T get() { return obj; }
} ///:~
```

Sie müssen den Quelltext im Ganzen betrachten und verstehen, ob er „kompliziert genug“ ist, um die Verwendung generischer Typen zu rechtfertigen. Wir behandeln die Beschränkung von Parametertypen und Platzhaltern in den Abschnitten 16.9 bis 16.12 in allen Einzelheiten.

Übungsaufgabe 20: (1) Legen Sie ein Interface an, das zwei Methoden deklariert sowie eine Klasse, die das Interface implementiert und eine zusätzliche Methode definiert. Legen Sie eine weitere

Klasse mit einer generischen Methode an, die einen durch das Interface beschränkten generischen Parameter erwartet. Zeigen Sie, daß die im Interface deklarierten Methoden in dieser generischen Methode aufgerufen werden können. Übergeben Sie in der `main()`-Methode der generischen Methode ein Objekt der ersteren Methode (die das Interface implementiert). ■

16.7.2 Migrationskompatibilität

[71] Zur Vorbeugung eventueller Verwirrung über die Typauslöschung müssen Sie sich darüber im Klaren sein, daß es *nicht* um eine Spracheigenschaft handelt. Die Typauslöschung ist ein Kompromiß bei der Implementierung der generischen Typen in Java, da sie nicht von Anfang an Teil der Sprache waren. Sie werden unter diesem Kompromiß leiden. Daher sollten Sie sich frühzeitig daran gewöhnen und verstehen, wie es zu dieser Entscheidung gekommen ist.

[72] Wären generische Typen von Java 1.0 an Bestandteil der Sprache gewesen, so wäre diese Eigenschaft nicht per Typauslöschung implementiert worden, sondern hätte Reifikation („Vergegenständlichung“, „Verdinglichung“) implementiert, um die Typparameter als Entitäten erster Klasse zu erhalten und Sie wären in die Lage versetzt worden, typbasierte Sprach- und reflective Operationen auf Typparametern auszuführen. Sie werden später in diesem Kapitel noch sehen, daß Typauslöschung die Generizität der generischen Typen einschränkt. Generische Typen sind bei Java dennoch nützlich, wenn auch nicht so nützlich als möglich und der Grund dafür ist die Typauslöschung.

[73] In einer auslöschungsbasierten Implementierung sind generische Typen Typen zweiter Klasse, die in einigen wichtigen Kontexten nicht verwendet werden können. Die generischen Typen sind nur während der statischen Typprüfung vorhanden, nach der jeder generische Typ im Programm entfernt und durch eine nicht-generische obere Schranke ersetzt wird. Beispielsweise wird `List<T>` durch `List` ausgetauscht und Typparameter werden durch `Object` ersetzt, falls keine Beschränkung festgelegt ist.

[74] Die Motivation der Entscheidung für die Typauslöschung besteht im Kern darin, daß sie einem generifizierten Quelltext gestattet, eine nicht-generifizierte Bibliothek zu nutzen und umgekehrt (häufig als „Migrationskompatibilität“ bezeichnet). Unter idealen Bedingungen würden sämtliche Bibliotheken und Anwendungen an einem einzigen Tag auf einmal generifiziert werden. In der Realität kommen die Entwickler, selbst wenn sie ausschließlich generischen Quelltext schreiben, nicht um nicht-generische Bibliotheken herum, die aus der Zeit vor der SE 5 stammen. Die Autoren dieser älteren Bibliotheken hatten vielleicht nie einen Anlaß, um ihren Quelltext zu generifizieren oder brauchen noch Zeit, um die Umstellung zu vollziehen.

[75] Java ist im Hinblick auf generische Typen also nicht nur gezwungen, Rückwärtskompatibilität zu garantieren, da existierende Klassen und Bibliotheken in ihrer Gültigkeit, Funktionalität und Bedeutung fortbestehen, sondern auch Migrationskompatibiltät zu gewährleisten, so daß Bibliotheken mit individueller Geschwindigkeit auf generische Typen umgeschrieben werden können und eine angepaßte Bibliothek die von ihr abhängigen Anwendungen und Bibliotheken nicht funktionsuntüchtig macht. Nach der Festlegung dieses Ziels entschieden sich die Entwickler von Java und die verschiedenen Gruppen, die sich mit dem Problem auseinandergesetzt hatten dafür, daß Typauslöschung der einzige gangbare Weg sei. Typauslöschung gestattet die Migration zu generischen Typen dadurch, daß nicht-generischer und generischer Quelltext koexistieren können.

[76] Stellen Sie sich zum Beispiel vor, daß eine Anwendung zwei Bibliotheken `X` und `Y` benötigt, wobei sich `Y` wiederum auf Bibliothek `Z` stützt. Nach dem Erscheinen der SE 5 werden die Entwickler dieser Anwendung und dieser Bibliothek voraussichtlich letztenendes zur Verwendung generischer Typen übergehen. Jeder dieser Entwickler hat eigene Gründe und Randbedingungen im Hinblick auf den Zeitpunkt an dem diese Migration stattfindet. Jede Bibliothek und jede Anwendung muß

hinsichtlich der Verwendung generischer Typen von allen anderen unabhängig sein, um nach der Migration Kompatibilität zu gewährleisten, darf also nicht fähig sein, festzustellen ob in einer anderen Bibliothek generische Typen vorkommen oder nicht. Folglich müssen die Anzeichen dafür „ausgelöscht“ werden, daß eine bestimmte Bibliothek generische Typen verwendet.

[77] Ohne Migrationskompatibilität würde jede im Laufe der Zeit entwickelte Bibliothek Gefahr laufen, von den Entwicklern geworfen zu werden, die sich für die Nutzung generischer Typen entschieden haben. Nachdem Bibliotheken wohl der Teil einer Programmiersprache mit den stärksten Auswirkungen auf die Produktivität sind, würde die Abkehr von einer oder mehreren Bibliotheken inakzeptable Unkosten verursachen. Ob die Typauslöschung der beste oder einzige Weg war, kann nur die Zeit ans Licht bringen.

16.7.3 Auswirkungen der Typauslöschung

[78] Die primäre Rechtfertigung für die Typauslöschung sind also der Übergang von nicht-generifiziertem zu generifiziertem Quelltext sowie die Eingliederung generischer Typen in die Sprache ohne die Funktionstüchtigkeit vorhandener Bibliotheken zu gefährden. Die Typauslöschung ermöglicht die unveränderte Weiterverwendung nicht-generischer Quelltexte solange, bis sie für die Nutzung generischer Typen umgeschrieben werden können. Diese Motivation ist nobel, da sie nicht schlagartig den gesamten vorhandenen Quelltext unbrauchbar macht.

[79] Die Typauslöschung hat andererseits auch ihren Preis. Generische Typen können nicht in Operationen verwendet werden, die sich explizit auf zur Laufzeit vorhandene Typen beziehen, beispielsweise Typumwandlungen, Vergleiche per `instanceof`-Operator und Objekterzeugung per `new`-Operator. Da alle Informationen über den Parametertyp verloren gehen, müssen Sie stets daran denken, daß Sie *nur scheinbar* Informationen über den Typ eines Parameters zur Verfügung haben. Wenn Sie zum Beispiel die folgende Klasse verwenden:

```
class Foo<T> {
    T var;
}
```

und ein `Foo`-Objekt mit dem Parametertyp `Cat` erzeugen:

```
Foo<Cat> f = new Foo<Cat>();
```

sollten eventuelle Anweisungen in der Klasse `Foo` „wissen“, daß sie nun mit einem Objekt vom Typ `Cat` arbeiten. Die Syntax deutet stark darauf hin, daß der Platzhalter `T` überall in der Klasse durch `Cat` ersetzt wird. Aber diese Ersetzung findet nicht statt und Sie müssen daran denken, daß das `var`-Feld lediglich den Typ `Object` hat, wenn Sie die Funktionalität der Klasse erweitern.

[80] Typauslöschung und Migrationskompatibilität haben außerdem zur Folge, daß die Verwendung generischer Typen nicht überall erzwungen wird, wo Sie es eventuell erwarten:

```
//: generics/ErasureAndInheritance.java
class GenericBase<T> {
    private T element;
    public void set(T arg) { arg = element; }
    public T get() { return element; }
}

class Derived1<T> extends GenericBase<T> {}

class Derived2 extends GenericBase {} // No warning

// class Derived3 extends GenericBase<?> {}
// Strange error:
```

```
// unexpected type found : ?
// required: class or interface without bounds

public class ErasureAndInheritance {
    @SuppressWarnings(''unchecked'')
    public static void main(String[] args) {
        Derived2 d2 = new Derived2();
        Object obj = d2.get();
        d2.set(obj); // Warning here!
    }
} ///:~
```

Die Klasse `Derived2` ist von `GenericBase` abgeleitet, ohne generische Parameter zu deklarieren und der Compiler gibt keine Warnung aus. Die Warnung unterbleibt bis zum Aufruf der `set()`-Methode.

[81] Java verfügt über eine Annotation, um diese Warnung zu unterdrücken (diese Annotation wurde vor der SE 5 nicht unterstützt):

```
@SuppressWarning(''unchecked'')
```

Beachten Sie, daß die Annotation bei der Methode platziert ist, welche die Warnung auslöst und nicht für die gesamte Klasse gilt. Es ist am besten, den „Geltungsbereich“ beim Unterdrücken einer Warnung so weit wie möglich einzuschränken, um nicht versehentlich durch zu breit angelegte Unterdrückung von Warnungen ein tatsächliches Problem zu verschleiern.

[82] Vermutlich bedeutet die von `Derived3` hervorgerufene Fehlermeldung, daß der Compiler eine ~~raw type~~ Basisklasse erwartet.

[83] Zusammen mit der zusätzlichen Mühe, Beschränkungen festzulegen, wenn Sie einen präziseren Parametertyp als einfach nur `Object` verwenden möchten, haben Sie, verglichen mit den parametrisierten Typen in Sprachen wie C++, Ada oder Eiffel, erheblich mehr Aufwand bei deutlich geringerem Nutzen. Das soll nicht heißen, daß Sie bei einer dieser Sprachen in der Mehrzahl der Programmierprobleme mehr Vorteile haben als mit Java, aber daß ihre Implementierungen für generische Typen flexibler und mächtiger sind, als ihr Äquivalent bei Java.

16.7.4 Typprüfung/-umwandlung beim Ein-/Austritt in eine Methode

[84] Die Typauslöschung ermöglicht den aus meiner Sicht am stärksten verwirrenden Aspekt auf dem Gebiet der generischen Typen, nämlich daß Sie Dinge darstellen können, die keine Bedeutung haben:

```
//: generics/ArrayMaker.java
import java.lang.reflect.*;
import java.util.*;

public class ArrayMaker<T> {
    private Class<T> kind;
    public ArrayMaker(Class<T> kind) { this.kind = kind; }
    @SuppressWarnings(''unchecked'')
    T[] create(int size) {
        return (T[]) Array.newInstance(kind, size);
    }
    public static void main(String[] args) {
        ArrayMaker<String> stringMaker = new ArrayMaker<String>(String.class);
        String[] stringArray = stringMaker.create(9);
        System.out.println(Arrays.toString(stringArray));
    }
```

```
} /* Output:
      [null, null, null, null, null, null, null, null, null]
   *///:~
```

Der Parametertyp des zu erzeugenden Arrays wird *scheinbar* in einem Feld vom Typ `Class<T>` gespeichert (`kind`). Die Typauslöschung bewirkt aber, daß das `kind`-Feld tatsächlich nur eine Referenz auf das Klassenobjekt des *parameterlosen* Typs `Class` speichert. Wenn Sie sich nun auf das `kind`-Feld beziehen, beispielsweise indem Sie ein Array des durch `kind` definierten Typs erzeugen, hat `Array.newInstance()` die von `kind` unterstellte Typinformation aber tatsächlich gar nicht zur Verfügung und kann das bezeichnete Ergebnis nicht liefern. Das Ergebnis erfordert daher eine Typumwandlung, die wiederum eine Warnung hervorruft, gegen die Sie nichts tun können.

[85] Beachten Sie, daß `Array.newInstance()` die zur Erzeugung von Arrays bei generischen Typen empfohlene Vorgehensweise ist.

[86] Die Situation liegt anders, wenn wir statt eines Arrays einen Container erzeugen:

```
//: generics/ListMaker.java
import java.util.*;

public class ListMaker<T> {
    List<T> create() { return new ArrayList<T>(); }
    public static void main(String[] args) {
        ListMaker<String> stringMaker = new ListMaker<String>();
        List<String> stringList = stringMaker.create();
    }
} ///:~
```

Der Compiler gibt keine Warnung aus, obwohl wir wissen, daß das `<T>` von `ArrayList<T>` in der `create()`-Methode (aufgrund der Typauslöschung) entfernt wird: Zur Laufzeit existiert `<T>` nicht mehr in der Klasse, scheint also bedeutungslos zu sein. Wenn Sie diese Bedeutungslosigkeit aber voraussetzen und `ArrayList<T>` durch `ArrayList` ersetzen, gibt der Compiler eine Warnung aus.

[87] Ist `<T>` in diesem Fall tatsächlich bedeutungslos? Was geschieht, wenn Sie einige Elemente in der Liste speichern, bevor sie sie zurückgeben:

```
//: generics/FilledListMaker.java
import java.util.*;

public class FilledListMaker<T> {
    List<T> create(T t, int n) {
        List<T> result = new ArrayList<T>();
        for(int i = 0; i < n; i++)
            result.add(t);
        return result;
    }
    public static void main(String[] args) {
        FilledListMaker<String> stringMaker = new FilledListMaker<String>();
        List<String> list = stringMaker.create("Hello", 4);
        System.out.println(list);
    }
} /* Output:
      [Hello, Hello, Hello, Hello]
   *///:~
```

[88] Obwohl der Compiler in `create()` nichts über `T` wissen kann, ist er in der Lage, zur Übersetzungszeit zu garantieren, daß jedes, der von `result` referenzierten Liste übergebene Objekt vom Typ `T` ist, also der Definition `ArrayList<T>` gehorcht. Der Compiler kann also, obwohl die Information über den eigentlichen Parametertyp im Körper einer Methode oder Klasse durch Typauslöschung

verloren geht, interne Konsistenz im Hinblick auf die Verwendung des Parametertyps in der Klasse beziehungsweise Methode garantieren.

[89] Da die Typauslöschung die Typinformation aus dem Methodenkörper entfernt, sind zur Laufzeit die *Ein- und Austrittspunkte einer Methode* ausschlaggebend. An diesen Punkten führt der Compiler Typprüfungen zur Übersetzungszeit durch und setzt Anweisungen zur Typumwandlung ein. Betrachten Sie das folgende nicht-generische Beispiel:

```
//: generics/SimpleHolder.java
public class SimpleHolder {
    private Object obj;
    public void set(Object obj) { this.obj = obj; }
    public Object get() { return obj; }
    public static void main(String[] args) {
        SimpleHolder holder = new SimpleHolder();
        holder.set("Item");
        String s = (String) holder.get();
    }
} ///:~
```

Wir dekompilieren die *.class* Datei per `javap -c SimpleHolder` (Ausgabe gekürzt):

```
public void set(java.lang.Object);
  Code:
   0: aload_0
   1: aload_1
   2: putfield #2; //Field obj:Ljava/lang/Object;
   5: return

public java.lang.Object get();
  Code:
   0: aload_0
   1: getfield #2; //Field obj:Ljava/lang/Object;
   4: areturn

public static void main(java.lang.String[]);
  Code:
   0: new #3; //class SimpleHolder
   3: dup
   4: invokespecial #4; //Method "<init>":()V
   7: astore_1
   8: aload_1
   9: ldc #5; //String Item
   11: invokevirtual #6; //Method set:(Ljava/lang/Object;)V
   14: aload_1
   15: invokevirtual #7; //Method get:()Ljava/lang/Object;
   18: checkcast #8; //class java/lang/String
   21: astore_2
   22: return

}
```

Die Methoden `set()` und `get()` speichern lediglich den Wert beziehungsweise fragen ihn ab. Die Typprüfung wird beim Aufruf der `get()`-Methode ausgeführt (Zeile 18).

[90] Wir schreiben das Beispiel nun in einen generischen Typ mit Typparameter `T` um:

```
//: generics/GenericHolder.java
public class GenericHolder<T> {
    private T obj;
```

```
    public void set(T obj) { this.obj = obj; }
    public T get() { return obj; }
    public static void main(String[] args) {
        GenericHolder<String> holder = new GenericHolder<String>();
        holder.set("Item");
        String s = holder.get();
    }
} ///:~
```

Die Typumwandlung bei `get()` ist nicht mehr notwendig, wobei wir allerdings auch wissen, daß der an `set()` übergebene Wert zur Übersetzungszeit geprüft wird. Der entsprechende Bytecode lautet (gekürzt):

```
public void set(java.lang.Object);
  Code:
   0: aload_0
   1: aload_1
   2: putfield #2; //Field obj:Ljava/lang/Object;
   5: return

public java.lang.Object get();
  Code:
   0: aload_0
   1: getfield #2; //Field obj:Ljava/lang/Object;
   4: areturn

public static void main(java.lang.String[]);
  Code:
   0: new #3; //class GenericHolder
   3: dup
   4: invokespecial #4; //Method "<init>":()V
   7: astore_1
   8: aload_1
   9: ldc #5; //String Item
   11: invokevirtual #6; //Method set:(Ljava/lang/Object;)V
   14: aload_1
   15: invokevirtual #7; //Method get:()Ljava/lang/Object;
   18: checkcast #8; //class java/lang/String
   21: astore_2
   22: return

}
```

Der Bytecode ist in beiden Fällen identisch. Die zusätzliche Arbeit, den übergebenen Typ in `set()` zu prüfen, ist umsonst, da sie vom Compiler ausgeführt wird. Die Umwandlung des von `get()` zurückgegebenen Typs ist noch immer vorhanden: Sie wird vom Compiler automatisch eingesetzt, so daß der Quelltext den Sie schreiben (und lesen) weniger geschwätzig ist.

[91] Da die Methoden `get()` und `set()` jeweils denselben Bytecode liefern folgt, daß die Einwirkungen im generischen Fall an den Ein- und Austrittspunkten der Methoden stattfindet: Sowohl die zusätzliche Prüfung der eingehenden Werte zur Übersetzungszeit als auch die Typumwandlung der ausgehenden Werte. Es ist beim Zurückdrängen der durch die Typauslöschung verursachten Verwirrung hilfreich, daran zu denken, daß die typbezogenen Eingriffe an den Ein- und Austrittspunkten der Methoden stattfinden.

16.8 Kompensieren der Typauslöschung

[92] Die Typauslöschung bewirkt, daß bestimmte Operationen in einem generischen Quelltext nicht ausgeführt werden können. Eine Operation, die zur Laufzeit die Kenntnis des exakten Typs voraussetzt, funktioniert nicht:

```
//: generics/Erased.java
// {CompileTimeError} (Won't compile)

public class Erased<T> {
    private final int SIZE = 100;
    public static void f(Object arg) {
        if(arg instanceof T) {} // Error
        T var = new T(); // Error
        T[] array = new T[SIZE]; // Error
        T[] array = (T) new Object[SIZE]; // Unchecked warning
    }
} ///:~
```

Hin und wieder können Sie um eine solche Situation „herum programmieren“, aber es gibt auch Fälle, in denen Sie die Typauslöschung mit Hilfe eines „Typ-Etiketts“ (*type tag*) kompensieren müssen, das heißt, eines Feldes, dem Sie explizit eine Referenz auf das Klassenobjekt des Parametertyps übergeben, um ihn in typabhängigen Ausdrücken verwenden zu können.

[93] Der Gebrauch des `instanceof`-Operators im obigen Beispiel (*Erased.java*) scheitert, da die Typinformation ausgelöscht wurde. Wenn Sie ein Typ-Etikett anlegen, können Sie statt dessen die `isInstance()`-Methode des Klassenobjektes aufrufen:

```
//: generics/ClassTypeCapture.java
class Building {}
class House extends Building {}

public class ClassTypeCapture<T> {
    Class<T> kind;
    public ClassTypeCapture(Class<T> kind) {
        this.kind = kind;
    }
    public boolean f(Object arg) {
        return kind.isInstance(arg);
    }
    public static void main(String[] args) {
        ClassTypeCapture<Building> ctt1 =
            new ClassTypeCapture<Building>(Building.class);
        System.out.println(ctt1.f(new Building()));
        System.out.println(ctt1.f(new House()));
        ClassTypeCapture<House> ctt2 =
            new ClassTypeCapture<House>(House.class);
        System.out.println(ctt2.f(new Building()));
        System.out.println(ctt2.f(new House()));
    }
} /* Output:
      true
      true
      false
      true
   *///:~
```

Der Compiler garantiert, daß das Typ-Etikett matches den Parametertyp.

Übungsaufgabe 21: (4) Ändern Sie das Beispiel *ClassTypeCapture.java*, in dem Sie zusätzlich eine Referenzvariable vom Typ `Map<String, Class<?>>`, eine Methode `add(String typename, Class<?> kind)` sowie eine Methode `createNew(String typename)` anlegen. Die `createNew()`-Methode erzeugt entweder ein Objekt der durch ihr `String`-Argument angegebenen Klasse oder eine Fehlermeldung. ■

16.8.1 Erzeugen eines Objektes vom Typ T

[94] Der Versuch im Beispiel *Erased.java* per `new T()` ein Objekt zu erzeugen scheitert teils aufgrund der Typauslöschung, teils aber auch, weil der Compiler nicht verifizieren kann, daß `T` einen Standardkonstruktor (einen argumentlosen Konstruktor) hat. In C++ ist diese Objekterzeugung dagegen natürlich, einfach und typsicher (Typprüfung zur Übersetzungszeit):

```
//: generics/InstantiateGenericType.cpp
// C++, not Java!

template<class T> class Foo {
  T x; // Create a field of type T
  T* y; // Pointer to T
public:
  // Initialize the pointer:
  Foo() { y = new T(); }
};

class Bar {};

int main() {
  Foo<Bar> fb;
  Foo<int> fi; // ... and it works with primitives
} ///:~
```

[95] Die Java-Lösung besteht darin, ein Fabrikobjekt zu übergeben, mit dessen Hilfe das neue Objekt erzeugt wird. Das Klassenobjekt eignet sich als Fabrikobjekt. Wenn Sie also ein Typ-Etikett anlegen, können Sie die `newInstance()`-Methode aufrufen, um ein neues Objekt des entsprechenden Typs zu erzeugen:

```
//: generics/InstantiateGenericType.java
import static net.mindview.util.Print.*;

class ClassAsFactory<T> {
    T x;
    public ClassAsFactory(Class<T> kind) {
        try {
            x = kind.newInstance();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class Employee {}

public class InstantiateGenericType {
    public static void main(String[] args) {
        ClassAsFactory<Employee> fe =
            new ClassAsFactory<Employee>(Employee.class);
        print("ClassAsFactory<Employee> succeeded");
        try {
            ClassAsFactory<Integer> fi =
```

```
                new ClassAsFactory<Integer>(Integer.class);
            } catch(Exception e) {
                print("ClassAsFactory<Integer> failed");
            }
        }
    } /* Output:
          ClassAsFactory<Employee> succeeded
          ClassAsFactory<Integer> failed
       *////:~
```

[96] Das Programm läßt sich zwar übersetzen, scheitert aber an `ClassAsFactory<Integer>`, da die Klasse `Integer` keinen Standardkonstruktor hat. Sun Microsystems mißbilligt diesen Ansatz, da der Fehler nicht zur Übersetzungszeit abgefangen wird und schlägt statt dessen vor, ein explizite Fabrikinterface anzulegen und den Parametertyp auf Klassen zu beschränken, die dieses Interface implementieren:

```
//: generics/FactoryConstraint.java
interface FactoryI<T> {
    T create();
}

class Foo2<T> {
    private T x;
    public <F extends FactoryI<T>> Foo2(F factory) {
        x = factory.create();
    }
    // ...
}

class IntegerFactory implements FactoryI<Integer> {
    public Integer create() {
        return new Integer(0);
    }
}

class Widget {
    public static class Factory implements FactoryI<Widget> {
        public Widget create() {
            return new Widget();
        }
    }
}

public class FactoryConstraint {
    public static void main(String[] args) {
        new Foo2<Integer>(new IntegerFactory());
        new Foo2<Widget>(new Widget.Factory());
    }
} ///:~
```

Dieser Ansatz ist lediglich eine Variation der Übergabe eines Klassenobjektes. Bei beiden Ansätzen wird ein Fabrikbobjekt übergeben, wobei das Klassenobjekt das eingebaute Fabrikobjekt ist, während hier ein explizites Fabrikobjekt erzeugt wird. Dafür bekommen Sie im letzten Fall Typprüfung zur Übersetzungszeit.

[97] Das Entwurfsmuster *Templatemethod* liefert einen dritten Ansatz. Im folgenden Beispiel ist `create()` die Schablonenmethode und wird in der abgeleiteten Klasse definiert, um ein Objekt dieses Typs zu erzeugen:

```
//: generics/CreatorGeneric.java
```

```
abstract class GenericWithCreate<T> {
    final T element;
    GenericWithCreate() { element = create(); }
    abstract T create();
}

class X {}

class Creator extends GenericWithCreate<X> {
    X create() { return new X(); }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
}

public class CreatorGeneric {
    public static void main(String[] args) {
        Creator c = new Creator();
        c.f();
    }
} /* Output:
      X
   *///:~
```

Übungsaufgabe 22: (6) Verwenden Sie ein Typ-Etikett und den Reflexionsmechanismus, um eine Methode anzulegen, welche die argumentbehaftete Version der `newInstance()`-Methode verwendet, um mittels eines argumentbehafteten Konstruktors ein Objekt einer Klasse zu erzeugen. ■

Übungsaufgabe 23: (1) Ändern Sie das Beispiel *FactoryConstraint.java*, so daß `create()` ein Argument erwartet. ■

Übungsaufgabe 24: (3) Ändern Sie Übungsaufgabe 21 (Seite 520) so, daß das `Map`-Objekt Fabrikobjekte anstelle von Klassenobjekten enthält. ■

16.8.2 Arrays vom generischen Typen und vom Typ T

[98] Sie können nicht ohne weiteres ein Array eines generischen Typs erzeugen (siehe *Erased.java*). Im allgemeinen können Sie an jeder Stelle an der Sie ein Array verwenden möchten, ein `ArrayList`-Objekt wählen:

```
//: generics/ListOfGenerics.java
import java.util.*;

public class ListOfGenerics<T> {
    private List<T> array = new ArrayList<T>();
    public void add(T item) { array.add(item); }
    public T get(int index) { return array.get(index); }
} ///:~
```

Sie bekommen Arrayverhalten, zusammen mit der durch generische Kollektionen geleisteten Typprüfung zur Übersetzungszeit.

[99] Ab und zu kommt es dennoch vor, daß Sie ein Array von Elementen eines generischen Typs erzeugen möchten (beispielsweise basiert `ArrayList` intern auf Arrays). Interessanterweise können Sie eine Referenzvariable so deklarieren, daß der Compiler zufrieden ist, zum Beispiel:

```
//: generics/ArrayOfGenericReference.java
class Generic<T> {}
```

```
public class ArrayOfGenericReference {
    static Generic<Integer>[] gia;
} ///:~
```

[100] Der Compiler akzeptiert diese Zeilen ohne Warnungen auszugeben. Es ist allerdings nicht möglich, ein Array von exakt diesem generischen Typ (inklusive der Typparameter) zu erzeugen, so daß die Möglichkeit eine entsprechende Referenzvariable zu deklarieren ein wenig Verwirrung stiftet. Da alle Arrays ungeachtet des Typs ihrer Elemente strukturell gleich sind (hinsichtlich der Länge ihres Elementtyps und des ~~array layout~~), müßten Sie eigentlich ein `Object`-Array erzeugen und in den gewünschten Arraytyp umwandeln können. Dieser Ansatz läßt sich tatsächlich übersetzen, scheitert aber zur Laufzeit mit einer Ausnahme vom Typ `ClassCastException`:

```
//: generics/ArrayOfGeneric.java
public class ArrayOfGeneric {
    static final int SIZE = 100;
    static Generic<Integer>[] gia;
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        // Compiles; produces ClassCastException:
        //! gia = (Generic<Integer>[]) new Object[SIZE];
        // Runtime type is the raw (erased) type:
        gia = (Generic<Integer>[]) new Generic[SIZE];
        System.out.println(gia.getClass().getSimpleName());
        gia[0] = new Generic<Integer>();
        //! gia[1] = new Object(); // Compile-time error
        // Discovers type mismatch at compile time:
        //! gia[2] = new Generic<Double>();
    }
} /* Output:
      Generic[]
   *////:~
```

[101] Das Problem besteht darin, daß ein sich Array seinen tatsächlichen Typ „merkt“, der beim Erzeugen des Arrayobjektes festgelegt wird. Obwohl das von `gia` referenzierte Objekt in den Typ `Generic<Integer>[]` umgewandelt wurde, stand diese Information nur zur Übersetzungszeit zur Verfügung (und ohne die Annotation `@SuppressWarnings` werden Sie im Hinblick auf diese Typumwandlung gewarnt). Zur Laufzeit hat das Array noch immer den Typ `Object[]` und das ist die Ursache des Problems. Die einzige Möglichkeit ein Array eines generischen Typs zu erzeugen besteht darin, ein Array des ~~raw type~~ zu erzeugen und anschließend umzuwandeln.

[102] Wir betrachten nun ein etwas anspruchsvolleres Beispiel, nämlich eine einfache Wrapperklasse um das Array:

```
//: generics/GenericArray.java
public class GenericArray<T> {
    private T[] array;
    @SuppressWarnings("unchecked")
    public GenericArray(int sz) {
        array = (T[]) new Object[sz];
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) { return array[index]; }
    // Method that exposes the underlying representation:
    public T[] rep() { return array; }
    public static void main(String[] args) {
```

```
        GenericArray<Integer> gai = new GenericArray<Integer>(10);
        // This causes a ClassCastException:
        //! Integer[] ia = gai.rep();
        // This is OK:
        Object[] oa = gai.rep();
    }
} ///:~
```

Wie zuvor, können wir nicht einfach `T[] array = new T[sz]` zuweisen, sondern erzeugen ein `Object`-Array und wandeln es um.

[103] Die Methode `rep()` gibt eine `T[]`-Referenz zurück, die in der `main()`-Methode für `gai` den Typ `Integer[]` haben sollte. Wenn Sie aber die `rep()`-Methode aufrufen und den Rückgabewert einer `Integer[]`-Referenzvariablen zuweisen, wird zur Laufzeit eine Ausnahme vom Typ `ClassCastException` ausgeworfen, da der tatsächliche Typ zur Laufzeit `Object[]` ist.

[104] Wenn Sie das Beispiel *GenericArray.java* übersetzen, nachdem Sie die Annotation `@SuppressWarnings` auskommentiert haben, gibt der Compiler die folgende Warnung aus:

```
Note: GenericArray.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Der Compiler liefert im Standardbetrieb nur diese einzelne Warnung aus und wir vermuten, daß sie sich auf die Typumwandlung bezieht. Wenn Sie sicher sein wollen, rufen Sie den Compiler nochmals auf und geben Sie dabei den Schalter `-Xlint:unchecked` an:

```
GenericArray.java:7: warning: [unchecked] unchecked cast
found   : java.lang.Object[]
required: T[]
        array = (T[])new Object[sz];
                     ^
1 warning
```

Der Compiler beschwert sich also tatsächlich über die Typumwandlung. Warnungen können ziemlich geschwätzig sein. Es ist das beste, eine Warnung mittels `@SuppressWarnings` abzustellen, nachdem Sie ihr Vorkommen nachgeprüft haben. Auf diese Weise nehmen Sie eine Warnung tatsächlich ernst, wenn sie auftritt.

[105] Aufgrund der Typauslöschung ist der Laufzeittyp des Arrays lediglich `Object[]`. Wenn wir den Typ des Arrays unmittelbar in `T[]` umwandeln, geht die Information über den eigentlichen Arraytyp zur Übersetzungszeit verloren und der Compiler läßt eventuell Fehlerprüfungen aus. Daher ist es besser, intern in der Kollektion ein `Object`-Array zu verwenden und die Typumwandlung nach `T` hinzuzufügen, wenn ein Arrayelement abgefragt wird. Wir überarbeiten das Beispiel *GenericArray.java*:

```
//: generics/GenericArray2.java
public class GenericArray2<T> {
    private Object[] array;
    public GenericArray2(int sz) {
        array = new Object[sz];
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    @SuppressWarnings("unchecked")
    public T get(int index) { return (T) array[index]; }
    @SuppressWarnings("unchecked")
    public T[] rep() {
```

```
        return (T[]) array; // Warning: unchecked cast
    }
    public static void main(String[] args) {
        GenericArray2<Integer> gai = new GenericArray2<Integer>(10);
        for(int i = 0; i < 10; i ++)
            gai.put(i, i);
        for(int i = 0; i < 10; i ++)
            System.out.print(gai.get(i) + " ");
        System.out.println();
        try {
            Integer[] ia = gai.rep();
        } catch(Exception e) { System.out.println(e); }
    }
} /* Output: (Sample)
      0 1 2 3 4 5 6 7 8 9
      java.lang.ClassCastException: [Ljava.lang.Object; cannot be cast to
        [Ljava.lang.Integer;
   */// :~
```

[106] Auf den ersten Blick fällt nur auf, daß die Typumwandlung an einer anderen Stelle stattfindet. Ohne `@SuppressWarnings` gibt der Compiler noch immer „unchecked"-Warnungen aus. Allerdings hat das Array nun den Typ `Object[]` anstelle von `T[]`. Der Rückgabewert der `get()`-Methode wird typsicher in `T` umgewandelt, da die Arrayelemente tatsächlich vom Typ `T` sind. Die `rep()`-Methode versucht, `Object[]` in `T[]` umzuwandeln. Diese Typumwandlung ist noch immer falsch und bewirkt eine Warnung zur Übersetzungs- und eine Ausnahme zur Laufzeit. Es gibt also keine Möglichkeit, den Typ des unterliegenden Arrays (`Object[]`) zu unterwandern. Das Array intern als `Object[]` und nicht als `T[]` zu behandeln, hat den Vorteil, daß weniger Gefahr besteht, daß Sie den Laufzeittyp des Arrays vergessen und versehentlich ein typfremdes Element eintragen (obwohl die meisten, vielleicht sogar alle derartigen Fehler zur Laufzeit schnell entdeckt werden).

[107] Wenn sie eine neue Klasse schreiben, sollten Sie ein Typ-Etikett übergeben. Wir überarbeiten unser Beispiel nochmals:

```
//: generics/GenericArrayWithTypeToken.java
import java.lang.reflect.*;

public class GenericArrayWithTypeToken<T> {
    private T[] array;
    @SuppressWarnings("unchecked")
    public GenericArrayWithTypeToken(Class<T> type, int sz) {
        array = (T[]) Array.newInstance(type, sz);
    }
    public void put(int index, T item) {
        array[index] = item;
    }
    public T get(int index) { return array[index]; }
    // Expose the underlying representation:
    public T[] rep() { return array; }
    public static void main(String[] args) {
        GenericArrayWithTypeToken<Integer> gai =
            new GenericArrayWithTypeToken<Integer>(Integer.class, 10);
        // This now works:
        Integer[] ia = gai.rep();
    }
} ///:~
```

Das Typ-Etikett wird per Konstruktor übergeben, um den gewählten Parametertyp nach der Typauslöschung zurückgewinnen zu können. Dadurch können wir ein Array vom eigentlich benötigten Typ erzeugen, müssen aber auch hier wieder die Warnung hinsichtlich der Typumwandlung per `@SuppressWarnings` unterdrücken. Nachdem das Typ-Etikett den eigentlichen Parametertyp gespeichert hat, können wir ihn als Rückgabetyp einsetzen und erhalten das gewünschte Ergebnis (siehe `main()`). Der Laufzeittyp des Arrays ist der exakte Typ `T[]` (hier `Integer[]`).

[108] Wenn Sie sich den Quelltext der Standardbibliotheken der SE 5 ansehen, werden Sie viele Beispiele für die Umwandlung von `Object`-Arrays in Arrays von Typparametern finden. Hier zum Beispiel der Konstruktor der Klasse `ArrayList`, der ein Argument vom Typ *`Collection`* erwartet und die Elemente der Kollektion in ein `ArrayList`-Objekt umwandelt (Kommentare entfernt, Quelltext vereinfacht):

```
public ArrayList(Collection c) {
    size = c.size();
    elementData = (E[]) new Object[size];
    c.toArray(elementData);
}
```

Sie finden viele solcher Typumwandlungen in *ArrayList.java*. Was geschieht, wenn wir *ArrayList.java* übersetzen?

```
Note: ArrayList.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Die Standardbibliothek produziert tatsächlich viele Warnungen. Wenn Sie mit C programmiert haben, vor allem mit Versionen vor ANSI-C, dann kennen Sie gewiß einen bestimmten Effekt von Warnungen: Stellt sich heraus, daß eine Warnung ignoriert werden kann, so wird sie ignoriert. Daher ist es am besten, wenn der Compiler keine Meldungen ausgibt, mit Ausnahme der Fälle, in denen der Programmier tatsächlich eingreifen muß.

[109] Neal Gafter, einer der führenden Entwickler der SE 5, macht in seinem Blog[2] darauf aufmerksam, daß er beim Überarbeiten der Bibliotheken faul war und wir nicht tun sollen, was er getan hat. Neal weist auch darauf hin, daß er einen Teil des Quelltextes der Bibliothek nicht ausbessern konnte, ohne die existierenden Schnittstellen zu verletzen. Auch wenn gewissen syntaktische Konstruktionen im Quelltext der Java-Bibliotheken vorkommen, sind sie nicht notwendig der richtige Weg. Sie können beim Lesen des Bibliotheksquelltextes also nicht voraussetzen, daß es sich um Beispiele handelt, nach denen Sie sich richten sollen.

16.9 Beschränkung von Typparametern

[110] Beschränkte Typparameter wurden bereits auf Seite 512 kurz eingeführt. Beschränkungen gestatten Ihnen, Grenzen für die Typparameter eines generischen Typs festzulegen. Auch wenn Sie durch Beschränkungen die Einhaltung von Regeln hinsichtlich der Parametertypen erzwingen können, die in Ihren generischen Typ eingesetzt werden, besteht eine noch wichtigere Auswirkung darin, daß Sie die Methoden der durch die Beschränkung definierten Teilmenge von Typen aufrufen können.

[111] Da die Typinformation durch Typauslöschung verloren geht, können Sie auf einem unbeschränkten Typparameter lediglich diejenigen Methoden aufrufen, die in der Klasse `Object` definiert sind. Wenn Sie den Typparameter allerdings auf eine Teilmenge von Typen einschränken können, so

[2] *http://gafter.blogspot.com/2004/09/puzzling-through-erasure-answer.html*

können Sie alle Methoden dieser Teilmenge aufrufen. Derartige Beschränkungen werden mit Hilfe des Schlüsselwortes `extends` festgelegt. Es ist wichtig, zu verstehen, daß `extends` im Kontext der generischen Typen eine andere Bedeutung hat, als sonst. Das folgende Beispiel definiert einige grundlegende Beschränkungen:

```
//: generics/BasicBounds.java
interface HasColor { java.awt.Color getColor(); }

class Colored<T extends HasColor> {
    T item;
    Colored(T item) { this.item = item; }
    T getItem() { return item; }
    // The bound allows you to call a method:
    java.awt.Color color() { return item.getColor(); }
}

class Dimension { public int x, y, z; }

// This won't work - class must be first, then interfaces:
// class ColoredDimension<T extends HasColor & Dimension> {

// Multiple bounds:
class ColoredDimension<T extends Dimension & HasColor> {
    T item;
    ColoredDimension(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}

interface Weight { int weight(); }

// As with inheritance, you can have only one
// concrete class but multiple interfaces:
class Solid<T extends Dimension & HasColor & Weight> {
    T item;
    Solid(T item) { this.item = item; }
    T getItem() { return item; }
    java.awt.Color color() { return item.getColor(); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
    int weight() { return item.weight(); }
}

class Bounded extends Dimension implements HasColor, Weight {
    public java.awt.Color getColor() { return null; }
    public int weight() { return 0; }
}

public class BasicBounds {
    public static void main(String[] args) {
        Solid<Bounded> solid = new Solid<Bounded>(new Bounded());
        solid.color();
        solid.getY();
        solid.weight();
    }
} ///:~
```

[112] Eventuell haben Sie beobachtet, daß das Beispiel *BasicBounds.java* einige Redundanzen aufweist, die sich mittels Ableitung beheben lassen. Im folgenden Beispiel sehen Sie, wie jede Ableitungsebene zusätzliche Beschränkungen einführt:

```
//: generics/InheritBounds.java
class HoldItem<T> {
    T item;
    HoldItem(T item) { this.item = item; }
    T getItem() { return item; }
}

class Colored2<T extends HasColor> extends HoldItem<T> {
    Colored2(T item) { super(item); }
    java.awt.Color color() { return item.getColor(); }
}

class ColoredDimension2<T extends Dimension & HasColor>
    extends Colored2<T> {
    ColoredDimension2(T item) { super(item); }
    int getX() { return item.x; }
    int getY() { return item.y; }
    int getZ() { return item.z; }
}

class Solid2<T extends Dimension & HasColor & Weight>
    extends ColoredDimension2<T> {
    Solid2(T item) { super(item); }
    int weight() { return item.weight(); }
}

public class InheritBounds {
    public static void main(String[] args) {
        Solid2<Bounded> solid2 =
            new Solid2<Bounded>(new Bounded());
        solid2.color();
        solid2.getY();
        solid2.weight();
    }
} ///:~
```

[113] Die Klasse `HoldItem` beinhaltet einfach nur ein Objekt. Dieses Verhalten geht durch Ableitung an die Klasse `Colored2` über, die außerdem verlangt, daß ihr Typparameter das Interface *`HasColor`* implementiert. Die Klassen `ColoredDimension2` und `Solid2` erweitern die Hierarchie nochmals und definieren jeweils zusätzliche Beschränkungen. Nun werden die Methoden vererbt und müssen nicht in jeder Klasse wiederholt werden.

[114] Noch ein Beispiel mit mehr Ebenen:

```
//: generics/EpicBattle.java
// Demonstrating bounds in Java generics.
import java.util.*;

interface SuperPower {}
interface XRayVision extends SuperPower {
    void seeThroughWalls();
}
interface SuperHearing extends SuperPower {
    void hearSubtleNoises();
}
interface SuperSmell extends SuperPower {
```

```
    void trackBySmell();
}

class SuperHero<POWER extends SuperPower> {
    POWER power;
    SuperHero(POWER power) { this.power = power; }
    POWER getPower() { return power; }
}

class SuperSleuth<POWER extends XRayVision>
    extends SuperHero<POWER> {
    SuperSleuth(POWER power) { super(power); }
    void see() { power.seeThroughWalls(); }
}

class CanineHero<POWER extends SuperHearing & SuperSmell>
    extends SuperHero<POWER> {
    CanineHero(POWER power) { super(power); }
    void hear() { power.hearSubtleNoises(); }
    void smell() { power.trackBySmell(); }
}

class SuperHearSmell implements SuperHearing, SuperSmell {
    public void hearSubtleNoises() {}
    public void trackBySmell() {}
}

class DogBoy extends CanineHero<SuperHearSmell> {
    DogBoy() { super(new SuperHearSmell()); }
}

public class EpicBattle {
    // Bounds in generic methods:
    static <POWER extends SuperHearing>
        void useSuperHearing(SuperHero<POWER> hero) {
        hero.getPower().hearSubtleNoises();
    }
    static <POWER extends SuperHearing & SuperSmell>
        void superFind(SuperHero<POWER> hero) {
        hero.getPower().hearSubtleNoises();
        hero.getPower().trackBySmell();
    }
    public static void main(String[] args) {
        DogBoy dogBoy = new DogBoy();
        useSuperHearing(dogBoy);
        superFind(dogBoy);
        // You can do this:
        List<? extends SuperHearing> audioBoys;
        // But you can't do this:
        // List<? extends SuperHearing & SuperSmell> dogBoys;
    }
} ///:~
```

Beachten Sie, daß der?-Platzhalter (siehe folgender Abschnitt) auf eine Beschränkung limitiert ist.

Übungsaufgabe 25: (2) Legen Sie zwei Interfaces sowie eine Klasse an, die beide Interfaces implementiert. Legen Sie zwei generische Methoden an, wobei der Typparameter der ersten Methode durch das erste und der Typparameter der zweiten Methode durch das zweite Interface beschränkt ist. Erzeugen Sie ein Objekt der Klasse, die beide Interfaces implementiert und zeigen Sie, daß dieses Objekt beiden generischen Methoden übergeben werden kann. ■

16.10 Der Fragezeichen-Platzhalter

[115] Sie haben in den Kapiteln 12 und 15 bereits einige einfache Anwendungsbeispiele für das Platzhalterzeichen „?“ bei generischen Typparametern gesehen. Dieser Abschnitt widmet sich dem Thema „Platzhalter“ mit angemessener Tiefe.

[116] Wir beginnen mit einem Beispiel, das ein bestimmtes Verhalten von Arrays zeigt: Sie können einer Referenzvariablen eines Basistyps ein Array eines von diesem Basistyp abgeleiteten Typs zuweisen:

```
//: generics/CovariantArrays.java
class Fruit {}
class Apple extends Fruit {}
class Jonathan extends Apple {}
class Orange extends Fruit {}

public class CovariantArrays {
    public static void main(String[] args) {
        Fruit[] fruit = new Apple[10];
        fruit[0] = new Apple(); // OK
        fruit[1] = new Jonathan(); // OK
        // Runtime type is Apple[], not Fruit[] or Orange[]:
        try {
            // Compiler allows you to add Fruit:
            fruit[0] = new Fruit(); // ArrayStoreException
        } catch(Exception e) { System.out.println(e); }
        try {
            // Compiler allows you to add Oranges:
            fruit[0] = new Orange(); // ArrayStoreException
        } catch(Exception e) { System.out.println(e); }
    }
} /* Output:
      java.lang.ArrayStoreException: Fruit
      java.lang.ArrayStoreException: Orange
   *///:~
```

Die erste Zeile im Körper der `main()`-Methode erzeugt ein `Apple`-Array und weist es einer Referenzvariablen vom Typ `Fruit[]` zu. Das ist vernünftig: Da `Apple` von `Fruit` abgeleitet ist, sollte ein `Apple`-Array zugleich ein `Fruit`-Array sein.

[117] Nachdem das Array eigentlich vom Typ `Apple[]` ist, sollten Sie nur Objekte vom Typ `Apple` oder einem von `Apple` abgeleiteten Typ speichern können. Dies funktioniert auch sowohl zur Übersetzungs- als auch zur Laufzeit. Beachten Sie, daß Ihnen der Compiler aber auch erlaubt, ein `Fruit`-Objekt in diesem Array zu speichern! Aus der Perspektive des Compilers ist das in Ordnung, weil eine Referenzvariable vom Typ `Fruit[]` auf das Array verweist. Warum sollte der Compiler die Speicherung eines Objektes vom Typ `Fruit` oder eines von `Fruit` abgeleiteten Typs wie `Orange` verweigern? Die Operation ist daher zur Übersetzungszeit erlaubt. Der zur Laufzeit wirksame Speicherungsmechanismus des Arrays „weiß“ dagegen, daß das Array vom Typ `Apple[]` ist und wirft beim Versuch, ein typfremdes Objekt zu speichern, eine Ausnahme vom Typ `ArrayStoreException` aus.

[118] „Aufwärts gerichtete Typumwandlung“ ist in diesem Zusammenhang ein unzutreffender Begriff. Sie weisen die Referenz auf ein Arrayobjekt einer Referenzvariablen zu. Das Verhalten eines Arrays besteht darin, Objekte zu beinhalten, but because we are able to upcast, leuchtet ein, daß Arrays die rules about the type ihrer gespeicherten Objekte erhalten. Arrays sind sich sozusagen dessen bewußt, was sie enthalten, so daß sie zwischen den Prüfungen zur Übersetzungszeit und den Prüfungen zur

Laufzeit nicht mißbraucht werden können.

[119] Diese Vereinbarung für Arrays ist nicht schlecht, da Sie zur Laufzeit *garantiert* entdecken, daß Sie ein Objekt eines unzulässigen Typs gespeichert haben. Eines der Hauptziele der generischen Typen von Java besteht aber darin, die Erkennung solcher Fehler bereits zur Übersetzungszeit zu gewährleisten. Was geschieht, wenn wir anstelle des Arrays einen generischen Container verwenden?

```
//: generics/NonCovariantGenerics.java
// {CompileTimeError} (Won't compile)
import java.util.*;

public class NonCovariantGenerics {
    // Compile Error: incompatible types:
    List<Fruit> flist = new ArrayList<Apple>();
} ///:~
```

[120] Auf den ersten Blick liest sich die Aussage: „Es ist nicht zulässig, einer Referenzvariablen vom Typ "Container vom Typ `Fruit`" einen Container vom Typ `Apple` zuzuweisen.“ Beachten Sie aber, daß generische Typen nicht alleine im Zusammenhang mit Containern vorkommen. Die Aussage lautet eigentlich: „Sie können einen generischen Container, der mit dem Parametertyp `Apple` verknüpft ist, keiner Referenzvariablen vom Typ eines generischen Containers zuweisen, der mit dem Parametertyp `Fruit` verknüpft ist.“ Wüßte der Compiler (wie bei Arrays) genug über den Quelltext, um ermitteln zu können, daß der generische Parametertyp mit einem Container verknüpft ist, so könnte er vielleicht etwas Spielraum zulassen. Der Compiler hat aber keine solche Information zur Verfügung und weigert sich daher, die „Typumwandlung“ zuzulassen. Tatsächlich liegt ohnehin keine Typumwandlung vor: Ein *`List<Apple>`*-Objekt ist kein *`List<Fruit>`*-Objekt. Ein *`List<Apple>`*-Objekt enthält Objekte vom Typ `Apple` sowie Objekte von aus `Apple` abgeleiteten Typen. Ein *`List<Fruit>`*-Objekt enthält Objekte vom Typ `Fruit` sowie Objekte von aus `Fruit` abgeleiteten Typen, darunter auch `Apple`, wird dadurch aber nicht zu einem *`List<Apple>`*-Objekt, sondern bleibt ein *`List<Fruit>`*-Objekt. Ein *`List<Apple>`*-Objekt ist nicht typäquivalent mit einem *`List<Fruit>`*-Objekt, auch wenn `Apple` ein von `Fruit` abgeleiteter Typ ist.

[121] Das Problem liegt eigentlich beim Typ des Containers und nicht beim Typ der Elemente. Im Gegensatz zu Arrays haben Container kein eingebautes Kovarianzverhalten, weil Arrays vollständig in der Programmiersprache definiert sind und daher über eingebaute Prüfungen sowohl zur Übersetzungs- als auch zur Laufzeit verfügen, während der Compiler und die Laufzeitumgebung bei generischen Typen offensichtlich nicht wissen können, was mit den Parametertypen geschehen wird und welche Regel dabei gelten sollen.

[122] Es gibt aber Situationen, in denen Sie eine Art von aufwärts gerichteter Typumwandlung zwischen beiden zulassen möchten. Diese Erlaubnis erteilen Sie mit Hilfe des `?`-Platzhalters:

```
//: generics/GenericsAndCovariance.java
import java.util.*;

public class GenericsAndCovariance {
    public static void main(String[] args) {
        // Wildcards allow covariance:
        List<? extends Fruit> flist = new ArrayList<Apple>();
        // Compile Error: can't add any type of object:
        // flist.add(new Apple());
        // flist.add(new Fruit());
        // flist.add(new Object());
        flist.add(null); // Legal but uninteresting
        // We know that it returns at least Fruit:
        Fruit f = flist.get(0);
    }
```

```
} ///:~
```

Der Typ des von `flist` referenzierten *`List`*-Objektes ist *`List<? extends Fruit>`* (lies: „Liste von Elementen deren Typ von `Fruit` abgeleitet ist). Dies bedeutet allerdings nicht, daß die Liste Elemente vom Typ `Fruit` oder einem beliebigen von `Fruit` abgeleiteten Typ enthält. Der Platzhalter (`?`) repräsentiert *einen* fest gewählten Typ, beschreibt also „einen bestimmten, bei der Deklaration der Referenzvariablen `flist` nicht angegebenen Typ". Ein dieser Referenzvariablen zugewiesenes *`List`*-Objekt muß also Elemente eines bestimmten Typs enthalten, etwa `Fruit` oder `Apple`. Durch die verallgemeinernde Typumwandlung hin zu `flist` ist dieser Typ aber bedeutungslos (*„don't actually care"*).

[123] Was können Sie mit einer Liste anfangen, wenn die einzige Einschränkung besagt, daß die Elemente vom Typ `Fruit` oder einem davon abgeleiteten Typ sein müssen, der Typ der Elemente aber tatsächlich überhaupt nicht beachtet wird? Wie können Sie typsicher Elemente in die Liste aufnehmen, wenn Sie nicht wissen, welchen Typ von Elementen die Liste beinhaltet? Wie beim aufwärts „umgewandelten" Array im Beispiel *CovariantArrays.java* können sie dies nicht. Allerdings verhindert der Compiler und nicht die Laufzeitumgebung typunsicheres Einsetzen. Sie werden früher auf das Problem aufmerksam.

[124] Sie können einwenden, daß die Dinge etwas zu weit gehen, wenn Sie nicht einmal ein `Apple`-Objekt in eine Liste aufnehmen können, über die Sie eben noch vereinbart haben, daß Sie `Apple`-Objekte enthalten soll. Das ist richtig, aber der Compiler „weiß" nichts davon. Es ist zulässig, daß eine Referenzvariable vom Typ *`List<? extends Fruit>`* auf ein Objekt vom Typ *`List<Orange>`* verweist. Nachdem Sie diese „Typumwandlung" vollzogen haben, können Sie nichts mehr in dieser Liste speichern, nicht einmal ein `Object`-Objekt.

[125] Andererseits können Sie eine Abfragemethode mit dem Rückgabetyp `Fruit` typsicher aufrufen, da ein Listenelement wenigstens vom Typ `Fruit` ist, weshalb der Compiler die Operation erlaubt.

Übungsaufgabe 26: (2) Zeigen Sie das Kovarianzverhalten von Arrays anhand der Elementtypen `Number` und `Integer`. ■

Übungsaufgabe 27: (2) Zeigen Sie anhand der Elementtypen `Number` und `Integer`, daß *`List`*-Objekte nicht kovariant sind. Führen Sie anschließend Platzhalter ein. ■

16.10.1 Wie schlau ist der Compiler?

[126] Eventuell vermuten Sie nun, daß Sie keine Methoden aufrufen können, die Argumente erwarten, aber sehen Sie sich das folgende Beispiel an:

```
//: generics/CompilerIntelligence.java
import java.util.*;

public class CompilerIntelligence {
    public static void main(String[] args) {
        List<? extends Fruit> flist = Arrays.asList(new Apple());
        Apple a = (Apple) flist.get(0); // No warning
        flist.contains(new Apple()); // Argument is 'Object'
        flist.indexOf(new Apple()); // Argument is 'Object'
    }
} ///:~
```

Sie sehen, daß sich die Methoden `contains()` und `indexOf()` mit einem Argument vom Typ `Apple` aufrufen lassen. Bedeutet das, daß der Compiler tatsächlich den Quelltext untersucht, um zu sehen, ob eine bestimmte Methode ihr Objekt modifiziert?

[127] Aus der API-Dokumentation der Klasse `ArrayList` geht hervor, daß der Compiler doch nicht so „schlau“ ist. Die `add()`-Methode erwartet ein Argument vom Typ des generischen Typparameters, `contains()` und `indexOf()` dagegen je ein Argument vom Typ `Object`. Wenn Sie den Typ der Referenzvariablen `flist` in `ArrayList<? extends Fruit>` umdeklarieren, wird auch der Typ des Argumentes von `add()` zu „`? extends Fruit`“. Der Compiler kann dieser Beschreibung nicht entnehmen, welcher von `Fruit` abgeleitete Typ dort verlangt wird und akzeptiert weder `Fruit` noch einen davon abgeleiteten Typ. Es hilft nicht, die `Apple`-Referenz vorher in `Fruit` umzuwandeln. Der Compiler verweigert einen Methodenaufruf (wie bei `add()`), wenn die Argumentliste einen Platzhalter enthält.

[128] Die Methoden `contains()` und `indexOf()` erwarten Argumente vom Typ `Object` und da keine Platzhalter involviert sind, erlaubt der Compiler die Methodenaufrufe. Das heißt, es obliegt dem Autor der generischen Klasse, zu entscheiden, welche Aufrufe „sicher“ sind und ihre Argumente vom Typ `Object` zu wählen. Wollen Sie den Methodenaufruf dagegen verbieten, wenn der Typparameter durch einen Ausdruck mit Platzhalter ersetzt wird, so verwenden Sie den Typparameter in der Parameterliste.

[129] Sie können das an der folgenden einfachen Klasse `Holder` nachvollziehen:

```
//: generics/Holder.java
public class Holder<T> {
    private T value;
    public Holder() {}
    public Holder(T val) { value = val; }
    public void set(T val) { value = val; }
    public T get() { return value; }
    public boolean equals(Object obj) {
        return value.equals(obj);
    }
    public static void main(String[] args) {
        Holder<Apple> Apple = new Holder<Apple>(new Apple());
        Apple d = Apple.get();
        Apple.set(d);
        // Holder<Fruit> Fruit = Apple; // Cannot upcast
        Holder<? extends Fruit> fruit = Apple; // OK
        Fruit p = fruit.get();
        d = (Apple) fruit.get(); // Returns 'Object'
        try {
            Orange c = (Orange) fruit.get(); // No warning
        } catch(Exception e) { System.out.println(e); }
        // fruit.set(new Apple()); // Cannot call set()
        // fruit.set(new Fruit()); // Cannot call set()
        System.out.println(fruit.equals(d)); // OK
    }
} /* Output: (Sample)
      java.lang.ClassCastException: Apple cannot be cast to Orange
      true
   *///:~
```

Die Klasse `Holder` definiert eine `set()`-Methode, die ein Argument vom Typ `T` erwartet sowie eine `equals()`-Methode, die ein Argument vom Typ `Object` erwartet. Wie Sie bereits wissen, läßt sich ein `Holder<Apple>`-Objekt *nicht* aufwärts in `Holder<Fruit>` umwandeln, wohl aber in `Holder<? extends Fruit>`. Der Rückgabetyp der `get()`-Methode ist `Fruit`, mehr „weiß“ `get()` nicht, angesichts der Beschränkung auf „einen beliebigen von `Fruit` abgeleiteten Typ“. Wenn Sie mehr über den Typ der zurückgegebenen Referenz wissen, können Sie den Rückgabewert in einen von `Fruit` abgeleiteten Typ umwandeln, ohne eine Warnung hervorzurufen, riskieren aber eine Ausnahme vom Typ `Class-`

`CastException`. Die `set()`-Methode akzeptiert als Argumenttyp weder `Apple` noch `Fruit`, da ihr Parametertyp ebenfalls „`? extends Fruit`“ ist, also alles sein kann, wobei der Compiler bei alles keine Typsicherheit gewährleisten kann.

[130] Die `equals()`-Methode funktioniert dagegen anstandslos, da sie ein Argument vom Typ `Object` anstelle von `T` erwartet. Der Compiler begnügt sich damit, auf den Typ des übergebenen beziehungsweise zurückgegebenen Objekts zu achten, statt den Quelltext dahingehend zu untersuchen, ob Sie Werte abfragen oder ändern.

16.10.2 Kontravarianz

[131] Entgegen der Ableitungsrichtung („kontravariant“) können Sie den Platzhalter durch einen Basistyp (`<? super MyClass>`) oder sogar durch einen Typparameter (`<? super T>`) beschränken. Es ist allerdings nicht möglich, einen Typparameter durch einen Basistyp zu beschränken, das heißt, die Syntax (`<T super MyClass>`) ist nicht erlaubt. Eine kontravariante Beschränkung gestattet Ihnen, einem generischen Typ Objekte zu übergeben, insbesondere also Objekte in eine generische Kollektion einzusetzen:

```
//: generics/SuperTypeWildcards.java
import java.util.*;

public class SuperTypeWildcards {
    static void writeTo(List<? super Apple> apples) {
        apples.add(new Apple());
        apples.add(new Jonathan());
        // apples.add(new Fruit()); // Error
    }
} ///:~
```

Das Argument `apples` ist eine Liste von Elementen, deren Typ ein Basistyp von `Apple` ist. Somit wissen Sie, daß es sicher ist, ein Objekt vom Typ `Apple` oder eines von `Apple` abgeleiteten Typs in die Liste aufzunehmen. Es ist andererseits nicht sicher, ein Objekt vom Typ `Fruit` in die Liste aufzunehmen, da die *untere Grenze* durch `Apple` definiert ist. Wäre die Aufnahme von `Fruit`-Objekten gestattet, so wäre die Aufnahme von Objekten möglich, die nicht dem Typ `Apple` angehören und die Typsicherheit wäre verletzt.

[132] Sie können sich die Beschränkung eines Parametertyps durch einen Unter- beziehungsweise Obertyp wie eine Lese- beziehungsweise Schreibberechtigung bezüglich einer Referenzvariablen vom Typ eines generischen Typparameters vorstellen.

[133] Kontravariante Beschränkungen lockern die Einschränkungen hinsichtlich what you can pass into a method:

```
//: generics/GenericWriting.java
import java.util.*;

public class GenericWriting {
    static <T> void writeExact(List<T> list, T item) {
        list.add(item);
    }
    static List<Apple> apples = new ArrayList<Apple>();
    static List<Fruit> fruit = new ArrayList<Fruit>();
    static void f1() {
        writeExact(apples, new Apple());
        // writeExact(fruit, new Apple()); // Error:
        // Incompatible types: found Fruit, required Apple
```

```
    }
    static <T> void writeWithWildcard(List<? super T> list, T item) {
        list.add(item);
    }
    static void f2() {
        writeWithWildcard(apples, new Apple());
        writeWithWildcard(fruit, new Apple());
    }
    public static void main(String[] args) { f1(); f2(); }
} ///:~
```

Die Methode `writeExact()` deklariert einen exakten Parametertyp (ohne Platzhalter). Wie Sie in `f1()` sehen, funktioniert `writeExact()` anstandslos, solange Sie der Liste nur `Apple`-Objekte übergeben. Andererseits trägt `writeExact()` kein `Apple`-Objekt in eine Liste von `Fruit`-Objekte ein, obwohl dieser Zugriff möglich sein sollte.

[134] Die Methode `writeWithWildcard()` deklariert ihren Listenparameter dagegen vom Typ *`List<? super T>`*, so daß das referenzierte Listenobjekt Elemente eines bestimmten von `T` abgeleiteten Typs enthält. Somit ist es sicher, den *`List`*-Methoden ein Argument vom Typ `T` oder einem von `T` abgeleiteten Typ zu übergeben. Wie Sie in `f2()` sehen, ist es wie zuvor möglich, einem *`List<Apple>`*-Objekt ein `Apple`-Objekt zu übergeben. Es ist erwartungsgemäß aber auch zulässig, ein `Apple`-Objekt in einem *`List<Fruit>`*-Objekt zu speichern.

[135] We can perform this same type of analysis as a review of covariance and wildcards:

```
//: generics/GenericReading.java
import java.util.*;

public class GenericReading {
    static <T> T readExact(List<T> list) {
        return list.get(0);
    }
    static List<Apple> apples = Arrays.asList(new Apple());
    static List<Fruit> fruit = Arrays.asList(new Fruit());
    // A static method adapts to each call:
    static void f1() {
        Apple a = readExact(apples);
        Fruit f = readExact(fruit);
        f = readExact(apples);
    }
    // If, however, you have a class, then its type is
    // established when the class is instantiated:
    static class Reader<T> {
        T readExact(List<T> list) { return list.get(0); }
    }
    static void f2() {
        Reader<Fruit> fruitReader = new Reader<Fruit>();
        Fruit f = fruitReader.readExact(fruit);
        // Fruit a = fruitReader.readExact(apples); // Error:
        // readExact(List<Fruit>) cannot be
        // applied to (List<Apple>).
    }
    static class CovariantReader<T> {
        T readCovariant(List<? extends T> list) {
            return list.get(0);
        }
    }
    static void f3() {
```

```
        CovariantReader<Fruit> fruitReader = new CovariantReader<Fruit>();
        Fruit f = fruitReader.readCovariant(fruit);
        Fruit a = fruitReader.readCovariant(apples);
    }
    public static void main(String[] args) {
        f1(); f2(); f3();
    }
} ///:~
```

Wie zuvor deklariert die erste Methode, `readExact()`, den Typ ihres `list`-Parameters exakt. Wenn Sie den Typ exakt (ohne Platzhalter) deklarieren, können Sie Elemente dieses Typs sowohl aus der Liste lesen als auch in die Liste schreiben. Außerdem paßt die statische generische Methode `readExact()` ihren Rückgabetyp an jeden einzelnen Methodenaufruf an, gibt also, mit einer *`List<Apple>`*-Referenz aufgerufen, ein `Apple`-Objekt zurück beziehungsweise mit einer *`List<Fruit>`*-Referenz aufgerufen, ein `Fruit`-Objekt. (siehe Methode `f1()`). Reicht eine statische generische Methode für Ihre Anforderungen aus, so können Sie auf eine kovariante Beschränkung verzichten, wenn Sie die Liste nur lesen.

[136] Bei einer generischen Klasse wird der Parametertyp hingegen festgelegt, wenn Sie ein Objekt der Klasse erzeugen. Die `readExact()`-Methode kann, mit der *`List<Fruit>`*-Referenzvariablen `fruit` aufgerufen, ein `Fruit`-Objekt lesen, da die Elementtypen exakt übereinstimmen. ~~But a List<Apple> should also produce Fruit objects, and the fruitReader doesn't allow this.~~

[137] Die `readCovariant()`-Methode der Klasse `CovariantReader` löst das Problem, in dem der Typ des `list`-Parameters als *`List<? extends T>`* deklariert ist. Somit können die Elemente einer passenden Liste typsicher abgefragt werden. (Sie wissen, daß jedes Element in dieser Liste dem Typ `T` oder einem von `T` abgeleiteten Typ angehört.) Die `f3()`-Methode zeigt, daß Sie nun ein `Fruit`-Objekt aus einem *`List<Apple>`*-Objekt lesen können.

Übungsaufgabe 28: (4) Legen Sie eine generische Klasse `Generic1<T>` mit einer einzelnen generischen Methode an, die ein Argument vom Typ `T` erwartet. Legen Sie eine weitere generische Klasse `Generic2<T>` mit einer einzelnen generischen Methode an, die einen Wert vom Typ `T` zurückgibt. Schreiben Sie eine generische Methode mit einem kontravarianten Argument der ersten generischen Klasse, die deren Methode aufruft. Schreiben Sie eine generische Methode mit einem kovarianten Argument der zweiten generischen Klasse, die deren Methode aufruft. Testen Sie Ihr Programm mit der `typeinfo.pets`-Bibliothek. ■

16.10.3 Unbeschränkter Fragezeichen-Platzhalter

[138] Das unbeschränkte Platzhalterzeichen (`<?>`) scheint für „alles“ zu stehen, so daß die Deklaration mit unbeschränktem Platzhalterzeichen mit der Verwendung des ~~raw type~~ gleichbedeutend zu sein scheint. In der Tat scheint sich der Compiler für's erste dieser Einschätzung anzuschließen:

```
//: generics/UnboundedWildcards1.java
import java.util.*;

public class UnboundedWildcards1 {
    static List list1;
    static List<?> list2;
    static List<? extends Object> list3;
    static void assign1(List list) {
        list1 = list;
        list2 = list;
        // list3 = list; // Warning: unchecked conversion
        // Found: List, Required: List<? extends Object>
```

```
    }
    static void assign2(List<?> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }
    static void assign3(List<? extends Object> list) {
        list1 = list;
        list2 = list;
        list3 = list;
    }
    public static void main(String[] args) {
        assign1(new ArrayList());
        assign2(new ArrayList());
        // assign3(new ArrayList()); // Warning:
        // Unchecked conversion. Found: ArrayList
        // Required: List<? extends Object>
        assign1(new ArrayList<String>());
        assign2(new ArrayList<String>());
        assign3(new ArrayList<String>());
        // Both forms are acceptable as List<?>:
        List<?> wildList = new ArrayList();
        wildList = new ArrayList<String>();
        assign1(wildList);
        assign2(wildList);
        assign3(wildList);
    }
} ///:~
```

Es gibt viele Situationen wie in diesem Beispiel, in denen der Compiler could care less, ob Sie den raw type oder <?> deklariert haben. In diesen Fällen können Sie sich <?> wie eine Dekoration vorstellen. Dennoch ist sie wertvoll, denn sie sagt aus, daß der Author bei der Deklaration die Zuweisungen eines Objektes von generischem Typ in Betracht gezogen hat, statt lediglich an den raw type zu denken, wobei der Typparameter aber völlig beliebig bewertet werden kann.

[139] Das zweite Beispiel zeigt einen anderen wichtigen Anwendungsfall für den unbeschränkte Platzhalter. Wenn Sie mehr als einen generischen Typparameter verwenden, ist es hin und wieder erforderlich, einen Typparameter frei wählbar zu lassen, während anstelle des anderen ein fester Typ eingesetzt wird:

```
//: generics/UnboundedWildcards2.java
import java.util.*;

public class UnboundedWildcards2 {
    static Map map1;
    static Map<?,?> map2;
    static Map<String,?> map3;
    static void assign1(Map map) { map1 = map; }
    static void assign2(Map<?,?> map) { map2 = map; }
    static void assign3(Map<String,?> map) { map3 = map; }
    public static void main(String[] args) {
        assign1(new HashMap());
        assign2(new HashMap());
        // assign3(new HashMap()); // Warning:
        // Unchecked conversion. Found: HashMap
        // Required: Map<String,?>
        assign1(new HashMap<String,Integer>());
        assign2(new HashMap<String,Integer>());
```

```
        assign3(new HashMap<String,Integer>());
    }
} ///:~
```

Wiederum scheint der Compiler nicht zwischen der Version mit ausschließlich unbeschränkten Platzhaltern und dem ~~raw type~~ zu unterscheiden, wie Sie an `Map<?,?>` sehen. Außerdem zeigt das erste Beispiel (*UnboundedWildcards1.java*), daß der Compiler `List<?>` und `List<? extends Object>` verschieden interpretiert.

[140] Verwirrenderweise kümmert sich der Compiler nicht immer um die Unterschiede zwischen beispielsweise `List` und `List<?>`, so daß beide Typen scheinbar gleich sind. Da ein generischer Typparameter bis zu seiner ersten Schranke hin gelöscht wird, sind `List<?>` und `List<Object>` dem Anschein nach gleichbedeutend und `List` entspricht effektiv `List<Object>`. Allerdings sind beide Aussagen nicht vollkommen wahr. Der Typ `List` repräsentiert eigentlich eine ~~raw~~ Liste, die [zugleich] Elemente vom Typ `Object` oder einem von `Object` abgeleiteten Typ enthalten kann, während der `List<?>` eine nicht-~~raw~~ Liste darstellt, die Elemente eines bestimmten, aber noch nicht bekannten Typs beinhaltet.

[141] Wann kümmert sich der Compiler tatsächlich um den Unterschied zwischen ~~raw~~ Typen und Typen mit unbeschränktem Platzhalter? Das folgende Beispiel verwendet die zuvor definierte Hilfsklasse `Holder<T>`. Die Methoden dieses Beispiels erwarten ein Argument vom Typ `Holder` in verschiedenen Formen: Als ~~raw type~~, mit einem bestimmten Parametertyp und mit unbeschränktem Platzhalter:

```
//: generics/Wildcards.java
// Exploring the meaning of wildcards.

public class Wildcards {
    // Raw argument:
    static void rawArgs(Holder holder, Object arg) {
        // holder.set(arg); // Warning:
        // Unchecked call to set(T) as a
        // member of the raw type Holder
        // holder.set(new Wildcards()); // Same warning

        // Can't do this; don't have any 'T':
        // T t = holder.get();

        // OK, but type information has been lost:
        Object obj = holder.get();
    }
    // Similar to rawArgs(), but errors instead of warnings:
    static void unboundedArg(Holder<?> holder, Object arg) {
        // holder.set(arg); // Error:
        // set(capture of ?) in Holder<capture of ?>
        // cannot be applied to (Object)
        // holder.set(new Wildcards()); // Same error

        // Can't do this; don't have any 'T':
        // T t = holder.get();

        // OK, but type information has been lost:
        Object obj = holder.get();
    }
    static <T> T exact1(Holder<T> holder) {
        T t = holder.get();
        return t;
    }
    static <T> T exact2(Holder<T> holder, T arg) {
```

```
        holder.set(arg);
        T t = holder.get();
        return t;
    }
    static <T> T wildSubtype(Holder<? extends T> holder, T arg) {
        // holder.set(arg); // Error:
        // set(capture of ? extends T) in
        // Holder<capture of ? extends T>
        // cannot be applied to (T)
        T t = holder.get();
        return t;
    }
    static <T> void wildSupertype(Holder<? super T> holder, T arg) {
        holder.set(arg);
        // T t = holder.get(); // Error:
        // Incompatible types: found Object, required T

        // OK, but type information has been lost:
        Object obj = holder.get();
    }
    public static void main(String[] args) {
        Holder raw = new Holder<Long>();
        // Or:
        raw = new Holder();
        Holder<Long> qualified = new Holder<Long>();
        Holder<?> unbounded = new Holder<Long>();
        Holder<? extends Long> bounded = new Holder<Long>();
        Long lng = 1L;

        rawArgs(raw, lng);
        rawArgs(qualified, lng);
        rawArgs(unbounded, lng);
        rawArgs(bounded, lng);

        unboundedArg(raw, lng);
        unboundedArg(qualified, lng);
        unboundedArg(unbounded, lng);
        unboundedArg(bounded, lng);

        // Object r1 = exact1(raw); // Warnings:
        // Unchecked conversion from Holder to Holder<T>
        // Unchecked method invocation: exact1(Holder<T>)
        // is applied to (Holder)
        Long r2 = exact1(qualified);
        Object r3 = exact1(unbounded); // Must return Object
        Long r4 = exact1(bounded);

        // Long r5 = exact2(raw, lng); // Warnings:
        // Unchecked conversion from Holder to Holder<Long>
        // Unchecked method invocation: exact2(Holder<T>,T)
        // is applied to (Holder,Long)
        Long r6 = exact2(qualified, lng);
        // Long r7 = exact2(unbounded, lng); // Error:
        // exact2(Holder<T>,T) cannot be applied to
        // (Holder<capture of ?>,Long)
        // Long r8 = exact2(bounded, lng); // Error:
        // exact2(Holder<T>,T) cannot be applied
        // to (Holder<capture of ? extends Long>,Long)

        // Long r9 = wildSubtype(raw, lng); // Warnings:
```

```
        // Unchecked conversion from Holder
        // to Holder<? extends Long>
        // Unchecked method invocation:
        // wildSubtype(Holder<? extends T>,T) is
        // applied to (Holder,Long)
        Long r10 = wildSubtype(qualified, lng);
        // OK, but can only return Object:
        Object r11 = wildSubtype(unbounded, lng);
        Long r12 = wildSubtype(bounded, lng);

        // wildSupertype(raw, lng); // Warnings:
        // Unchecked conversion from Holder
        // to Holder<? super Long>
        // Unchecked method invocation:
        // wildSupertype(Holder<? super T>,T)
        // is applied to (Holder,Long)
        wildSupertype(qualified, lng);
        // wildSupertype(unbounded, lng); // Error:
        // wildSupertype(Holder<? super T>,T) cannot be
        // applied to (Holder<capture of ?>,Long)
        // wildSupertype(bounded, lng); // Error:
        // wildSupertype(Holder<? super T>,T) cannot be
        // applied to (Holder<capture of ? extends Long>,Long)
    }
} ///:~
```

[142] In der Methode `rawArgs()` „weiß“ der Compiler, daß `Holder` ein generischer Typ ist. Auch wenn `Holder` als ~~raw type~~ auftritt, „weiß“ der Compiler, daß die Übergabe eines Objektes vom Typ `Object` nicht typsicher ist. Da `Holder` als ~~raw type~~ deklariert ist, können Sie ein Objekt beliebigen Typs übergeben und das Argument wird anschließend aufwärts in den Typ `Object` umgewandelt. Wenn Sie einen ~~raw type~~ als Typ einer Referenzvariablen deklarieren, geben Sie die Typprüfung zur Übersetzungszeit auf. Dasselbe gilt für `get()`: Da kein `T` vorhanden ist, kommt als Rückgabetyp nur `Object` in Frage.

[143] Es ist leicht, sich an den Gedanken zu gewöhnen, daß der ~~raw type~~ `Holder` und `Holder<?>` im Großen und Ganzen dasselbe sind. Die Methode `unboundedArg()` betont aber die Verschiedenheit dadurch, daß sie dasselbe Problem entdeckt und nicht in Form einer Warnung, sondern eines Fehlers anzeigt. Der ~~raw type~~ `Holder` kann eine Kombination von Objekten beliebiger Typen enthalten, während `Holder<?>` eine homogene Kollektion von Objekte *eines bestimmten Typs* enthält, so daß Sie nicht einfach ein Element vom Typ `Object` übergeben können.

[144] Die Parameterlisten der Methoden `exact1()` und `exact2()` deklarieren exakte generische Typparameter, keine Platzhalter. Aufgrund des zusätzlichen Parameters `arg` unterliegt `exact2()` anderen (zusätzlichen) Beschränkungen als `exact1()`.

[145] Die Parameterliste der Methode `wildSubtype()` lockert die Einschränkungen des `Holder`-Parameters, so daß ein übergebenes `Holder`-Objekt Elemente eines beliebigen von `T` abgeleiteten Typs enthalten darf. Es sei zur Wiederholung nochmals darauf hingewiesen, daß der Typparameter `T` mit `Fruit` bewertet sein kann, während `holder` ein `Holder<Apple>`-Objekt referenziert. Um zu verhindern, daß ein `Orange`-Element in ein `Holder<Apple>`-Objekt eingetragen wird, ist das Aufrufen der `set()`-Methode (oder einer anderen Methode, die ein Argument vom Typ des Parametertyps erwartet verboten. Andererseits wissen Sie, daß jedes, einem, von einer `Holder<? extends Fruit>`-Referenzvariablen referenzierten `Holder`-Objekt entnomme Element mindestens den Typ `Fruit` hat, weshalb `get()` (oder eine andere Methode, deren Rückgabetyp der Parametertyp ist) erlaubt ist.

[146] Der kontravariant begrenzte Typparameter in der Methode `wildSupertype()` bewirkt das `wildSubtype()` entgegengesetzte Verhalten: Die lokale Variable `holder` darf ein `Holder`-Objekt referenzieren, das ein Objekt eines Typs enthält, der eine Basisklasse von `T` ist. Somit kann `set()` `T`-Objekte akzeptieren, da alle bezüglich des Basistyps möglichen Operationen polymorph auf die abgeleiteten Typen übergehen (this a T). Das Aufrufen der `get()`-Methode ist dagegen not helpful, da das in einem `Holder`-Objekt gespeicherte Element einen beliebigen Basistyp haben kann, so daß nur die Typwahl `Object` sicher ist.

[147] Das Beispiel zeigt in der Methode `unboundedArg()` außerdem die bestehenden Einschränkungen, wenn die Parameterliste einen unbeschränkten Platzhalter hat: Sie können das `T`-Objekt weder abfragen noch ändern, da Sie `T` nicht zur Verfügung haben.

[148] In der `main()`-Methode sehen Sie, welche dieser Methoden welche Argumenttypen ohne Warnungen und Fehler akzeptiert. Aufgrund des Zwangs zur Migrationskompatibilität nimmt `rawArgs()` alle Variationen von `Holder`-Objekten an, ohne Warnungen auszugeben. Auch die Methode `unboundedArg()` akzeptiert sämtliche Varianten, obwohl sie sie, wie zuvor erwähnt, im Methodenkörper unterschiedlich behandelt.

[149] Wenn Sie einer Methode, die ein Objekt generischen Typs mit exakt bewertetem Typparameter (ohne Platzhalter) erwartet, eine raw type `Holder`-Referenz übergeben, gibt der Compiler eine Warnung aus, da der Methodenparameter eine Information erwartet, die beim raw type nicht vorhanden ist. Wenn Sie `exact1()` eine Referenz mit unbeschränktem Typparameter übergeben, ist keine Typinformation vorhanden, aus der der Rückgabetyp bestimmt werden kann.

[150] Wie Sie sehen, bestehen bei `exact2()` die meisten Einschränkungen, da die Methode sowohl eine Referenz auf ein `Holder<T>`-Objekt als auch eine weitere auf ein `T`-Objekt erwartet. Die Methode erzeugt solange Warnungen und Fehlermeldungen bis Sie exakte Argumente übergeben. Diese Strenge ist hin und wieder notwendig. Wenn die Einschränkungen übertrieben sind, können Sie Platzhalter verwenden, je nachdem ob Sie die Rückgabewerte oder die Argumente anhand des generischen Argumentes typisieren wollen (siehe `wildSubtype()` beziehungsweise `wildSupertype()`).

[151] Der Nutzen exakter Typisierung anstelle von Platzhaltern besteht darin, daß Sie mehr mit den Typparametern tun können. Platzhalter gestatten dagegen, eine Bandbreite parametrisierter Typen als Argument. Sie müssen von Fall zu Fall entscheiden, welcher Kompromiß besser zu Ihren Anforderungen paßt.

16.10.4 *Capture Conversion*

[152] Es gibt eine Situation, in der die Verwendung des unbeschränkten Platzhalters (`<?>`) erforderlich ist. Wenn Sie einer Methode, die einen Parameter mit unbeschränktem Platzhalter hat, ein raw type Objekt übergeben, so kann der Compiler auf den tatsächlichen Parametertyp schließen, wodurch die Methode in die Lage versetzt wird, eine weitere Methode aufrufen, die diesen exakten Parametertyp verlangt. Das folgende Beispiel führt diesen, capture conversion genannten, Effekt vor. Die Bezeichnung rührt daher, daß der nicht angegebene Platzhaltertyp abgefangen und in einen exakten Typ umgewandelt wird. Die in den Kommentaren angekündigten Warnungen treten nur auf, wenn Sie die Annotation `@SuppressWarnings` entfernen:

```
//: generics/CaptureConversion.java
public class CaptureConversion {
    static <T> void f1(Holder<T> holder) {
        T t = holder.get();
        System.out.println(t.getClass().getSimpleName());
    }
    static void f2(Holder<?> holder) {
```

```
            f1(holder); // Call with captured type
        }
        @SuppressWarnings("unchecked")
        public static void main(String[] args) {
            Holder raw = new Holder<Integer>(1);
            // f1(raw); // Produces warnings
            f2(raw); // No warnings
            Holder rawBasic = new Holder();
            rawBasic.set(new Object()); // Warning
            f2(rawBasic); // No warnings
            // Upcast to Holder<?>, still figures it out:
            Holder<?> wildcarded = new Holder<Double>(1.0);
            f2(wildcarded);
        }
    } /* Output:
          Integer
          Object
          Double
       *///:~
```

[153] Alle Vorkommen des Typparameters `T` in `f1()` sind exakt, das heißt ohne Platzhalter oder Beschränkungen. Der Typ des `holder`-Parameters in `f2()` ist `Holder<?>`, der Typparameter kann also unbeschränkt gewählt werden, ist also dem Anschein nach effektiv unbekannt. Andererseits ruft `f2()` die Methode `f1()` auf, welche einen exakt angegebenen Typparameter verlangt. Der Parametertyp wird beim Aufrufen von `f2()` abgefangen und kann somit beim Aufrufen von `f1()` übergeben werden.

[154] Vielleicht haben Sie sich überlegt, ob dieser Trick auch beim Schreiben funktioniert, wofür Sie aber einen bestimmten Typ zusammen mit `Holder<?>` übergeben müßten. *Capture conversion* funktioniert nur, wenn Sie im Methodenkörper mit dem exakten Typ arbeiten müssen. Beachten Sie, daß Sie den Typ `T` nicht aus `f2()` zurückgeben können, da `T` für `f2()` unbekannt ist. *Capture conversion* ist ein interessanter aber sehr begrenzter Effekt.

Übungsaufgabe 29: (5) Schreiben Sie eine generische Methode, die Argument vom Typ `Holder<List<?>>` erwartet. Probieren Sie aus, welche Methoden Sie auf dem `Holder`-Objekt beziehungsweise auf dem *`List`*-Objekt aufrufen können. Wiederholen Sie die Aufgabe für ein Argument vom Typ *`List<Holder<?>>`*. ■

16.11 Weitere Folgen der Eigenschaften generischer Typen

[155] Dieser Abschnitt beschreibt eine Auswahl von Effekten, die Ihnen bei der Arbeit mit den generischen Typen von Java begegnen können.

16.11.1 Keine primitive Typen als Parametertypen

[156] Eine der Einschränkungen der generischen Typen von Java besteht darin, daß Sie keine primitiven Typen als Parametertypen wählen können, das heißt, Sie können beispielsweise kein `ArrayList<int>`-Objekt erzeugen.

[157] Die Lösung besteht darin, daß Sie die Wrapperklasse des primitiven Typs verwenden und den seit der SE 5 vorhandenen Autoboxingmechanismus nutzen. Wenn Sie ein `ArrayList<Integer>`-Objekt mit `int`-Elementen „betreiben“, zeigt sich, daß Autoboxing die Umwandlungen zwischen

`int` und `Integer` in beide Richtungen automatisch bewerkstelligt, als ob Sie ein `ArrayList<int>`-Objekt hätten:

```
//: generics/ListOfInt.java
// Autoboxing compensates for the inability to use
// primitives in generics.
import java.util.*;

public class ListOfInt {
    public static void main(String[] args) {
        List<Integer> li = new ArrayList<Integer>();
        for(int i = 0; i < 5; i++)
            li.add(i);
        for(int i : li)
            System.out.print(i + " ");
    }
} /* Output:
      0 1 2 3 4
   *////:~
```

Beachten Sie, daß Autoboxing sogar im Kopf der erweiterten `for`-Schleife wirkt.

[158] Diese Lösung funktioniert und reicht in der Regel aus, um Elemente vom Typ `int` zu speichern und abzufragen. Es gibt zwar einige Umwandlungen, die aber im Verborgenen durchgeführt werden. Wenn sich die Performanz als Problem herausstellt, können Sie eine spezialisierte, an Elemente primitiven Typs angepaßte Version des Containers wählen. Sie finden eine quelloffene Implementierung solcher Container im Package `org.apache.commons.collections.primitives`.

[159] Das nächste Beispiel zeigt einen anderen Ansatz (*`Set<Byte>`*):

```
//: generics/ByteSet.java
import java.util.*;

public class ByteSet {
    Byte[] possibles = { 1,2,3,4,5,6,7,8,9 };
    Set<Byte> mySet = new HashSet<Byte>(Arrays.asList(possibles));
    // But you can't do this:
    // Set<Byte> mySet2 = new HashSet<Byte>(
    // Arrays.<Byte>asList(1,2,3,4,5,6,7,8,9));
} ///:~
```

Beachten Sie, daß Autoboxing einige, aber nicht alle Probleme löst. Das folgende Beispiel zeigt eine Anwendung des generischen Interfaces *`Generator<T>`*, welches eine `next()`-Methode deklariert, die eine Referenz auf ein Objekt des Parametertyps zurückgibt. Die generische Methode `fill()` der Klasse `FArray` ruft einen Generator auf, um ein Array mit Elementen zu füllen (die Klasse generisch anzulegen würde in diesem Fall nicht funktionieren, da die Methode statisch ist). Die Implementierungen des *`Generator`*-Interfaces stammen aus Kapitel 17. In der `main()`-Methode der Klasse `PrimitiveGenericTest` wird `FArray.fill()` aufgerufen, um Arrays zu füllen:

```
//: generics/PrimitiveGenericTest.java
import net.mindview.util.*;

// Fill an array using a generator:
class FArray {
    public static <T> T[] fill(T[] a, Generator<T> gen) {
        for(int i = 0; i < a.length; i++)
            a[i] = gen.next();
        return a;
    }
```

```
}

public class PrimitiveGenericTest {
    public static void main(String[] args) {
        String[] strings = FArray.fill(
            new String[7], new RandomGenerator.String(10));
        for(String s : strings)
            System.out.println(s);
        Integer[] integers = FArray.fill(
            new Integer[7], new RandomGenerator.Integer());
        for(int i: integers)
            System.out.println(i);
        // Autoboxing won't save you here. This won't compile:
        // int[] b =
        // FArray.fill(new int[7], new RandIntGenerator());
    }
} /* Output:
      YNzbrnyGcF
      OWZnTcQrGs
      eGZMmJMRoE
      suEcUOneOE
      dLsmwHLGEa
      hKcxrEqUCB
      bkInaMesbt
      7052
      6665
      2654
      3909
      5202
      2209
      5458
   *///:~
```

Nachdem die Klasse `RandomGenerator.Integer` das Interface *`Generator<Integer>`* implementiert, hoffte ich, daß der Rückgabewert der `next()`-Methode automatisch von `Integer` nach `int` umgewandelt werden würde. Der Autoboxingmechanismus wirkt allerdings nicht bei Arrays.

Übungsaufgabe 30: (2) Erzeugen Sie zu jedem Wrappertyp eines primitiven Typs ein `Holder`-Objekt (siehe Seite 533) und zeigen Sie, daß Autoboxing und Autounboxing bei den Methoden `set()` und `get()` jedes Objektes funktioniert. ■

16.11.2 Implementierung parametrisierter Interfaces

[160] Eine Klasse kann nicht zwei Ausprägungen desselben generischen Interfaces implementieren. Aufgrund der Typauslöschung sind beide Ausprägungen gleich. Das folgende Beispiel zeigt eine solche Kollision:

```
//: generics/MultipleInterfaceVariants.java
// {CompileTimeError} (Won't compile)

interface Payable<T> {}

class Employee implements Payable<Employee> {}
class Hourly extends Employee
    implements Payable<Hourly> {} ///:~
```

Die Klasse `Hourly` läßt sich nicht übersetzen, da *`Payable<Employee>`* und *`Payable<Hourly>`* auf denselben Typ reduziert werden, nämlich *`Payable`*, so daß `Hourly` das Interface doppelt implemen-

tieren würde. Interessanterweise läßt sich der Quelltext übersetzen, wenn Sie, wie der Compiler, die generischen Parameter beider Vorkommen von `Payable` entfernen.

[161] Dieser Aspekt kann lästig werden, wenn Sie einige der eher grundlegenden Interfaces wie `Comparable<T>` verwenden (siehe später in diesem Kapitel).

Übungsaufgabe 31: (1) Entfernen Sie die generischen Typparameter aus dem Beispiel *MultipleInterfaceVariants.java* und ändern Sie den Quelltext so, daß er sich übersetzen läßt. ■

16.11.3 Typumwandlungen und Warnungen

[162] Eine explizite Typumwandlung oder die Verwendung des `instanceof`-Operators mit einem generischen Typparameter ist wirkungslos. Die folgende Containerklasse `FixedSizeStack` speichert Werte intern als `Object` und wandelt sie vor der Rückgabe in `T` um:

```
//: generics/GenericCast.java
class FixedSizeStack<T> {
    private int index = 0;
    private Object[] storage;
    public FixedSizeStack(int size) {
        storage = new Object[size];
    }
    public void push(T item) { storage[index++] = item; }
    @SuppressWarnings("unchecked")
    public T pop() { return (T) storage[--index]; }
}

public class GenericCast {
    public static final int SIZE = 10;
    public static void main(String[] args) {
        FixedSizeStack<String> strings = new FixedSizeStack<String>(SIZE);
        for(String s : "A B C D E F G H I J".split(" "))
            strings.push(s);
        for(int i = 0; i < SIZE; i++) {
            String s = strings.pop();
            System.out.print(s + " ");
        }
    }
} /* Output:
      J I H G F E D C B A
   *///:~
```

Ohne die `@SuppressWarnings`-Annotation gibt der Compiler eine „unchecked“-Warnung für die `pop()`-Methode aus. Aufgrund der Typauslöschung kann der Compiler nicht wissen, ob die Umwandlung typsicher ist, so daß die `pop()`-Methode eigentlich keine Typumwandlung durchführt. Der Typparameter wird bis zu seiner ersten Schranke ausgelöscht, im obigen Fall also `Object`, das heißt `pop()` „wandelt“ `Object` in `Object` um.

[163] Es gibt Situationen im Umgang mit generischen Typen, in denen die Notwendigkeit der Typumwandlung unumgänglich ist, der Compiler dabei aber unangebrachterweise eine Warnung ausgibt, zum Beispiel:

```
//: generics/NeedCasting.java
import java.io.*;
import java.util.*;

public class NeedCasting {
```

```
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream(args[0]));
        List<Widget> shapes = (List<Widget>)in.readObject();
    }
} ///:~
```

Sie lernen in Kapitel 19, daß die `readObject()`-Methode nicht wissen kann, was sie einliest so daß ihr Rückgabetyp umgewandelt werden muß. Wenn Sie nun die Annotation `@SuppressWarnings` auskommentieren erhalten Sie die folgende Warnung:

```
Note: NeedCasting.java uses unchecked or unsafe operations.
Note: Recompile with -Xlint:unchecked for details.
```

Wenn Sie dem Hinweis folgen und den Compiler mit dem Schalter `-Xlint:unchecked` nochmals aufrufen:

```
NeedCasting.java:10: warning: [unchecked] unchecked cast
found   : java.lang.Object
required: java.util.List<Widget>
        List<Widget> shapes = (List<Widget>)in.readObject();
                                                        ^
1 warning
```

Obwohl Sie gezwungen sind, eine Typumwandlung anzuwenden, ~~and yet you're told you shouldn't~~. Zur Lösung dieses Problems müssen Sie eine neue Form der Typumwandlung gebrauchen, die erst seit der SE 5 vorhanden ist, nämlich die `cast()`-Methode eines generischen Klassenobjektes (siehe Unterabschnitt 15.2.3):

```
//: generics/ClassCasting.java
import java.io.*;
import java.util.*;

public class ClassCasting {
    @SuppressWarnings("unchecked")
    public void f(String[] args) throws Exception {
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream(args[0]));
        // Won't Compile:
        // List<Widget> lw1 =
        // List<Widget>.class.cast(in.readObject());
        List<Widget> lw2 = List.class.cast(in.readObject());
    }
} ///:~
```

Sie können die von `readObject()` zurückgegebene Referenz allerdings nicht in den eigentlichen Typ (*`List<Widget>`*) umwandeln, die folgende Syntax ist also nicht erlaubt:

```
List<Widget>.class.cast(in.readObject())
```

Auch wenn Sie eine zusätzliche Umwandlung verwenden

```
List<Widget>List.class.cast(in.readObject())
```

gibt der Compiler eine Warnung aus.

Übungsaufgabe 32: (1) Verifizieren Sie, daß die Klasse `FixedSizeStack` im Beispiel *GenericCast.java* Ausnahmen hervorruft, wenn Sie versucht, über die Grenzen des Arrays hinauszugehen. Ist die Prüfung auf Einhaltung der Arraygrenzen nicht erforderlich? ■

Übungsaufgabe 33: (3) Reparieren Sie das Beispiel *GenericCast.java* mit Hilfe von `ArrayList`. ■

16.11.4 Überladen generischer Methoden

[164] Das folgende Beispiel wirkt zwar vernünftig, läßt sich aber nicht übersetzen:

```
//: generics/UseList.java
// {CompileTimeError} (Won't compile)
import java.util.*;

public class UseList<W,T> {
    void f(List<T> v) {}
    void f(List<W> v) {}
} ///:~
```

Die beiden überladenen Versionen von `f()` haben nach der Typauslöschung identische Signaturen.

Sie müssen unterscheidbare Methodennamen verwenden, wenn die Eindeutigkeit der Parameterliste nach der Typumwandlung nicht mehr gewährleistet ist:

```
//: generics/UseList2.java
import java.util.*;

public class UseList2<W,T> {
    void f1(List<T> v) {}
    void f2(List<W> v) {}
} ///:~
```

Zum Glück bemerkt der Compiler diese Art von Problemen.

16.11.5 Basisklasse stiehlt Interface

[165] Angenommen, Sie haben eine Klasse für Haustiere, die über das Interface *`Comparable`* gestattet, Objekte dieser Klasse miteinander zu vergleichen:

```
//: generics/ComparablePet.java
public class ComparablePet implements Comparable<ComparablePet> {
    public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Es ist sinnvoll, die Vergleichstypen für eine von `ComparablePet` abgeleitete Klasse zu beschränken. Beispielsweise sollte ein `Cat`-Objekt nur mit anderen `Cat`-Objekt verglichen werden können:

```
//: generics/HijackedInterface.java
// {CompileTimeError} (Won't compile)

class Cat extends ComparablePet implements Comparable<Cat>{
    // Error: Comparable cannot be inherited with
    // different arguments: <Cat> and <Pet>
    public int compareTo(Cat arg) { return 0; }
} ///:~
```

Leider funktioniert diese Lösung nicht. Ist der Typparameter des *`Comparable`*-Interfaces in der Klasse `ComparablePet` einmal mit `ComparablePet` bewertet, kann keine von `ComparablePet` abgeleitete Klasse ihre Objekte mit einem anderen Typ als `ComparablePet` vergleichen:

```
//: generics/RestrictedComparablePets.java
class Hamster extends ComparablePet
    implements Comparable<ComparablePet> {
```

```
    public int compareTo(ComparablePet arg) { return 0; }
}

// Or just:

class Gecko extends ComparablePet {
    public int compareTo(ComparablePet arg) { return 0; }
} ///:~
```

Die Klasse `Hamster` zeigt, daß das Interface der Basisklasse neu implementiert werden kann, wenn der Parametertyp exakt übereinstimmt. Das bedeutet allerdings nicht anderes, als die Methode der Basisklasse zu überschrieben (siehe Klasse `Gecko`).

16.12 Selbstbeschränkte Typen

[166] Im Kontext der generischen Typen von Java tritt von Zeit zu Zeit eine Spracheigentümlichkeit (*idiom*) auf, bei der sich das Gehirn zu krümmen scheint:

```
class SelfBounded<T extends SelfBounded<T>> { // ...
```

Diese Deklaration entfaltet die schwindelerregende Wirkung zweier gegeneinander gerichteter Spiegel, das heißt einer unendlichen Reflexion. Die Klasse `SelfBounded` hat einen beschränkten generischen Typparameter `T`, wobei die Schranke durch `SelfBounded`, wiederum mit generischem Typparameter `T`, gegeben ist.

[167] Die Syntax ist auf den ersten Blick schwierig zu analysieren und unterstreicht, daß das Schlüsselwort `extends` bei der Beschränkung von Parametertypen eine völlig andere Bedeutung hat, als beim Ableiten von Unterklassen.

16.12.1 *Curiously Recurring Generics*

[168] Um zu verstehen, was es mit einem selbstbeschränkten Typ (*self-bounded type*) auf sich hat, beginnen mit einer einfacheren Version der Schreibweise, ohne Selbstbeschränkung. Sie können keinen Typ direkt aus einem generischen Typparameter ableiten. Sie *können* allerdings eine Unterklasse aus einer Klasse mit einem generischen Typparameter ableiten, zum Beispiel:

```
//: generics/CuriouslyRecurringGeneric.java
class GenericType<T> {}

public class CuriouslyRecurringGeneric
    extends GenericType<CuriouslyRecurringGeneric> {} ///:~
```

Man könnte diese Konstruktion in Anlehnung an Jim Copliens Entwurfsmuster *Curiously-Recurring-Template* für C++ vielleicht „Curiously Recurring Generics (CRG)“ nennen, übersetzt etwa „in ausgefallener Weise wiederkehrender generischer Typ“. Der Teil „in ausgefallener Weise wiederkehrend“ bezieht sich darauf, daß die abgeleitete Klasse in ihrer Basisklasse vorkommt.

[169] Sprechen Sie laut nach, um zu verstehen, was dies bedeutet: „Ich lege eine neue Klasse an, die von einem generischen Typ abgeleitet ist, der meine neue Klasse als Parameter erwartet.“ Was kann der generische Basistyp mit mit dem Namen der abgeleiteten Klasse anfangen? Nun, die generischen Typen von Java drehen sich um Argument- und Rückgabetypen. Die Basisklasse kann also den abgeleiteten Typ als Argument- beziehungsweise Rückgabetyp ihrer Methoden einsetzen. Die abgeleitete Klasse kann außerdem als Typ von Feldern verwendet werden, auch wenn der Typ durch die Typauslöschung zu `Object` wird. Die folgende generische Klasse `GenericHolder` implementiert diese Funktionalität:

```
//: generics/BasicHolder.java
public class BasicHolder<T> {
    T element;
    void set(T arg) { element = arg; }
    T get() { return element; }
    void f() {
        System.out.println(element.getClass().getSimpleName());
    }
} ///:~
```

Es handelt sich um eine gewöhnliche generische Klasse mit Methoden, die Objekte des Parametertyps als Argument erwarten beziehungsweise zurückgeben sowie einer Methode, die auf dem Feld operiert (allerdings können auf diesem Feld nur `Object`-Methoden aufgerufen werden).

[170] We can use BasicHolder in a curiously recurring generic:

```
//: generics/CRGWithBasicHolder.java
class Subtype extends BasicHolder<Subtype> {}

public class CRGWithBasicHolder {
    public static void main(String[] args) {
        Subtype st1 = new Subtype(), st2 = new Subtype();
        st1.set(st2);
        Subtype st3 = st1.get();
        st1.f();
    }
} /* Output:
      Subtype
   *////:~
```

Beachten Sie die folgende wichtige Eigenschaft der abgeleiteten Klasse `Subtype`: Die neue Klasse erwartet Argumente und liefert Rückgabewert vom Typ `Subtype`, nicht `BasicHolder`. Dies ist der Extrakt des „Entwurfsmusters“ CRG: Die Basisklasse ersetzt ihren Typparameter durch den Namen der abgeleiteten Klasse. Die generische Basisklasse wird somit hinsichtlich ihrer abgeleiteten Klassen zu einer Art Vorlage für allgemeine Funktionalität, welche aber den Typ der abgeleiteten Klasse als Argumenttyp für `set()` und Rückgabetyp für `get()` verwendet. In der resultierenden Klasse wird also der exakte Typ anstelle des Basistyps verwendet. In der Klasse `Subtype` ist sowohl das Argument von `set()` als auch der Rückgabewert von `get()` vom Typ `Subtype`.

16.12.2 Selbstbeschränkung

[171] Die Klasse `BasicHolder` gestattet einen beliebigen Parametertyp:

```
//: generics/Unconstrained.java
class Other {}
class BasicOther extends BasicHolder<Other> {}

public class Unconstrained {
    public static void main(String[] args) {
        BasicOther b = new BasicOther(), b2 = new BasicOther();
        b.set(new Other());
        Other other = b.get();
        b.f();
    }
} /* Output:
      Other
   *////:~
```

Die Selbstbeschränkung erzwingt dagegen, daß der Parametertyp eines generischen Typs durch den generischen Typ selbst beschränkt ist. Das folgende Beispiel zeigt, wie eine mit Selbstbeschränkung definierte generische Klasse verwendet beziehungsweise nicht verwendet werden kann:

```
//: generics/SelfBounding.java
class SelfBounded<T extends SelfBounded<T>> {
    T element;
    SelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A extends SelfBounded<A> {}
class B extends SelfBounded<A> {} // Also OK

class C extends SelfBounded<C> {
    C setAndGet(C arg) { set(arg); return get(); }
}

class D {}
// Can't do this:
// class E extends SelfBounded<D> {}
// Compile error: Type parameter D is not within its bound

// Alas, you can do this, so you can't force the idiom:
class F extends SelfBounded {}

public class SelfBounding {
    public static void main(String[] args) {
        A a = new A();
        a.set(new A());
        a = a.set(new A()).get();
        a = a.get();
        C c = new C();
        c = c.setAndGet(new C());
    }
} ///:~
```

Die Selbstbeschränkung erzwingt die Definition einer abgeleiteten Klasse bezüglich einer Ableitungsbeziehung:

```
class A extends SelfBounded<A> {}
```

Sie sind gezwungen, den Namen der Klasse, die Sie definieren wollen, als Parameter an die Basisklasse zu übergeben.

[172] Welcher Nutzen ergibt sich aus der Selbstbeschränkung des Parametertyps. Der Parametertyp muß mit der definierten Klasse übereinstimmen. Die Definition von Klasse `B` zeigt, derive from a SelfBounded that uses a parameter of another SelfBounded, obwohl der bei Klasse `A` gezeigte Fall vorherrscht. Der Versuch der Definition von Klasse `E` zeigt, daß Sie keinen Parametertyp wählen können, der nicht von `SelfBounded` abgeleitet ist.

[173] Bedauerlicherweise läßt sich `F` ohne Warnungen übersetzen, das heißt die vollständige Notation der Selbstbeschränkung läßt sich nicht erzwingen. Wenn die vollständige Schreibweise wirklich notwendig ist, brauchen Sie eventuell ein externes Werkzeug, um sicherzustellen, daß keine raw types anstelle parametrisierter Typen verwendet werden.

[174] Beachten Sie, daß sich alle Klassen auch dann übersetzen lassen, wenn die Selbstbeschränkung entfernt wird. Allerdings wird dann auch `E` übersetzt:

```
//: generics/NotSelfBounded.java
public class NotSelfBounded<T> {
    T element;
    NotSelfBounded<T> set(T arg) {
        element = arg;
        return this;
    }
    T get() { return element; }
}

class A2 extends NotSelfBounded<A2> {}
class B2 extends NotSelfBounded<A2> {}

class C2 extends NotSelfBounded<C2> {
    C2 setAndGet(C2 arg) { set(arg); return get(); }
}

class D2 {}
// Now this is OK:
class E2 extends NotSelfBounded<D2> {} ///:~
```

Die Selbstbeschränkung dient somit offenbar nur dem Erzwingen der Ableitungsbeziehung. Wenn Sie einer generischen Klasse Selbstbeschränkung auferlegen, wissen Sie, daß der von der Klasse verwendete Parametertyp demselben Basistyp angehört, als die Klasse selbst, die den Parametertyp verwendet. Die Selbstbeschränkung zwingt jeden Anwender dieser Klasse, sich dieser Form anzuschließen.

[175] Die Selbstbeschränkung ist auch bei generischen Methoden anwendbar:

```
//: generics/SelfBoundingMethods.java
public class SelfBoundingMethods {
    static <T extends SelfBounded<T>> T f(T arg) {
        return arg.set(arg).get();
    }
    public static void main(String[] args) {
        A a = f(new A());
    }
} ///:~
```

Die Methode wird dadurch vor der Übergabe eines Argumentes geschützt, daß nicht in der beschriebenen Weise selbstbeschränkt ist.

16.12.3 Kovariante Argumente

[176] Der Wert selbstbeschränkter generischer Typen besteht darin, daß sie *kovariante Argumenttypen* ermöglichen, das heißt die Typen der Argumente von Methoden können kovariant („mit der Ableitungsrichtung“) variieren.

[177] Selbstbeschränkte generische Typen liefern auch Rückgabetypen, die mit dem Typ der abgeleiteten Klasse übereinstimmen, aber das ist weniger bedeutsam, da seit der SE 5 *kovariante Rückgabetypen* erlaubt sind:

```
//: generics/CovariantReturnTypes.java
class Base {}
class Derived extends Base {}
```

```
interface OrdinaryGetter {
    Base get();
}

interface DerivedGetter extends OrdinaryGetter {
    // Return type of overridden method is allowed to vary:
    Derived get();
}

public class CovariantReturnTypes {
    void test(DerivedGetter d) {
        Derived d2 = d.get();
    }
} ///:~
```

Die `get()`-Methode der Klasse *`DerivedGetter`* überschreibt die `get()`-Methode von *`OrdinaryGetter`* *und* gibt einen Typ zurück, der vom Rückgabetyp von *`OrdinaryGetter`*`.get()` abgeleitet ist. Obwohl vollkommen einleuchtet, daß eine abgeleitete Methode in der Lage sein sollte, auch einen abgeleiteten Typ zurückzugeben, war dieses Verhalten bei den früheren Java-Versionen nicht zulässig.

[178] Ein selbstbeschränkter generischer Typ liefert tatsächlich den exakten abgeleiteten Typ als Rückgabewert, wie die folgende `get()`-Methode zeigt:

```
//: generics/GenericsAndReturnTypes.java
interface GenericGetter<T extends GenericGetter<T>> {
    T get();
}

interface Getter extends GenericGetter<Getter> {}

public class GenericsAndReturnTypes {
    void test(Getter g) {
        Getter result = g.get();
        GenericGetter gg = g.get(); // Also the base type
    }
} ///:~
```

Beachten Sie, daß sich dieses Beispiel nicht würde haben übersetzen lassen, bis mit der SE 5 die kovarianten Rückgabetypen in die Sprache aufgenommen worden waren.

[179] In einem nicht-generischen Quelltext dürfen die Argumenttypen dagegen nicht mit der Ableitungsrichtung variieren:

```
//: generics/OrdinaryArguments.java
class OrdinarySetter {
    void set(Base base) {
        System.out.println("OrdinarySetter.set(Base)");
    }
}

class DerivedSetter extends OrdinarySetter {
    void set(Derived derived) {
        System.out.println("DerivedSetter.set(Derived)");
    }
}

public class OrdinaryArguments {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
```

```
        DerivedSetter ds = new DerivedSetter();
        ds.set(derived);
        ds.set(base); // Compiles: overloaded, not overridden!
    }
} /* Output:
      DerivedSetter.set(Derived)
      OrdinarySetter.set(Base)
   *///:~
```

Beide Methoden, `set(derived)` und `set(base)` sind gültig, aber *`DerivedGetter`*`.set()` überschreibt *`OrdinaryGetter`*`.set()` nicht, sondern überlädt sie. Die Ausgabe zeigt, daß die Klasse *`DerivedGetter`* zwei Methoden enthält, die Basisklassenversion also noch immer vorhanden und die `set()`-Methode somit überladen ist.

[180] Bei selbstbeschränkten generischen Typen enthält die abgeleitete Klasse nur eine Methode und diese akzeptiert den abgeleiteten Typ als Argument, nicht den Basistyp:

```
//: generics/SelfBoundingAndCovariantArguments.java
interface SelfBoundSetter<T extends SelfBoundSetter<T>> {
    void set(T arg);
}

interface Setter extends SelfBoundSetter<Setter> {}

public class SelfBoundingAndCovariantArguments {
    void testA(Setter s1, Setter s2, SelfBoundSetter sbs) {
        s1.set(s2);
        // s1.set(sbs); // Error:
        // set(Setter) in SelfBoundSetter<Setter>
        // cannot be applied to (SelfBoundSetter)
    }
} ///:~
```

Der Compiler erkennt den Versuch nicht, der `set()`-Methode ein Argument vom Basistyp zu übergeben, da es keine Methode mit entsprechender Signatur gibt. Die Methode wurde effektiv überschrieben.

[181] Ohne Selbstbeschränkung schaltet sich der gewöhnliche Ableitungsmechanismus dazwischen und Sie erhalten Überladung, wie im nicht-generischen Fall:

```
//: generics/PlainGenericInheritance.java
class GenericSetter<T> { // Not self-bounded
    void set(T arg){
        System.out.println("GenericSetter.set(Base)");
    }
}

class DerivedGS extends GenericSetter<Base> {
    void set(Derived derived){
        System.out.println("DerivedGS.set(Derived)");
    }
}

public class PlainGenericInheritance {
    public static void main(String[] args) {
        Base base = new Base();
        Derived derived = new Derived();
        DerivedGS dgs = new DerivedGS();
        dgs.set(derived);
        dgs.set(base); // Compiles: overloaded, not overridden!
```

```
    }
} /* Output:
      DerivedGS.set(Derived)
      GenericSetter.set(Base)
   *///:~
```

Das Beispiel imitiert *OrdinaryArguments.java*. Dort ist `DerivedGetter` von der Klasse `OrdinaryGetter` abgeleitet, die eine `set(Base)`-Methode enthält. Hier ist `DerivedGS` von der Klasse `GenericSetter<Base>` abgeleitet, die ebenfalls eine `set(Base)`-Methode enthält (erzeugt anhand des gewählten Parametertyps). Wie bei *OrdinaryArguments.java* hat auch `DerivedGS` zwei überladene Versionen von `set()`. Ohne Selbstbeschränkung bewirken ableitungsverwandte Argumenttypen Überladung. Mit Selbstbeschränkung haben Sie *eine* Version der Methode, die den exakten Argumentyp erwartet.

Übungsaufgabe 34: (4) Schreiben Sie eine selbstbeschränkte generische Klasse die eine abstrakte Methode enthält, deren Argument und Rückgabewert den Typ des generischen Typparameters haben. Rufen Sie die abstrakte Methode aus einer konkreten Methode der Klasse auf und geben Sie den Rückgabewert zurück. Leiten Sie eine Klasse von ihrer selbstbeschränkten generischen Klasse ab und testen Sie sie. ■

16.13 Typsicherheit zur Laufzeit bei Containern vor SE5

[182] Ein generischer Container kann in einem Teil einer Anwendung verarbeitet werden, der vor der SE 5 geschrieben wurde, wodurch die Gefahr besteht, daß ein typfremdes Element in Ihren generischen Container einsetzt wird. Die Klasse `java.util.Collections` definiert seit der SE 5 die folgenden statischen Methoden, um das Problem der Typprüfung in dieser Situation zu lösen: `checkedCollection()`, `checkedList()`, `checkedMap()`, `checkedSet()`, `checkedSortedMap()` und `checkedSortedSet()`. Jede dieser Methoden erwartet den zur Laufzeit zu überwachenden Container als erstes und den zu erzwingenden Typ als zweites Argument.

[183] Ein geprüfter Container wirft beim Versuch ein typfremdes Element *einzusetzen* eine Ausnahme vom Typ `ClassCastException` aus. Ein nicht-generischer (*raw type*) Container informiert Sie im Gegensatz dazu, wenn Sie das Element *entnehmen*. Im letzteren Fall wissen Sie, daß es ein Problem gibt, nicht aber, wer der Übeltäter ist. Bei einem geprüften Container wissen Sie, wer versucht hat, das typfremde Element einzusetzen.

[184] Wir verwenden das Einsetzen eines `Cat`-Objektes in eine Liste von `Dog`-Objekten als Beispiel. Die Methode `oldStyleMethod()` repräsentiert den älteren Quelltext, indem sie eine *raw type* `List`-Referenz als Argument erwartet. Die Annotation `@SuppressWarnings` ist erforderlich, um die resultierende Warnung zu unterdrücken:

```
//: generics/CheckedList.java
// Using Collection.checkedList().
import typeinfo.pets.*;
import java.util.*;

public class CheckedList {
    @SuppressWarnings("unchecked")
    static void oldStyleMethod(List probablyDogs) {
        probablyDogs.add(new Cat());
    }
    public static void main(String[] args) {
        List<Dog> dogs1 = new ArrayList<Dog>();
        oldStyleMethod(dogs1); // Quietly accepts a Cat
```

```
        List<Dog> dogs2 =
            Collections.checkedList(new ArrayList<Dog>(), Dog.class);
        try {
            oldStyleMethod(dogs2); // Throws an exception
        } catch(Exception e) {
            System.out.println(e);
        }
        // Derived types work fine:
        List<Pet> pets =
            Collections.checkedList(new ArrayList<Pet>(), Pet.class);
        pets.add(new Dog());
        pets.add(new Cat());
    }
} /* Output:
      java.lang.ClassCastException: Attempt to insert class typeinfo.pets.Cat
        element into collection with element type class typeinfo.pets.Dog
   *///:~
```

Wenn Sie das Programm ausführen, sehen Sie, daß sich das `Cat`-Objekt unangefochten in die von `dogs1` referenzierte Liste einsetzen läßt, während `dogs2` beim Einsetzen des typfremden Elementes unmittelbar eine Ausnahme auswirft. Sie sehen außerdem, daß Sie Elemente eines abgeleiteten Typs in einen Container einsetzen können, der die Elemente hinsichtlich ihres Basistyps prüft.

Übungsaufgabe 35: (1) Ändern Sie das Beispiel *CheckedList.java* so, daß es die in diesem Kapitel definierten `Coffee`-Klassen verwendet. ■

16.14 Generische throws-Klausel bei Methoden

[185] Aufgrund der Typauslöschung ist die Generizität von Ausnahme sehr eng begrenzt. Eine `catch`-Klausel kann keine Ausnahme von generischem Typ abfangen, da der exakte Ausnahmetyp sowohl zur Übersetzungs- als auch zur Lauftzeit bekannt sein muß. Eine generische Klasse kann außerdem weder direkt noch indirekt von `Throwable` abgeleitet werden (wodurch Sie daran gehindert werden, generische Ausnahmen zu definieren, die nicht abgefangen werden können).

[186] In der `throws`-Klausel einer Methodendeklaration sind Typparamater erlaubt. Dadurch können Sie ihren Quelltext generisch anlegen, so daß das Verhalten des Programms vom Typ einer geprüften Ausnahme abhängt:

```
//: generics/ThrowGenericException.java
import java.util.*;

interface Processor<T,E extends Exception> {
    void process(List<T> resultCollector) throws E;
}

class ProcessRunner<T,E extends Exception> extends ArrayList<Processor<T,E>> {
    List<T> processAll() throws E {
        List<T> resultCollector = new ArrayList<T>();
        for(Processor<T,E> processor : this)
            processor.process(resultCollector);
        return resultCollector;
    }
}

class Failure1 extends Exception {}

class Processor1 implements Processor<String,Failure1> {
```

```
    static int count = 3;
    public void process(List<String> resultCollector) throws Failure1 {
        if(count-- > 1)
            resultCollector.add("Hep!");
        else
            resultCollector.add("Ho!");
        if(count < 0)
            throw new Failure1();
    }
}

class Failure2 extends Exception {}

class Processor2 implements Processor<Integer,Failure2> {
    static int count = 2;
    public void process(List<Integer> resultCollector) throws Failure2 {
        if(count-- == 0)
            resultCollector.add(47);
        else {
            resultCollector.add(11);
        }
        if(count < 0)
            throw new Failure2();
    }
}

public class ThrowGenericException {
    public static void main(String[] args) {
        ProcessRunner<String,Failure1> runner =
            new ProcessRunner<String,Failure1>();
        for(int i = 0; i < 3; i++)
            runner.add(new Processor1());
        try {
            System.out.println(runner.processAll());
        } catch(Failure1 e) {
            System.out.println(e);
        }

        ProcessRunner<Integer,Failure2> runner2 =
            new ProcessRunner<Integer,Failure2>();
        for(int i = 0; i < 3; i++)
            runner2.add(new Processor2());
        try {
            System.out.println(runner2.processAll());
        } catch(Failure2 e) {
            System.out.println(e);
        }
    }
} ///:~
```

[187] Das Interface *Processor* deklariert eine Methode `process()`, die eine Ausnahme vom Typ `E` auswerfen kann. Das Ergebnis der `process()`-Methode wird in dem von der *List<T>*-Referenzvariablen `resultCollector` referenzierten Objekt gespeichert (einem sogenannten collecting parameter). Die `processAll()`-Methode der Klasse `ProcessRunner` verarbeitet alle vorgehaltenen *Processor*-Objekte und gibt die gesammelten Ergebnisse zurück.

[188] Wenn Sie die ausgeworfenen Ausnahmen nicht parametrisieren könnten, wären Sie aufgrund der geprüften Ausnahmen nicht in der Lage, dieses Programm generisch zu schreiben.

Übungsaufgabe 36: (2) Deklarieren Sie im Interface `Processor` eine zweite parametrisierte Ausnahme und zeigen Sie, daß die Ausnahmen can vary independently. ■

16.15 Mixins

[189] Der Begriff „Mixin“ hat im Laufe der Zeit zahlreiche Bedeutungen angenommen. Das grundlegende Konzept besteht darin, die Fähigkeiten mehrerer Klassen zu einer neuen Klasse zu kombinieren, die sämtliche Typen dieses Verbunds repräsentiert. Diese Kombinationsvorgang ist häufig der letzte Schritt, which makes it convenient Klassen mühelos zusammenzusetzen.

[190] Ein Vorteil von Mixins besteht darin, daß sie Eigenschaften und Verhalten konsistent über mehrere Klassen hinweg verteilen. Wenn Sie eine Klasse des Mixins modifizieren, wird die Änderung in allen Klassen wirksam, die sich auf die geänderte Komponente stützen. Because of this, mixins have part of the flavor of aspect-oriented programming (AOP), and aspects are often suggested to solve the mixin problem.

16.15.1 Mixins in C++

[191] Eines der stärksten Argumente zugunsten der Mehrfachvererbung bei C++ ist die Konstruktion von Mixins. Ein interessanterer und eleganterer Ansatz für Mixins ist aber die Verwendung parametrisierter Typen. Ein Mixin ist hierbei eine Klasse, die von ihrem Typparameter abgeleitet ist. In C++ läßt sich ein Mixin mühelos konstruieren, da sich C++ den Typ des Templateparameters „merkt“.

[192] Das folgende Beispiel ist in C++ geschrieben und hat zwei Mixin-Typen: Ein Typ gestattet Ihnen, jedem Objekt einen Zeitstempel hinzuzufügen, der andere implementiert eine Seriennummer:

```
//: generics/Mixins.cpp
#include <string>
#include <ctime>
#include <iostream>
using namespace std;

template<class T> class TimeStamped : public T {
  long timeStamp;
public:
  TimeStamped() { timeStamp = time(0); }
  long getStamp() { return timeStamp; }
};

template<class T> class SerialNumbered : public T {
  long serialNumber;
  static long counter;
public:
  SerialNumbered() { serialNumber = counter++; }
  long getSerialNumber() { return serialNumber; }
};

// Define and initialize the static storage:
template<class T> long SerialNumbered<T>::counter = 1;

class Basic {
  string value;
public:
  void set(string val) { value = val; }
```

```
  string get() { return value; }
};

int main() {
  TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
  mixin1.set(''test string 1'');
  mixin2.set(''test string 2'');
  cout << mixin1.get() << '' '' << mixin1.getStamp() <<
    '' '' << mixin1.getSerialNumber() << endl;
  cout << mixin2.get() << '' '' << mixin2.getStamp() <<
    '' '' << mixin2.getSerialNumber() << endl;
} /* Output: (Sample)
test string 1 1129840250 1
test string 2 1129840250 2
*///:~
```

Der resultierende Typ von `mixin1` und `mixin2` hat alle Methoden der Mixin-Komponenten `TimeStamped` und `SerialNumbered`. Sie können sich ein Mixin wie eine Funktion vorstellen, die vorhandene Klassen auf neue Unterklassen abbildet. Beachten Sie, wie einfach es ist, mit diesem Ansatz ein Mixin zu erzeugen. Sie sagen einfach „Das will ich“ und es geschieht:

```
TimeStamped<SerialNumbered<Basic>> mixin1, mixin2;
```

Leider gestatten die generischen Typen von Java diesen Ansatz nicht. Aufgrund der Typauslöschung geht der Typ der Basisklasse verloren, so daß eine generische Klasse nicht direkt von einem generischen Parameter abgeleitet werden kann.

16.15.2 Mixins mit Interfaces

[193] Häufig wird vorgeschlagen, den Mixin-Effekt mit Hilfe von Interfaces herbeizuführen, zum Beispiel:

```
//: generics/Mixins.java
import java.util.*;

interface TimeStamped { long getStamp(); }

class TimeStampedImp implements TimeStamped {
    private final long timeStamp;
    public TimeStampedImp() {
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}

interface SerialNumbered { long getSerialNumber(); }

class SerialNumberedImp implements SerialNumbered {
    private static long counter = 1;
    private final long serialNumber = counter++;
    public long getSerialNumber() { return serialNumber; }
}

interface Basic {
    public void set(String val);
    public String get();
}

class BasicImp implements Basic {
    private String value;
```

```
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Mixin extends BasicImp
    implements TimeStamped, SerialNumbered {
    private TimeStamped timeStamp = new TimeStampedImp();
    private SerialNumbered serialNumber = new SerialNumberedImp();
    public long getStamp() { return timeStamp.getStamp(); }
    public long getSerialNumber() {
        return serialNumber.getSerialNumber();
    }
}

public class Mixins {
    public static void main(String[] args) {
        Mixin mixin1 = new Mixin(), mixin2 = new Mixin();
        mixin1.set("test string 1");
        mixin2.set("test string 2");
        System.out.println(mixin1.get() + " " +
                           mixin1.getStamp() + " " + mixin1.getSerialNumber());
        System.out.println(mixin2.get() + " " +
                           mixin2.getStamp() + " " + mixin2.getSerialNumber());
    }
} /* Output: (Sample)
      test string 1 1132437151359 1
      test string 2 1132437151359 2
   *///:~
```

Die Klasse `Mixin` beruht im wesentlichen auf Delegation. Jede Mixin-Komponente bekommt ein Feld in `Mixin`, um Methodenaufrufe an das entsprechende Objekt weiterzuleiten. Die Klassen in diesem Beispiel sind trivial, aber der Quelltext wächst bei komplexeren Mixins rasch an.[3]

Übungsaufgabe 37: (2) Legen Sie im Beispiel *Mixins.java* eine neue Mixin-Komponente `Colored` an, verknüpfen Sie sie mit der Klasse `Mixin` und zeigen Sie, daß das Programm funktioniert. ■

16.15.3 Das Decorator-Entwurfsmuster

[194] Das Mixin-Konzept scheint eng mit dem *Decorator*-Entwurfsmuster verwandt zu sein, wenn Sie sich die Art und Weise seiner Verwendung ansehen.[4] Dekoratoren werden häufig eingesetzt, wenn das schlichte Ableiten neuer Klassen mit dem Ziel, alle möglichen Kombinationen zu erfassen, derart Klassen mit sich bringt, das die Vorgehensweise nicht mehr praktikabel ist.

[195] Das *Decorator*-Entwurfsmuster arbeitet mit mehrschichtigen Objekten und weist individuellen Objekten dynamisch und transparent zusätzliche Funktionalität zu. Das Entwurfsmuster schreibt vor, daß alle um Ihr anfängliches Objekt gelegten Objekte diesselbe grundlegende Schnittstelle haben. Ein Klasse ist dekorierbar und Sie erweitern die Funktionalität, indem Sie weitere Klassen um die dekorierbare Klasse herumlegen. Die Verwendung der Dekoratoren wird dadurch transparent, daß es eine Menge häufiger Nachrichten gibt, die Sie dem Objekt senden können, gleichgültig ob es dekoriert wurde oder nicht. Eine dekorierende Klasse kann auch Methoden hinzufügen, allerdings nur in begrenztem Umfang, wie Sie sehen werden.

[3] Manche Entwicklungsumgebungen wie Eclipse oder IntelliJ Idea erzeugen den zur Delegation benötigten Quelltext automatisch.

[4] Entwurfsmuster sind Gegenstand des Buches *Thinking in Patterns (with Java)* unter der Webadresse *http://www-.mindview.net*. Siehe auch Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[196] Dekoratoren werden mittels Komposition und formaler Strukturen (the decoratable decorator hierarchy) implementiert, während Mixins auf Ableitung aufbauen. Sie können sich ein auf einem parametrisierten Typ basierendes Mixin also als generischen Dekorationsmechanismus vorstellen, der die Ableitungsstruktur des *Decorator*-Entwurfsmusters nicht benötigt.

[197] Das vorige Beispiel läßt sich in Richtung des *Decorator*-Entwurfsmusters umformen:

```
//: generics/decorator/Decoration.java
package generics.decorator;
import java.util.*;

class Basic {
    private String value;
    public void set(String val) { value = val; }
    public String get() { return value; }
}

class Decorator extends Basic {
    protected Basic basic;
    public Decorator(Basic basic) { this.basic = basic; }
    public void set(String val) { basic.set(val); }
    public String get() { return basic.get(); }
}

class TimeStamped extends Decorator {
    private final long timeStamp;
    public TimeStamped(Basic basic) {
        super(basic);
        timeStamp = new Date().getTime();
    }
    public long getStamp() { return timeStamp; }
}

class SerialNumbered extends Decorator {
    private static long counter = 1;
    private final long serialNumber = counter++;
    public SerialNumbered(Basic basic) { super(basic); }
    public long getSerialNumber() { return serialNumber; }
}

public class Decoration {
    public static void main(String[] args) {
        TimeStamped t = new TimeStamped(new Basic());
        TimeStamped t2 = new TimeStamped(new SerialNumbered(new Basic()));
        //! t2.getSerialNumber(); // Not available
        SerialNumbered s = new SerialNumbered(new Basic());
        SerialNumbered s2 = new SerialNumbered(new TimeStamped(new Basic()));
        //! s2.getStamp(); // Not available
    }
} ///:~
```

Die bei einem Mixin entstehende Klasse enthält alle interessanten Methoden, während ein Objekt nach Anwendung eines oder mehrerer Dekoratoren den Typ des letzten Dekorators annimmt. Obwohl es *möglich* ist, mehr als eine Schicht anzulegen, ist der resultierende Typ durch die letzte Schicht definiert, das heißt nur die Methoden der letzten Schicht sind sichtbar, während ein Mixin *alle* Typen des Verbunds repräsentiert. Ein deutlicher Nachteil des *Decorator*-Entwurfsmuster besteht darin, daß effektiv nur eine Dekorationsschicht funktionstüchtig ist, nämlich die letzte, wohingegen der Mixin-Ansatz wohl natürlicher ist. Das *Decorator*-Entwurfsmuster ist also nur eine eingeschränkte Lösung für die Aufgabe, ein Mixin zu konstruieren.

Übungsaufgabe 38: (4) Entwickeln Sie ein einfaches *Decorator*-System, beginnend mit schwarzem Kaffee. Schreiben Sie Dekoration für heiße Milch, Milchschaum, Schokoladen- und Caramelsirup sowie für Schlagsahne. ■

16.15.4 Mixins mit dynamischen Stellvertretern

[198] Mit Hilfe eines dynamischen Stellvertreters läßt sich ein Mechanismus konstruieren, der Mixins besser modelliert, als das *Decorator*-Entwurfsmuster (die Funktionsweise von dynamischen Stellvertretern bei Java wird in Abschnitt 15.7 beschrieben). Mit einem dynamischen Stellvertreter ist der *dynamische* Typ der entstehenden Klasse aus allen im Verbund befindlichen Komponenten kombiniert.

[199] Aufgrund der Randbedingungen für dynamische Stellvertreter muß jede Klasse, die in den Verbund integiert wird, das Interface `InvocationHandler` implementieren:

```
//: generics/DynamicProxyMixin.java
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Tuple.*;

class MixinProxy implements InvocationHandler {
    Map<String,Object> delegatesByMethod;
    public MixinProxy(TwoTuple<Object,Class<?>>... pairs) {
        delegatesByMethod = new HashMap<String,Object>();
        for(TwoTuple<Object,Class<?>> pair : pairs) {
            for(Method method : pair.second.getMethods()) {
                String methodName = method.getName();
                // The first interface in the map
                // implements the method.
                if (!delegatesByMethod.containsKey(methodName))
                    delegatesByMethod.put(methodName, pair.first);
            }
        }
    }
    public Object invoke(Object proxy, Method method,
                         Object[] args) throws Throwable {
        String methodName = method.getName();
        Object delegate = delegatesByMethod.get(methodName);
        return method.invoke(delegate, args);
    }
    @SuppressWarnings("unchecked")
    public static Object newInstance(TwoTuple... pairs) {
        Class[] interfaces = new Class[pairs.length];
        for(int i = 0; i < pairs.length; i++) {
            interfaces[i] = (Class) pairs[i].second;
        }
        ClassLoader cl = pairs[0].first.getClass().getClassLoader();
        return Proxy.newProxyInstance(cl, interfaces, new MixinProxy(pairs));
    }
}

public class DynamicProxyMixin {
    public static void main(String[] args) {
        Object mixin = MixinProxy.newInstance(
            tuple(new BasicImp(), Basic.class),
            tuple(new TimeStampedImp(), TimeStamped.class),
```

```
            tuple(new SerialNumberedImp(), SerialNumbered.class));
        Basic b = (Basic) mixin;
        TimeStamped t = (TimeStamped) mixin;
        SerialNumbered s = (SerialNumbered) mixin;
        b.set("Hello");
        System.out.println(b.get());
        System.out.println(t.getStamp());
        System.out.println(s.getSerialNumber());
    }
} /* Output: (Sample)
      Hello
      1132519137015
      1
   *///:~
```

Da nur der dynamische Typ, nicht aber der statische, alle Typen des Mixins repräsentiert, ist diese Lösung noch nicht so schön wie der Ansatz in C++. Sie müssen den /// zum entsprechenden Typ hin umwandeln, bevor Sie eine seiner Methoden aufrufen können, aber der Ansatz ist deutlich näher an einem echten Mixin.

[200] Es wurde viel Arbeit in die Mixin-Unterstützung von Java investiert, darunter in die Entwicklung wenigstens einer Spracherweiterung namens Jam, speziell um Mixins zu unterstützen.

Übungsaufgabe 39: (1) Legen Sie im Beispiel *DynamicProxyMixin.java* eine neue Mixin-Komponente `Colored` an, verknüpfen Sie sie mit `mixin` und zeigen Sie, daß das Programm funktioniert. ■

16.16 Verborgene Typisierung

[201] Am Anfang dieses Kapitels wurde die Idee vorgestellt, einen Quelltext so zu schreiben, daß er so allgemein wie möglich einsetzbar ist. Zu diesem Zweck müssen wir die Auflagen (*constraints*) hinsichtlich der in unserem Quelltext vorkommenden Typen lockern. Wir versetzen uns damit in die Lage, Quelltext zu schreiben, der unverändert in mehreren Situationen angewendet werden kann, also „generischer“ ist.

[202] Die generischen Typen von Java scheinen einen Schritt in diese Richtung zu gehen. Ein zur schlichten Speicherung von Objekten geschriebener oder verwendeter generischer Typ kann auf jeden Parametertyp angewendet werden (außer primitiven Typen, wobei Autoboxing diese Lücke schließt). Ein generischer „Behälter“ kümmert sich nicht um den Typ seiner Elemente. Ein Quelltext, der sich nicht darum kümmert, mit welchem Elementtyp er arbeitet, läßt sich in der Tat überall anwenden, ist also ziemlich „generisch“.

[203] Sie haben gelernt, daß es zu Problemen kommen kann, wenn Sie Methoden eines Objektes (über die Methoden der Klasse `Object` hinaus) von generischem Typ aufrufen wollen, da Sie, bedingt durch die Typauslöschung, die Beschränkung des Parametertyps angeben müssen, um typsicher bestimmte Methoden des generischen Objektes aufrufen zu können. Die Beschränkung des Parametertyps auf die Ableitung von einer bestimmten Klasse oder das Implementieren eines bestimmten Interfaces ist eine deutliche Eingrenzung des generischen Konzeptes. Im einen oder anderen Fall kommen Sie letztendlich bei einer gewöhnlichen Klasse oder einem gewöhnlichen Interface an, da sich der beschränkte Parametertyp nicht mehr von einem gewöhnlichen Typ unterscheidet.

[204] Einige Programmiersprachen offerieren eine Lösung namens „verborgene Typisierung“ (*latent typing*) oder gleichbedeutend ~~structural typing~~. Eine nette weitere Bezeichnung ist „Duck-Typing“ und bezieht sich auf die Charakterisierung: „If it walks like a duck and talks like a duck, you might as well treat it like a duck.“ Übersetzt etwa: „Wenn sich etwas wie eine Ente bewegt und wie eine

Ente schnattert, können Sie es wie eine Ente behandeln.“ Der Begriff „Duck-Typing“ ist sehr beliebt, vielleicht deshalb, weil er nicht, wie andere Begriffe, mit historischen Altlasten beladen ist.

[205] Ein generischer Quelltext ruft typischerweise nur wenige Methoden eines Objektes von generischem Typ auf. Eine Sprache mit verborgener Typisierung lockert die Anforderungen dahingehend (und gestattet somit generischen Quelltext), daß sie im Gegensatz zu einer bestimmten Klasse oder einem bestimmten Interface, lediglich die Implementierung eines Teils der Methoden verlangt. Dadurch ermöglicht die verborgene Typisierung, die Grenzen von Klassenhierarchien zu überschreiten und Methoden aufrufen, die nicht zur normalen (*common*) Schnittstelle gehören. Eine Anweisung nimmt sozusagen den Standpunkt ein: „Ich kümmere mich nicht darum welchem Typ Du angehörst, solange Du sprechen (`speak()`) und sitzen (`sit()`) kannst.“ Ihr Quelltext wird generischer, da kein bestimmter Typ verlangt wird.

[206] Die verborgene Typisierung ist ein Mechanismus zur Quelltextorganisation und -wiederverwendung. Er gestattet Ihnen, ein Stück Quelltext zu schreiben, das Sie leichter wiederverwenden können, als ohne verborgene Typisierung. Organisation und Wiederverwendung sind die Schalthebel der Softwareentwicklung: Einmal schreiben, mehrmals benutzen und an einer einzigen Stelle deponieren. Da Sie kein Interface exakt benennen müssen, in dem Ihre Methode deklariert ist, schreiben Sie mit verborgener Typisierung weniger Anweisungen und können sie leichter an mehr Stellen gebrauchen.

[207] Zwei Beispiele für Programmiersprachen, die verborgene Typisierung unterstützen, sind Python (frei herunterladbar unter der Webadresse *www.python.org*) und C++.[5] Python ist eine dynamisch typisierte Sprache (nahezu alle Typprüfungen geschehen zur Laufzeit), während C++ eine statisch typisierte Sprache ist (die Typprüfungen finden zur Übersetzungszeit statt), das heißt verborgene Typisierung setzt weder statische noch dynamische Typprüfung voraus.

[208] Die obige Beschreibung läßt sich in Python etwa so ausdrücken:

```
#!/usr/bin/python

class Dog:
    def speak(self):
        print ''Arf!''
    def sit(self):
        print ''Sitting''
    def reproduce(self):
        pass

class Robot:
    def speak(self):
        print ''Click!''
    def sit(self):
        print ''Clank!''
    def oilChange(self):
        pass
    def perform(anything):
        anything.speak()
        anything.sit()

a = Dog()
b = Robot()
perform(a)
perform(b)
#
```

Python verwendet Einrückungen, um Geltungsbereiche zu bestimmen (geschweiften Klammern sind

[5] Auch die Sprachen Ruby und Smalltalk unterstützen verborgene Typisierung.

daher nicht notwendig) und einen Doppelpunkt, um einen neuen Geltungsbereich zu beginnen. Das Zeichen `#` zeigt einen Kommentar an, der sich bis zum Zeilenende erstreckt, wie `//` bei Java. Die Methoden einer Klasse geben explizit das Äquivalent der Selbstreferenz `this` als erstes Argument an (konventionsgemäß `self`). Beim Aufruf eines Konstruktors ist kein `new`-Schlüsselwort erforderlich. Wie `perform()` zeigt, erlaubt Python reguläre (nicht an eine Klasse gebundene) Funktionen.

[209] Beachten Sie bei `perform(anything)`, daß für `anything` kein Typ angegeben und `anything` lediglich ein Bezeichner ist. Das von `anything` referenzierte Objekt muß in der Lage sein, die von der `perform()`-Funktion ausgeführten Operationen durchzuführen, wodurch eine Schnittstelle gegeben ist. Sie brauchen diese Schnittstelle aber niemals anzugeben: sie ist *verborgen*. Die `perform()`-Funktion achtet nicht darauf, welchen Typ ihr Argument hat, das heißt Sie können jedes Objekt übergeben, welches über die Methoden `speak()` und `sit()` verfügt. Wenn Sie `perform()` ein Objekt übergeben, das diese Methoden nicht hat, wird zur Laufzeit eine Ausnahme ausgeworfen.

[210] Das folgende Beispiel implementiert denselben Effekt in C++:

```
//: generics/DogsAndRobots.cpp
class Dog {
public:
  void speak() {}
  void sit() {}
  void reproduce() {}
};

class Robot {
public:
  void speak() {}
  void sit() {}
  void oilChange() {
};

template<class T> void perform(T anything) {
  anything.speak();
  anything.sit();
}

int main() {
  Dog d;
  Robot r;
  perform(d);
  perform(r);
} ///:~
```

In beiden Beispielen, Python und C++, haben `Dog` und `Robot` keine Gemeinsamkeiten, außer zwei Methoden mit übereinstimmenden Signaturen, sind also hinsichtlich ihrer Typen völlig verschieden voneinander. Die `perform()`-Methode kümmert sich aber nicht um den Typ ihres Argumentes und die verborgene Typisierung ermöglicht der Methode, beide Typen von Objekten zu akzeptieren.

[211] C++ garantiert, daß die Methoden tatsächlich aufgerufen werden können. Der Compiler gibt eine Fehlermeldung aus, wenn Sie einen falschen Typ übergeben (diese Fehlermeldungen waren schon immer fürchterlich und geschwätzig; sie sind der Hauptgrund für den schlechten Ruf der Templates von C++). Obwohl zu unterschiedlichen Zeitpunkten, C++ zur Übersetzungs- und Python zur Laufzeit, gewährleisten beide Sprachen, daß Typen nicht mißbraucht werden können und gelten daher als stark typisierte[6] (*strongly typed*) Programmiersprachen. Die verborgene Typisierung

[6] Sie können das ~~type system~~ mit Typumwandlung effektiv außer Kraft setzen. Daher argumentieren einige Kollegen, C++ sei schwach typisiert, aber dies ist eine extreme Sichtweise. Es ist wahrscheinlich sicherer zu sagen, daß C++ „stark typisiert ist, wobei es aber eine Falltür gibt.“

beeinträchtigt die starke Typisierung nicht.

[212] Da die generischen Typen spät zu Java hinzugefügt wurden, bestand keine Gelegenheit mehr, verborgene Typisierung zu implementieren, so daß Java diese Fähigkeit nicht unterstützt. Die generischen Typen von Java scheinen somit zunächst „weniger generisch“ zu sein, als eine Sprache mit verborgener Typisierung.[7] Die Übertragung unseres Beispiels zwingt uns, eine Klasse oder ein Interface zu verwenden und in einer Typparameterbeschränkung anzugeben:

```
//: generics/Performs.java
public interface Performs {
    void speak();
    void sit();
} ///:~

//: generics/DogsAndRobots.java
// No latent typing in Java
import typeinfo.pets.*;
import static net.mindview.util.Print.*;

class PerformingDog extends Dog implements Performs {
    public void speak() { print("Woof!"); }
    public void sit() { print("Sitting"); }
    public void reproduce() {}
}

class Robot implements Performs {
    public void speak() { print("Click!"); }
    public void sit() { print("Clank!"); }
    public void oilChange() {}
}

class Communicate {
    public static <T extends Performs> void perform(T performer) {
        performer.speak();
        performer.sit();
    }
}

public class DogsAndRobots {
    public static void main(String[] args) {
        PerformingDog d = new PerformingDog();
        Robot r = new Robot();
        Communicate.perform(d);
        Communicate.perform(r);
    }
} /* Output:
      Woof!
      Sitting
      Click!
      Clank!
   *////:~
```

Beachten Sie aber, daß die Funktionsweise der `perform()`-Methode keinen generischen Typ benötigt. Die Methode kann so überarbeitet werden, daß sie ein *`Performs`*-Objekt akzeptiert:

```
//: generics/SimpleDogsAndRobots.java
// Removing the generic; code still works.
```

[7] Die Implementierung generischer Typen bei Java wird hin und wieder als „generische Typen zweiter Klasse“ bezeichnet.

```
class CommunicateSimply {
    static void perform(Performs performer) {
        performer.speak();
        performer.sit();
    }
}

public class SimpleDogsAndRobots {
    public static void main(String[] args) {
        CommunicateSimply.perform(new PerformingDog());
        CommunicateSimply.perform(new Robot());
    }
} /* Output:
      Woof!
      Sitting
      Click!
      Clank!
   *///:~
```

In diesem Fall war einfach kein generischer Typ erforderlich, da die Klassen bereits das Interface *Performs* implementieren.

16.17 Ausgleich der fehlenden verborgenen Typisierung

[213] Die Tatsache, daß Java verborgene Typisierung nicht unterstützt bedeutet nicht, daß Sie Ihren bounded generic code nicht über mehrere Klassehierarchien hinweg verwenden können. Es ist also möglich, echten generischen Quelltext zu schreiben, erfordert aber zusätzliche Anstrengung.

16.17.1 Reflexion

[214] Der Reflexionsmechanismus ermöglicht einen Ansatz. Die `perform()`-Methode in diesem Beispiel verwendet verborgene Typisierung:

```
//: generics/LatentReflection.java
// Using Reflection to produce latent typing.
import java.lang.reflect.*;
import static net.mindview.util.Print.*;

// Does not implement Performs:
class Mime {
    public void walkAgainstTheWind() {}
    public void sit() { print("Pretending to sit"); }
    public void pushInvisibleWalls() {}
    public String toString() { return "Mime"; }
}

// Does not implement Performs:
class SmartDog {
    public void speak() { print("Woof!"); }
    public void sit() { print("Sitting"); }
    public void reproduce() {}
}

class CommunicateReflectively {
    public static void perform(Object speaker) {
        Class<?> spkr = speaker.getClass();
```

```
        try {
            try {
                Method speak = spkr.getMethod("speak");
                speak.invoke(speaker);
            } catch(NoSuchMethodException e) {
                print(speaker + " cannot speak");
            }
            try {
                Method sit = spkr.getMethod("sit");
                sit.invoke(speaker);
            } catch(NoSuchMethodException e) {
                print(speaker + " cannot sit");
            }
        } catch(Exception e) {
            throw new RuntimeException(speaker.toString(), e);
        }
    }
}

public class LatentReflection {
    public static void main(String[] args) {
        CommunicateReflectively.perform(new SmartDog());
        CommunicateReflectively.perform(new Robot());
        CommunicateReflectively.perform(new Mime());
    }
} /* Output:
      Woof!
      Sitting
      Click!
      Clank!
      Mime cannot speak
      Pretending to sit
   *///:~
```

Die Klassen `Dog`, `Robot` und `Mime` sind völlig von einander getrennt und haben außer `Object` keine Basisklasse und kein Interface gemeinsam. Der Reflexionsmechanis gestattet der statischen `CommunicateReflectively`-Methode `perform()`, zur Laufzeit zu ermitteln, ob die gewünschten Methoden vorhanden sind und sie gegebenenfalls aufzurufen. Die `perform()`-Methode kann sogar damit umgehen, daß die Klasse `Mime` nur eine der verlangten Methoden definiert und somit ihre Tätigkeit wenigstens teilweise zu verrichten.

16.17.2 Anwendung einer Methode auf eine Folge von Objekten

[215] Der Reflexionsmechanismus bietet zwar einige interessante Möglichkeiten, verbannt die Typprüfung aber in die Laufzeit. Wenn Sie die Typprüfung zur Übersetzungszeit bewerkstelligen können, sollten Sie von dieser Möglichkeit Gebrauch machen. Ist es aber möglich, Typprüfung zur Übersetzungszeit und verborgene Typisierung zu vereinen?

[216] Das folgende Beispiel untersucht das Problem. Stellen Sie sich vor, Sie möchten eine `apply()`-Methode anlegen, die eine beliebige Methode eines beliebigen Objektes aus einer Folge von Objekten aufruft. Dies ist eine Situation in der sich Interfaces nicht zu eigenen scheinen. Ihr Ziel ist, eine beliebige Methode auf den Objekten aus einer Kollektion aufzurufen und Interfaces schränken Sie zu stark ein, um eine „beliebige Methode“ beschreiben zu können. Wie läßt sich diese Situation in Java implementieren?

[217] Wir lösen das Problem zunächst mit Hilfe des Reflexionsmechanismus. Die Lösung ist durch die Argumentlisten variabler Länge der SE 5 ziemlich elegant:

```
//: generics/Apply.java
// {main: ApplyTest}
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Apply {
    public static <T, S extends Iterable<? extends T>>
        void apply(S seq, Method f, Object... args) {
        try {
            for(T t: seq)
                f.invoke(t, args);
        } catch(Exception e) {
            // Failures are programmer errors
            throw new RuntimeException(e);
        }
    }
}

class Shape {
    public void rotate() { print(this + " rotate"); }
    public void resize(int newSize) {
        print(this + " resize " + newSize);
    }
}

class Square extends Shape {}

class FilledList<T> extends ArrayList<T> {
    public FilledList(Class<? extends T> type, int size) {
        try {
            for(int i = 0; i < size; i++)
                // Assumes default constructor:
                add(type.newInstance());
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
}

class ApplyTest {
    public static void main(String[] args) throws Exception {
        List<Shape> shapes = new ArrayList<Shape>();
        for(int i = 0; i < 10; i++)
            shapes.add(new Shape());
        Apply.apply(shapes, Shape.class.getMethod("rotate"));
        Apply.apply(shapes, Shape.class.getMethod("resize", int.class), 5);
        List<Square> squares = new ArrayList<Square>();
        for(int i = 0; i < 10; i++)
            squares.add(new Square());
        Apply.apply(squares, Shape.class.getMethod("rotate"));
        Apply.apply(squares, Shape.class.getMethod("resize", int.class), 5);

        Apply.apply(new FilledList<Shape>(Shape.class, 10),
                    Shape.class.getMethod("rotate"));
        Apply.apply(new FilledList<Shape>(Square.class, 10),
                    Shape.class.getMethod("rotate"));
```

```
        SimpleQueue<Shape> shapeQ = new SimpleQueue<Shape>();
        for(int i = 0; i < 5; i++) {
            shapeQ.add(new Shape());
            shapeQ.add(new Square());
        }
        Apply.apply(shapeQ, Shape.class.getMethod("rotate"));
    }
} /* (Execute to see output) *////:~
```

[218] Wir nutzen in der Klasse `Apply` den glücklichen Umstand, daß die Klassen der Containerbibliothek von Java das Interface *`Iterable`* implementieren. Somit kann die `apply()`-Methode alle Argumente akzeptieren, die *`Iterable`* implementieren, darunter alle *`Collection`*-Klassen, wie etwa *`List`*. Die Methode akzeptiert aber auch jedes andere Objekt, dessen Klasse das Interface *`Iterable`* implementiert, beispielsweise die folgende Klasse `SimpleQueue`, die in der obigen `main()`-Methode verwendet wird:

```
//: generics/SimpleQueue.java
// A different kind of container that is Iterable
import java.util.*;

public class SimpleQueue<T> implements Iterable<T> {
    private LinkedList<T> storage = new LinkedList<T>();
    public void add(T t) { storage.offer(t); }
    public T get() { return storage.poll(); }
    public Iterator<T> iterator() {
        return storage.iterator();
    }
} ///:~
```

In der `catch`-Klausel in der Klasse `Apply` werden die Ausnahmen der `invoke()`-Methode in einer Ausnahme vom Typ `RuntimeException` verpackt und erneut ausgeworfen, da sie Programmierfehler anzeigen nach deren Eintreten sich kein sinnvoller Zustand zur Fortführung des Programms herbeiführen läßt.

[219] Beachten Sie die beschränkten Platzhalter, damit `Apply` und `FilledList` in allen erwünschten Situationen funktionieren. Entfernen Sie die Beschränkungen versuchshalber und Sie werden feststellen, daß einige applications von `Apply` und `FilledList` nicht funktionieren.

[220] Die Klasse `FilledList` stellt ein Dilemma dar. Die Verwendbarkeit des von `type` referenzierten Klassenobjektes setzt voraus, daß die entsprechende Klasse einen argumentlosen Konstruktor besitzt. Java hat keine Möglichkeit, die Existenz eines solchen Konstruktors zur Übersetzungszeit festzustellen, so daß die Angelegenheit zur Laufzeit geklärt wird. Ein häufig geäußerter Vorschlag, um Typprüfung zur Übersetzungszeit zu gewährleisten besteht darin, ein Fabrikinterface anzulegen, welches eine Methode deklariert, die Objekte erzeugt. Der Konstruktor der Klasse `FilledList` würde dann das Fabrikinterface anstelle des Typ-Etiketts akzeptieren. Das Problem an dieser Lösung ist, daß alle Klassen, deren Klassenobjekte Sie dem Konstruktor von `FilledList` übergeben, nun das Fabrikinterface implementieren müssen. Die meisten Klassen werden aber implementiert, ohne daß der Autor von Ihrem Fabrikinterface Kenntnis erhält. Wir besprechen später in diesem Kapitel noch eine Lösung mit Adaptern.

[221] Der obige Ansatz mit Typ-Etikett ist vielleicht ein vernünftiger Kompromiß (wenigstens für den Anfang). Bei diesem Ansatz ist die Klasse `FilledList` einfach genug, um verwendet und nicht verworfen zu werden. Da die Fehler zur Laufzeit gemeldet werden, brauchen Sie die Gewißheit, daß diese Fehler frühzeitig im Entwicklungsprozeß auftreten.

[222] Beachten Sie, daß der Ansatz mit dem Typ-Etikett in der Java-Literatur empfohlen wird, etwa in Gilad Brachas Publikation *Generics in the Java Programming Language* („Generics Tutorial“,

siehe Unterabschnitt „Literaturempfehlungen“ am Kapitelende). Der Autor schreibt: „It's an idiom that's used extensively in the new APIs for manipulating annotations, for example.“ Übersetzt etwa: „Eine Schreibweise, die beispielsweise bei den neuen Methoden zur Manipulation von Annotationen ausgiebig genutzt wird.“ Ich habe allerdings eine gewisse Unvereinbarkeit (*inconsistency*) hinsichtlich der Akzeptanz dieses Ansatzes beobachtet: Manche Entwickler ziehen den Fabrikansatz bei weitem vor (siehe früher in diesem Kapitel).

[223] So elegant die obige Java-Lösung auch sein mag, müssen wir berücksichtigen, daß der Reflexionsmechanismus (obwohl in den letzten Version deutlich beschleunigt) langsamer als eine Lösung ohne Reflexion sein kann, da zur Laufzeit viel Aktivität stattfindet. Das sollte Sie aber nicht davon abhalten, diesen Ansatz, wenigstens für den Anfang, zu wählen (damit Sie nicht den Einflüsterungen der voreiligen Optimierung erliegen). Dennoch bleibt ein Unterschied zwischen beiden Ansätzen.

Übungsaufgabe 40: (3) Legen Sie für jedes Haustier im Package `typeinfo.pets` eine `speak()`-Methode an. Ändern Sie das Beispiel *Apply.java* so, daß bei den Elementen einer heterogenen Kollektion von Haustieren die `speak()`-Methode aufgerufen wird. ■

16.17.3 Wenn Sie das richtige Interface nicht zur Verfügung haben

[224] Das obige Beispiel (*Apply.java*) profitiert vom eingebauten Interface *`Iterable`*, das genau unseren Erfordernissen entspricht. Aber was tun Sie im allgemeinen Fall, wenn es kein Interface gibt, das bereits zu Ihren Anforderungen paßt?

[225] Das folgende Beispiel verallgemeinert die Klasse `FilledList` und definiert eine parametrisierte `fill()`-Methode, die eine (leere) Kollektion erwartet und mittels eines Generators füllt. Beim Schreiben dieses Programms in Java tritt das Problem auf, daß des kein zweckdienliches Interface „*`Addable`*“ gibt, das die Rolle von *`Iterable`* im vorigen Beispiel übernimmt. Statt „jedes Element, dessen Klasse eine `add()`-Methode hat“ müssen Sie „jedes Element, dessen Klasse das Interface *`Collection`* implementiert“ wählen. Der resultierende Quelltext ist nicht besonders generisch, da er auf Untertypen von *`Collection`* beschränkt ist. Wenn ich eine Klasse wähle, die *`Collection`* nicht implementiert, funktioniert der generische Quelltext nicht:

```
//: generics/Fill.java
// Generalizing the FilledList idea
// {main: FillTest}
import java.util.*;

// Doesn't work with ''anything that has an add().'' There is
// no ''Addable'' interface so we are narrowed to using a
// Collection. We cannot generalize using generics in
// this case.
public class Fill {
    public static <T> void
    fill(Collection<T> collection, Class<? extends T> classToken, int size) {
        for(int i = 0; i < size; i++)
            // Assumes default constructor:
            try {
                collection.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
}

class Contract {
```

```
    private static long counter = 0;
    private final long id = counter++;
    public String toString() {
        return getClass().getName() + " " + id;
    }
}

class TitleTransfer extends Contract {}

class FillTest {
    public static void main(String[] args) {
        List<Contract> contracts = new ArrayList<Contract>();
        Fill.fill(contracts, Contract.class, 3);
        Fill.fill(contracts, TitleTransfer.class, 2);
        for(Contract c: contracts)
            System.out.println(c);
        SimpleQueue<Contract> contractQueue = new SimpleQueue<Contract>();
        // Won't work. fill() is not generic enough:
        // Fill.fill(contractQueue, Contract.class, 3);
    }
} /* Output:
      Contract 0
      Contract 1
      Contract 2
      TitleTransfer 3
      TitleTransfer 4
   *///:~
```

[226] Hier würde sich eine Implementierung parametrisierter Typen mit verborgener Typisierung auszahlen, da Sie in diesem Fall nicht mehr auf Gedeih und Verderb den früheren Design-Entscheidungen eines bestimmten Entwicklers einer Bibliothek ausgeliefert wären, also nicht jedesmal Ihr Programm überarbeiten müßten, wenn Sie auf eine neue Bibliothek stoßen, die Ihre Anforderungen nicht berücksichtigt (der Quelltext wäre wirklich generisch). Im obigen Fall sind wir, da die Java-Designer (verständlicherweise) keinen Bedarf für ein „*`Addable`*"-Interface gesehen haben, auf die Hierarchie unter dem Interface *`Collection`* beschränkt und die Klasse `SimpleQueue` ist nicht anwendbar, obwohl sie eine `add()`-Methode besitzt. Durch die Beschränkung auf die Untertypen von *`Collection`* ist der Quelltext nicht besonders generisch. Mit verborgener Typisierung wäre dies nicht der Fall.

16.17.4 Simulieren der verborgenen Typisierung mit Adaptern

[227] Die generischen Typen von Java unterstützen keine verborgene Typisierung. Andererseits brauchen wir etwas wie verborgene Typisierung, um Quelltext entwickeln zu können, der sich über die Grenzen von Klassenhierarchien hinweg verwenden läßt (also „generischen" Quelltext). Gibt es eine Möglichkeit, diese Einschränkung zu umgehen?

[228] Was würde verborgene Typisierung hier bewirken? Verborgene Typisierung bedeutet, den Standpunkt einzunehmen: „Mir ist gleichgültig, welcher Typ hier vorliegt, wenn er über die benötigten Methoden verfügt." Die verborgene Typisierung definiert effektiv ein *implizites Interface*, das die gewünschten Methoden „deklariert". Das Problem sollte sich also lösen lassen, indem wir das erforderliche Interface von Hand schreiben (Java kann uns diese Aufgabe nicht abnehmen).

[229] Das Schreiben der gewünschten Schnittstelle zu einer gegebenen Schnittstelle ist ein Beispiel für das Entwurfsmuster *Adapter*. Wir können Adapter verwenden, um vorhandene Klassen mit der gewünschten Schnittstelle auszustatten, wozu nur relativ wenige Anweisungen erforderlich sind. Das folgende Beispiel bedient sich der in Abschnitt 16.3 definierten `Coffee`-Hierarchie, um verschiedene

Möglichkeiten zur Entwicklung von Adaptern vorzuführen:

```
//: generics/Fill2.java
// Using adapters to simulate latent typing.
// {main: Fill2Test}
import generics.coffee.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

interface Addable<T> { void add(T t); }

public class Fill2 {
    // Classtoken version:
    public static <T> void
    fill(Addable<T> addable, Class<? extends T> classToken, int size) {
        for(int i = 0; i < size; i++)
            try {
                addable.add(classToken.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
    // Generator version:
    public static <T> void
    fill(Addable<T> addable, Generator<T> generator, int size) {
        for(int i = 0; i < size; i++)
            addable.add(generator.next());
    }
}

// To adapt a base type, you must use composition.
// Make any Collection Addable using composition:
class AddableCollectionAdapter<T> implements Addable<T> {
    private Collection<T> c;
    public AddableCollectionAdapter(Collection<T> c) {
        this.c = c;
    }
    public void add(T item) { c.add(item); }
}

// A Helper to capture the type automatically:
class Adapter {
    public static <T> Addable<T> collectionAdapter(Collection<T> c) {
        return new AddableCollectionAdapter<T>(c);
    }
}

// To adapt a specific type, you can use inheritance.
// Make a SimpleQueue Addable using inheritance:
class AddableSimpleQueue<T> extends SimpleQueue<T> implements Addable<T> {
    public void add(T item) { super.add(item); }
}

class Fill2Test {
    public static void main(String[] args) {
        // Adapt a Collection:
        List<Coffee> carrier = new ArrayList<Coffee>();
        Fill2.fill(new AddableCollectionAdapter<Coffee>(carrier), Coffee.class, 3);
        // Helper method captures the type:
        Fill2.fill(Adapter.collectionAdapter(carrier), Latte.class, 2);
```

```
        for(Coffee c: carrier)
            print(c);
        print("-----------");
        // Use an adapted class:
        AddableSimpleQueue<Coffee> coffeeQueue = new AddableSimpleQueue<Coffee>();
        Fill2.fill(coffeeQueue, Mocha.class, 4);
        Fill2.fill(coffeeQueue, Latte.class, 1);
        for(Coffee c: coffeeQueue)
            print(c);
    }
} /* Output:
      Coffee 0
      Coffee 1
      Coffee 2
      Latte 3
      Latte 4
      -----------
      Mocha 5
      Mocha 6
      Mocha 7
      Mocha 8
      Latte 9
   *///:~
```

[230] Die `fill()`-Methode der Klasse `Fill2` verlangt keine Kollektion, wie die Version im vorigen Beispiel. Statt dessen erwartet die Methode ein Argument dessen Klasse das Interface ***Addable*** implementiert, wobei ***Addable*** alleine für dieses Beispiel geschrieben ist. Das Interface ***Addable*** repräsentiert die verborgene Typisierung, die der Compiler prüfen soll.

[231] Die zweite Version der überladenen `fill()`-Methode erwartet ein Argument vom Typ ***Generator*** anstelle eines Typ-Etiketts. Der Compiler garantiert zur Übersetzungszeit, daß Sie nur ein Argument vom Typ ***Generator*** übergeben können, so daß keine Ausnahme ausgeworfen werden kann.

[232] Die erste Adapterklasse, `AddableCollectionAdapter`, arbeitet mit dem Basistyp ***Collection***, akzeptiert also jede Implementierung des Interfaces ***Collection***. Diese Version speichert die ***Collection***-Referenz einfach und verwendet sie zur Implementierung der `add()`-Methode.

[233] Wenn Sie einen spezialisierten Typ anbinden wollen, der unterhalb des Basistyps steht, können Sie beim Schreiben des Adapters durch Ableitung Arbeit sparen (siehe `AddableSimpleQueue`).

[234] In der `main()`-Methode der Klasse `Fill2` sehen Sie die verschiedenen Adapter bei der Arbeit. Zuerst wird ein ***Collection***-Typ per `AddableCollectionAdapter` angebunden. Beim nächsten `fill()`-Aufruf ist eine generische Hilfsmethode zwischengeschaltet, die den Typ abfängt, so daß er nicht explizit angegeben werden muß. Dieser bequeme Trick liefert eleganteren Quelltext.

[235] Next, the pre-adapted AddableSimpleQueue is used. Note that in both cases the adapters allow the classes that previously didn't implement Addable to be used with Fill2.fill().

[236] Die in diesem Unterabschnitt gezeigte Verwendung von Adaptern scheint das Fehlen der verborgenen Typisierung auszugleichen und gestattet Ihnen, echten generischen Quelltext zu schreiben. Der Ansatz erfordert allerdings einen zusätzlichen Schritt und führt eine Konstruktion ein, die sowohl vom Autor als auch vom Nutzer der Bibliothek verstanden werden muß. Außerdem nehmen weniger erfahrene Programmierer das Konzept eventuell nicht bereitwillig an. Der Wert des zusätzlichen Schritts besteht darin, den generischen Quelltext leichter anwendbar zu machen.

Übungsaufgabe 41: (1) Ändern Sie das Beispiel *Fill2.java* so, daß die Klassen aus dem Package

`typeinfo.pets` anstelle der `Coffee`-Klassen verwendet werden. ■

16.18 Funktionsobjekte als Strategien

[237] Das letzte Beispiel dieses Abschnitts zeigt echten generischen Quelltext, angelegt mit Hilfe des Adapteransatzes aus dem vorigen Unterabschnitt. Das Beispiel begann bei mir vor langer Zeit mit dem Versuch, die Summe einer Folge von Elementen (eines beliebigen summierbaren Typs) zu bilden, hat sich aber durch funktionalen Programmierstil zur Ausführung allgemeiner Operationen hin entwickelt.

[238] Wenn Sie sich nur den Vorgang des Hinzufügens von Objekten ansehen, können Sie erkennen, daß ein Fall einer allgemeinen Operation vorliegt, der in keiner Basisklasse erfaßt ist, die wir angeben können: Einmal können Sie den `+`-Operator nehmen, ein anderes Mal eine `add`-Methode. So stellt sich die Situation dar, wenn Sie generischen Quelltext schreiben wollen, der über mehrere Klassen hinweg anwendbar sein soll. Hinzu kommt, wie im vorliegenden Fall, daß bereits mehrere Klassen existieren, die wir nicht anpassen können. Selbst wenn Sie den vorliegenden Fall auf die von `Number` abgeleiteten Klassen einschränken könnten, sagt diese Basisklasse nicht über „Hinzufügbarkeit“ aus.

[239] Die Lösung besteht in der Anwendung des *Strategy*-Entwurfsmusters, welches zu eleganterem Quelltext führt, da es „das veränderliche Ding“ vollkommen in einem Funktionsobjekt[8] (*function object*) isoliert. Ein Funktionsobjekt ist ein Objekt, das sich in gewisser Weise wie eine Funktion verhält: Typischerweise existiert eine interessante Methode (bei Sprachen, die Operatorüberladung unterstützen, *you can make the call to this method look like an ordinary method call*). Der Nutzen eines Funktionsobjektes besteht darin, das es, im Gegensatz zu einer gewöhnlichen Methode herumgereicht werden kann und seinen Zustand zwischen zwei Methodenaufrufen beibehält. Natürlich läßt sich so etwas mit jeder Methode einer Klasse bewerkstelligen, aber das Funktionsobjekt ist (wie jedes Entwurfsmuster) hauptsächlich durch seinen Zweck bestimmt. Der Zweck besteht hier darin, etwas zu erzeugen, was sich wie eine einzelne Methode verhält, die Sie umherreichen können, ist also eng verbunden und manchmal kaum zu unterscheiden vom Entwurfsmuster *Strategy*.

[240] Bei vielen Entwurfsmustern sind die Abgrenzungen undeutlich; so auch hier. Wir erzeugen Funktionsobjekte, die Adapterfunktionalität ausüben und übergeben Sie an Methoden, wo sie als Strategien verwendet werden.

[241] Diesem Ansatz folgend habe ich die verschiedenen generischen Methoden angelegt, die ich ursprünglich vorgesehen hatte und weitere hinzugefügt. Hier ist das Ergebnis:

```
//: generics/Functional.java
import java.math.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Different types of function objects:
interface Combiner<T> { T combine(T x, T y); }
interface UnaryFunction<R,T> { R function(T x); }
interface Collector<T> extends UnaryFunction<T,T> {
    T result(); // Extract result of collecting parameter
}
interface UnaryPredicate<T> { boolean test(T x); }

public class Functional {
```

[8] Funktionsobjekte werden gelegentlich als „Funktoren“ bezeichnet. Ich verwende den Begriff „Funktionsobjekt“ statt „Funktor“, da der letztere in der Mathematik eine bestimmte und abweichende Bedeutung hat.

```
// Calls the Combiner object on each element to combine
// it with a running result, which is finally returned:
public static <T> T
    reduce(Iterable<T> seq, Combiner<T> combiner) {
    Iterator<T> it = seq.iterator();
    if(it.hasNext()) {
        T result = it.next();
        while(it.hasNext())
            result = combiner.combine(result, it.next());
        return result;
    }
    // If seq is the empty list:
    return null; // Or throw exception
}
// Take a function object and call it on each object in
// the list, ignoring the return value. The function
// object may act as a collecting parameter, so it is
// returned at the end.
public static <T> Collector<T>
    forEach(Iterable<T> seq, Collector<T> func) {
    for(T t : seq)
        func.function(t);
    return func;
}
// Creates a list of results by calling a
// function object for each object in the list:
public static <R,T> List<R>
    transform(Iterable<T> seq, UnaryFunction<R,T> func) {
    List<R> result = new ArrayList<R>();
    for(T t : seq)
        result.add(func.function(t));
    return result;
}
// Applies a unary predicate to each item in a sequence,
// and returns a list of items that produced ''true'':
public static <T> List<T>
    filter(Iterable<T> seq, UnaryPredicate<T> pred) {
    List<T> result = new ArrayList<T>();
    for(T t : seq)
        if(pred.test(t))
            result.add(t);
    return result;
}
// To use the above generic methods, we need to create
// function objects to adapt to our particular needs:
static class IntegerAdder implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
        return x + y;
    }
}
static class
    IntegerSubtracter implements Combiner<Integer> {
    public Integer combine(Integer x, Integer y) {
        return x - y;
    }
}
static class
```

```
        BigDecimalAdder implements Combiner<BigDecimal> {
        public BigDecimal combine(BigDecimal x, BigDecimal y) {
            return x.add(y);
        }
    }
    static class
        BigIntegerAdder implements Combiner<BigInteger> {
        public BigInteger combine(BigInteger x, BigInteger y) {
            return x.add(y);
        }
    }
    static class
        AtomicLongAdder implements Combiner<AtomicLong> {
        public AtomicLong combine(AtomicLong x, AtomicLong y) {
            // Not clear whether this is meaningful:
            return new AtomicLong(x.addAndGet(y.get()));
        }
    }
    // We can even make a UnaryFunction with an ''ulp''
    // (Units in the last place):
    static class BigDecimalUlp
        implements UnaryFunction<BigDecimal,BigDecimal> {
        public BigDecimal function(BigDecimal x) {
            return x.ulp();
        }
    }
    static class GreaterThan<T extends Comparable<T>>
        implements UnaryPredicate<T> {
        private T bound;
        public GreaterThan(T bound) { this.bound = bound; }
        public boolean test(T x) {
            return x.compareTo(bound) > 0;
        }
    }
    static class MultiplyingIntegerCollector
        implements Collector<Integer> {
        private Integer val = 1;
        public Integer function(Integer x) {
            val *= x;
            return val;
        }
        public Integer result() { return val; }
    }
    public static void main(String[] args) {
        // Generics, varargs & boxing working together:
        List<Integer> li = Arrays.asList(1, 2, 3, 4, 5, 6, 7);
        Integer result = reduce(li, new IntegerAdder());
        print(result);

        result = reduce(li, new IntegerSubtracter());
        print(result);

        print(filter(li, new GreaterThan<Integer>(4)));

        print(forEach(li, new MultiplyingIntegerCollector()).result());

        print(forEach(filter(li, new GreaterThan<Integer>(4)),
            new MultiplyingIntegerCollector()).result());

        MathContext mc = new MathContext(7);
```

```
        List<BigDecimal> lbd =
            Arrays.asList(new BigDecimal(1.1, mc), new BigDecimal(2.2, mc),
                          new BigDecimal(3.3, mc), new BigDecimal(4.4, mc));
        BigDecimal rbd = reduce(lbd, new BigDecimalAdder());
        print(rbd);

        print(filter(lbd, new GreaterThan<BigDecimal>(new BigDecimal(3))));

        // Use the prime-generation facility of BigInteger:
        List<BigInteger> lbi = new ArrayList<BigInteger>();
        BigInteger bi = BigInteger.valueOf(11);
        for(int i = 0; i < 11; i++) {
            lbi.add(bi);
            bi = bi.nextProbablePrime();
        }
        print(lbi);

        BigInteger rbi = reduce(lbi, new BigIntegerAdder());
        print(rbi);
        // The sum of this list of primes is also prime:
        print(rbi.isProbablePrime(5));

        List<AtomicLong> lal =
            Arrays.asList(new AtomicLong(11), new AtomicLong(47),
                          new AtomicLong(74), new AtomicLong(133));
        AtomicLong ral = reduce(lal, new AtomicLongAdder());
        print(ral);

        print(transform(lbd,new BigDecimalUlp()));
    }
} /* Output:
      28
      -26
      [5, 6, 7]
      5040
      210
      11.000000
      [3.300000, 4.400000]
      [11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]
      311
      true
      265
      [0.000001, 0.000001, 0.000001, 0.000001]
   *///:~
```

[242] Die Klasse `Functional` beginnt mit der Definition einiger Interfaces für die verschiedenen Typen von Funktionsobjekten. Ich habe die Klassen der Funktionsobjekte bei Bedarf geschrieben, während ich die verschiedenen Methoden entwickelt und die Notwendigkeit entdeckt habe. Das Interface *`Combiner`* wurde von einem anonymen Leser eines Artikels auf meiner Website beigetragen und abstrahiert von den spezifischen Einzelheiten beim Versuch weg, zwei Objekte zu addieren, indem sie einfach vorgibt, daß die Objekte auf irgendeine Weise kombiniert werden. Die Klassen `IntegerAdder` und `IntegerSubtracter` implementieren das Interface *`Combiner`*.

[243] Eine unäre Funktion (*`UnaryFunction`*) erwartet genau ein Argument und erzeugt ein Ergebnis, wobei Argument und Rückgabewert nicht vom selben Typ sein müssen. Das Interface *`Collector`* repräsentiert einen collecting parameter und Sie können das Ergebnis am Ende extrahieren. Das Interface *`UnaryPredicate`* liefert ein `boolean`-Ergebnis. Es gibt noch weitere Typen von Funktionsobjekten, aber die obige Auswahl genügt, um das Prinzip vorzuführen.

[244] Die Klasse `Functional` enthält eine Anzahl generischer Methoden, die Funktionsobjekte auf Folgen von Elementen anwenden. Die `reduce()`-Methode wendet die Funktion eines Funktionsobjektes vom Typ *`Combiner`* auf jedes Element einer Folge an und erzeugt auf diese Weise ein Ergebnis.

[245] Die `forEach()`-Methode erwartet ein Funktionsobjekt vom Typ *`Collector`* und wendet seine Funktion auf jedes Element an, beachtet aber das Ergebnis des Funktionsaufrufs nicht. Diese Funktion kann einerseits wegen ihres Seiteneffektes aufgerufen werden (diese wäre allerdings keine funktionale Programmierung mehr, wohl aber nützlich) oder das *`Collector`*-Funktionsobjekt kann seinen internen Zustand aufrechterhalten und kann, wie in diesem Beispiel, als collecting parameter verwendet werden.

[246] Die `transform()`-Methode erzeugt durch Aufrufen einer unären Funktion auf jedem Objekt einer Folge und Abfangen des Ergebnisses eine Liste.

[247] Schließlich wendet die `filter()`-Methode ein unäres Prädikat (*`UnaryPredicate`*) auf jedes Objekt einer Folge an und speichert diejenigen Elemente, deren Test `true` liefert in einer Liste, die sie zurückgibt.

[248] Sie können weitere generische Funktionen definieren. Die STL von C++ enthält viele generische Funktionen. Es gibt auch quelloffene Bibliotheken für Java, die sich dieser Aufgabe verschrieben haben, beispielsweise die JGA (Generic Algorithms for Java; siehe *http://jga.sourceforge.net*).

[249] Bei C++ kümmert sich die verborgene Typisierung beim Funktionsaufruf um matching up der Operationen. Bei Java müssen wir Funktionsobjekte schreiben, um die generischen Methoden an unsere Anforderungen anzupassen. Der nächste Abschnitt der Klasse zeigt verschiedene Implementierungen von Funktionsobjekten. Beachten Sie, daß beispielsweise die Klasse `IntegerAdder` und `BigDecimalAdder` die gleiche Aufgabe erfüllen, nämlich zwei Objekte zu addieren, in dem sie die ihrem Typ entsprechenden Operationen aufrufen. Somit sind die Entwurfsmuster *Adapter* und *Strategy* kombiniert.

[250] In der `main()`-Methode sehen Sie, daß bei jedem Methodenaufruf eine Folge von Elementen zusammen mit einem Funktionsobjekt übergeben wird. Die Ausdrücke können ziemlich komplex werden, zum Beispiel:

```
forEach(filter(li, new GreaterThan(4)),
    new MultiplyingIntegerCollector()).result()
```

Dieser Aufruf erzeugt zunächst eine Liste aller Elemente in der von `li` referenzierten Liste, die größer als 4 sind, wendet anschließend das Funktionsobjekt `MultiplyingIntegerCollector` auf diese Liste an und extrahiert das Ergebnis per `result()`. Die verbleibenden Einzelheiten werden nicht erläutert. Wahrscheinlich verstehen Sie was geschieht, wenn Sie den Quelltext lesen.

Übungsaufgabe 42: (5) Legen Sie zwei separate Klassen an, die keinerlei Gemeinsamkeiten haben. Jede Klasse speichert einen Wert und verfügt zumindest über Abfrage- und Änderungsmethoden für diesen. Ändern Sie das Beispiel *Functional.java* so, daß es funktionale Operationen auf Kollektionen aus Elementen dieser Klassen ausführt (die Operationen müssen nicht arithmetisch sein, wie in *Functional.java*). ■

16.19 Zusammenfassung

[251] Nachdem ich C++ Templates von Anfang an gelehrt habe, habe ich die folgende Auseinandersetzung wohl länger befeuert, als die meisten Kollegen. Ich habe erst vor kurzem aufgehört, mich

zu fragen, ob diese Diskussion überhaupt berechtigt ist, das heißt wie oft das Problem tatsächlich auftritt, das wir in dieser Zusammenfassung des Kapitels über die generische Typen von Java besprechen wollen.

[252] Die Diskussion beginnt so: Die Containerklassen unter den Interfaces *`List`*, *`Set`*, *`Map`* und so weiter (siehe Kapitel 12 und 18) gehören zu den natürlichsten Anwendungsfällen für einen generischen Typmechanismus. Vor der SE 5 wurde ein einem Container übergebenes Objekt in den Typ `Object` umgewandelt, das heißt die Typinformation ging verloren. Bei der Entnahme eines Objektes aus dem Container mußte es in seinen eigentlichen Typ zurück umgewandelt werden. Wir haben eine Liste (*`List`*) von `Cat`-Objekten verwendet (eine andere Version mit Äpfeln und Orangen steht am Anfang von Kapitel 12). Ohne die generische Version des Containers aus der SE 5 wurden Elemente vom Typ `Object` gespeichert und Elemente vom Typ `Object` entnommen, so daß mühelos ein `Dog`-Objekt in eine Liste von `Cat`-Objekten eingesetzt werden konnte.

[253] Dennoch gestattete das vorgenerische Java keinen Mißbrauch der in einem Container gespeicherten Objekte. Enthielt ein Container für `Cat`-Objekte ein `Dog`-Objekt, wobei jedes entnommene Objekt als `Cat` behandelt wurde, so wurde bei Entnahme des `Dog`-Objektes und dem Versuch, es in `Cat` umzuwandeln eine Ausnahme ausgeworfen. Das Problem wurde bemerkt, allerdings erst zur Laufzeit im Gegensatz zur Übersetzungszeit.

[254] In den früheren Auflagen dieses Buchs bin ich fortgefahren:

> „Das ist mehr als einfach ärgerlich. Dieses Verhalten kann zu schwer auffindbaren Fehlern führen. Wenn einer oder mehrere Programmteile Objekte in einen Container einsetzen und Sie in einem anderen Teil lediglich per Ausnahme erfahren, daß der Container ein typfremdes Element enthält, müssen Sie sich auf die Suche machen, an welcher Stelle dieses Element eingetragen wurde.“

Im Laufe der Diskussion begann ich mir Gedanken zu machen, zunächst: Wie oft tritt ein solcher Fehler auf? Ich kann mich nicht daran erinnern, daß mir dieser Fehler jemals unterlaufen wäre und keiner der Kollegen, die ich bei Konferenzen befragte, gab an, diesen Fall selbst erlebt zu haben. Ein Buch enthielt ein Beispiel mit einer Liste von `String`-Objekten, die von einem Feld namens `files` referenziert wurde. Es wäre ein völlig natürlicher Gedanke, ein `File`-Objekt in dieser Liste zu speichern, das heißt, daß `fileNames` eventuell ein besserer Bezeichner gewesen wäre. Gleichgültig wieviele Typprüfungen Java durchführt, ist es möglich, undurchsichtige Programme zu schreiben und ein schlecht geschriebenes Programm bleibt schlecht geschrieben, auch wenn es sich übersetzen läßt. Die meisten Programmierer verwenden wohl sinnvoll gewählte Bezeichner wie `cats` für Ihre Container, die eine erkennbare Warnung für den Programmierer sind, etwas anderes als `Cat`-Objekte einzutragen. Selbst wenn ein typfremdes Objekt eingesetzt wird: Wie lange würde es unentdeckt bleiben? Wenn Sie beginnen, das Programm mit realistischen Daten zu testen, würde die Ausnahme schnell hervorgerufen.

[255] Ein Autor behauptet gar, daß ein solcher Fehler über Jahre hinweg verborgen bleiben kann. Ich kann mich allerdings an keine Flut von Berichten erinnern, in der Programmierer große Schwierigkeiten hatten, ein `Dog`-Objekt in einer Liste von `Cat`-Objekten zu entdecken oder solche Fehler häufig gemacht haben. In Kapitel 22 lernen Sie dagegen, daß sich bei Anwesenheit von Threads, leicht und häufig Fehler einschleichen, die extrem selten auftreten, wobei Sie nur eine vage Ahnung hinsichtlich der Ursache haben. Ist also das `Dog`-Objekt in der Liste von `Cat`-Objekten tatsächlich der Grund, aus dem Java um diese ~~very significant~~ und ziemlich komplexe Eigenschaft ergänzt wurde?

[256] Der Grund für die Implementierung der universellen Spracheigenschaft „generische Typen“ (unabhängig von ihrer Java-spezifischen Ausprägung) ist nach meiner Überzeugung *Ausdrucksfähigkeit* (*expressiveness*) und nicht nur Typsicherheit bei Containern. Typsichere Container sind eine

Nebenwirkung der Fähigkeit, universelleren Quelltext schreiben zu können.

[257] Das Argument „`Dog`-Objekt in einer Liste von `Cat`-Objekten“ ist fragwürdig, obwohl es häufig vorgebracht wird, um die generischen Typen zu rechtfertigen. Wie ich bereits am Anfang des Kapitels festgestellt habe, glaube ich nicht, daß dieses Argument der Kern des Konzeptes der generischen Typen ist. Stattdessen gestatten generische Typen, ihrem Namen entsprechend, generischeren Quelltext zu schreiben, der weniger durch die Typen beschränkt ist, auf die er angewendet werden kann. Ein Stück Quelltext kann auf diese Weise auf mehr Typen angewendet werden. Sie haben in diesem Kapitel gesehen, daß es relativ einfach ist, echte generische „Behälter“-Klassen zu schreiben (die Containerklassen sind solche Klassen), wobei das Schreiben einer generischen Methode, die ihr generisches Argument modifiziert, zusätzlichen Aufwand erfordert. Der zusätzliche Aufwand liegt sowohl auf der Seite des Autors der generischen Klasse, als auch auf der Seite des Programmiers, die sich mit Konzept und Implementierung des *Adapter*-Entwurfsmusters vertraut machen müssen. Dieser Zusatzaufwand vermindert die einfache Anwendung dieser Eigenschaft und bewirkt eventuell, daß sie nicht an Stellen angewendet wird, an denen sie sich als wertvoll erweisen könnte.

[258] Beachten Sie auch, daß einige Container, da die generischen Typen nachträglich in die Sprache eingebaut, statt von Anfang an integriert wurden, nicht so robust sein können, als sie sollten. Nehmen Sie *`Map`* als Beispiel, insbesondere die Methode `containsKey(Object key)` und `get(Object key)`. Wären generische Typen zum Zeitpunkt des Entwurfs dieser Klassen verfügbar gewesen, so würden ihre Methoden parametrisierte Typen statt `Object` verwenden, also die Prüfungen zur Übersetzungszeit leisten, die die generischen Typen liefern sollen. Beim Typ `map` von C++ beispielsweise, wird der Parametertyp zur Übersetzungszeit geprüft.

[259] Eines ist klar: Die Aufnahme eines generischen Typmechanismus in eine späte Version einer Programmiersprache, die sich bereits allgemein durchgesetzt hat, ist ein sehr schmutziges Vorhaben und läßt sich nicht bewerkstelligen, ohne Schmerzen zu verursachen. Bei C++ wurden die Templates in die erste ISO-Version eingebettet, waren also von Anfang an Teil der Sprache (auch dieser Schritt war nicht schmerzlos, da bereits eine frühere Version ohne Template in Gebrauch war, bevor die erste Version von Standard C++ erschien). Bei Java wurden die generischen Typen erst eingeführt, nachdem die Sprache fast zehn Jahre zuvor herausgegeben worden war, so daß erhebliche Gesichtspunkte hinsichtlich der Migration zu den generischen Typen hin in Betracht gezogen werden mußten und sich deutlich auf das Design der generischen Typen ausgewirkt haben. Im Ergebnis leiden Sie, der Programmierer, unter dem Mangel an Voraussicht der Java-Designer bei der Arbeit an Java 1.0. Als die Designer die erste Version von Java ins Leben riefen, wußten sie selbstverständlich von den Templates bei C++. Und obwohl sie die Aufnahme eines solchen Konzeptes erwogen hatten, entschieden sie sich aus irgend einem Grund dafür, das Konzept zu verwerfen (es gibt Anzeichen dafür, daß die Designer in Eile waren). Im Ergebnis leiden sowohl die Sprache als auch die Programmierer, die damit arbeiten. Nur die Zeit kann zeigen, welchen Einfluß der Java-Ansatz für generische Typen letztendlich auf die Sprache hat.

[260] Es gibt Sprachen mit einem saubereren und weniger durchschlagenden Ansatz für parametrisierte Typen, darunter Nice (siehe *http://nice.sourceforge.net*; die Sprache generiert Java-Bytecode und nutzt existierende Java-Bibliotheken) und NextGen (siehe *http://www.cs.rice.edu/~javaplt/nextgen*). Es ist nicht unmöglich, sich eine solche Sprache als Nachfolger von Java vorzustellen, die präzise dem Ansatz von C++ in Bezug auf C folgen: Vorhandenes verwenden und verbessern.

16.19.1 Literaturempfehlungen

[261] Das „Generics Tutorial“ von Gilad Bracha (*Generics in the Java Programming Language*) ist eine Einführung in die generischen Typen bei Java. Sie finden den Text unter der Webadresse *http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf*.

[262] Angelika Langers *Java Generics FAQ* ist eine nützliche Quelle: *http://www.angelikalanger-.com/GenericsFAQ/JavaGenericsFAQ.html*.

[263] Weiterführende Informationen über Platzhalter finden Sie in *Adding Wildcards to the Java Programming Language* von Mads Torgersen, Christian Plesner Hansen, Erik Ernst, Peter von der Ahe, Gilad Bracha and Neal Gafter unter der Webadresse *http://www.jot.fm/issues/issue_2004_12/article5*.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 17

Arrays

Inhaltsübersicht

Am Ende von Kapitel 12 haben Sie gelernt, ein Array anzulegen und zu initialisieren.

[0] Einfach betrachtet, erzeugen Sie ein Arrayobjekt, setzen Elemente ein, und wählen Elemente mit Hilfe eines ganzzahligen Indexes aus. Die Länge eines Arrays ist unveränderlich. In der Regel ist das alles, was Sie wissen müssen, aber hin und wieder brauchen Sie anspruchsvollere Arrayoperationen oder müssen sich zwischen einem Array und einem flexibleren Container entscheiden. Dieses Kapitel vermittelt Ihnen einen tieferen Einblick in das Wesen von Arrays.

17.1 Eigenschaften von Arrays

[1−2] Es gibt mehrere Möglichkeiten, um Objekte zu speichern. Wodurch zeichnen sich Arrays aus? Arrays unterscheiden sich durch drei Eigenschaften von anderen Containertypen: Effizienz, Typ und die Fähigkeit, Elemente primitiven Typs zu enthalten. Das Array ist die effizienteste Möglichkeit bei Java, um eine Reihe von Objektreferenzen zu speichern und wahlfreien Zugriff auf die einzelnen Elemente zu gestatten. Das Array ist eine einfache lineare Aneinanderreihung, wodurch der Zugriff

auf die Elemente schnell geschieht. Der Preis für die Zugriffsgeschwindigkeit ist, daß die Länge eines Arrayobjektes feststeht und während seines Lebenszyklus' nicht verändert werden kann. Nach dem Studium von Kapitel 12 liegt ein `ArrayList`-Objekt als Alternative nahe, das automatisch mehr Speicherplatz allokiert, indem es ein neues [Array!] erzeugt und alle Referenzen vom alten [Array] in das neue kopiert. Auch wenn Sie im allgemeinen ein `ArrayList`-Objekt einem Array vorziehen sollten, bringt die Flexibilität auch Unkosten mit sich: Ein `ArrayList`-Objekt ist meßbar weniger effizient als ein Array.

[3] Sowohl Arrays als auch Container garantieren, daß kein Mißbrauch möglich ist. Gleichgültig ob Sie ein Array oder einen Container wählen, wird eine Ausnahme vom Typ `RuntimeException` ausgeworfen, wenn Sie durch einen Programmierfehler die Indexgrenzen überschreiten.

[4] Vor der Einführung der generischen Typen behandelten die Containerklassen ihre Elemente, als ob sie keinen bestimmten Typ hätten, das heißt jedem Elemente wurde der Typ `Object` zugeschrieben, die Wurzelklasse aller Java-Klassen. Arrays sind den vor-generischen Containern überlegen, da sie unter Angabe eines bestimmten Elementtyps erzeugt werden. Dadurch erhalten Sie Typprüfung zur Übersetzungszeit und sind vor dem Einsetzen typfremder Elemente sowie Typverwechselungen beim Abfragen von Elementen geschützt. Selbstverständlich verhindert Java entweder zur Übersetzungs- oder zur Laufzeit, daß Sie eine nicht definierte Methode aufrufen. Der eine Weg ist nicht riskanter als der andere, aber es ist besser, wenn der Compiler auf den Fehler hinweist. Dadurch ist es weniger wahrscheinlich, daß der Anwender von einer Ausnahme überrascht wird.

[5] Ein Array kann Elemente primitiven Typs speichern, nicht aber ein vor-generischer Container. Ein generischer Container kann dagegen den Typ seiner Elemente festlegen und prüfen. Durch Autoboxing kann sich ein Container verhalten, als wäre er in der Lage, Elemente primitiven Typs aufzunehmen, da die Umwandlung automatisch vor sich geht. Das folgende Beispiel vergleicht Arrays und generische Container:

```
//: arrays/ContainerComparison.java
import java.util.*;
import static net.mindview.util.Print.*;

class BerylliumSphere {
    private static long counter;
    private final long id = counter++;
    public String toString() { return "Sphere " + id; }
}

public class ContainerComparison {
    public static void main(String[] args) {
        BerylliumSphere[] spheres = new BerylliumSphere[10];
        for(int i = 0; i < 5; i++)
            spheres[i] = new BerylliumSphere();
        print(Arrays.toString(spheres));
        print(spheres[4]);

        List<BerylliumSphere> sphereList = new ArrayList<BerylliumSphere>();
        for(int i = 0; i < 5; i++)
            sphereList.add(new BerylliumSphere());
        print(sphereList);
        print(sphereList.get(4));

        int[] integers = { 0, 1, 2, 3, 4, 5 };
        print(Arrays.toString(integers));
        print(integers[4]);

        List<Integer> intList =
            new ArrayList<Integer>(Arrays.asList(0, 1, 2, 3, 4, 5));
```

```
        intList.add(97);
        print(intList);
        print(intList.get(4));
    }
} /* Output:
      [Sphere 0, Sphere 1, Sphere 2, Sphere 3, Sphere 4,
      null, null, null, null, null]
      Sphere 4
      [Sphere 5, Sphere 6, Sphere 7, Sphere 8, Sphere 9]
      Sphere 9
      [0, 1, 2, 3, 4, 5]
      4
      [0, 1, 2, 3, 4, 5, 97]
      4
   *///:~
```

Beide Möglichkeiten zur Speicherung von Objekten sind typgeprüft und der einzige erkennbare Unterschied besteht darin, daß Arrays den `[]`-Operator für den Zugriff auf ihre Elemente benötigen, während Listen Methoden wie `add()` und `get()` verwenden. Die Ähnlichkeit zwischen Arrays und `ArrayList`-Objekten ist beabsichtigt, so daß conceptually leicht zwischen beiden gewechselt werden kann. Wie Sie aber in Kapitel 12 gelernt haben, verfügen Container über erheblich mehr Funktionalität als Arrays.

[6] Nach dem Erscheinen des Autoboxingmechanismus sind Container beinahe so einfach auf Elemente primitiven Typs anzuwenden als Arrays. Der einzige verbleibende Vorteil der Arrays ist ihre Effizienz. Bei allgemeineren Programmierproblemen können Sie Arrays zu sehr einschränken. In solchen Fällen wählen Sie eine Containerklasse.

17.2 Arrays sind vollwertige Objekte

[7] Ungeachtet des Arraytyps mit dem Sie arbeiten, ist der Arraybezeichner eine Referenz auf ein echtes Objekt im dynamischen Speicher. Dieses Objekt beinhaltet die Referenzen auf die anderen Objekte und wird entweder implizit als Teil Initialisierungsyntax oder explizit durch einen `new`-Ausdruck erzeugt. Das `length`-Feld eines Arrayobjektes ist die einzige Komponente, auf die Sie zugreifen können und gibt an, wieviele Elemente in diesem Arrayobjekt gespeichert werden können. Der `[]`-Operator ist Ihre einzige andere Zugriffsmöglichkeit auf das Arrayobjekt.

[8] Das folgende Beispiel faßt die verschiedenen Wege zusammen, auf denen Sie ein Array initialisieren und einer Referenzvariable zuweisen können. Das Beispiel zeigt außerdem, daß Arrays von Objekten und Arrays von Elementen primitiven Typs hinsichtlich ihres Gebrauchs weitestgehend identisch sind. Der einzige Unterschied besteht darin, daß Arrays von Objekten Referenzen beinhalten, während Arrays von Elementen primitiven Typs die Werte direkt speichern.

```
//: arrays/ArrayOptions.java
// Initialization & re-assignment of arrays.
import java.util.*;
import static net.mindview.util.Print.*;

public class ArrayOptions {
    public static void main(String[] args) {
        // Arrays of objects:
        BerylliumSphere[] a; // Local uninitialized variable
        BerylliumSphere[] b = new BerylliumSphere[5];
        // The references inside the array are
        // automatically initialized to null:
```

```
        print("b: " + Arrays.toString(b));
        BerylliumSphere[] c = new BerylliumSphere[4];
        for(int i = 0; i < c.length; i++)
            if(c[i] == null) // Can test for null reference
                c[i] = new BerylliumSphere();
        // Aggregate initialization:
        BerylliumSphere[] d = {
            new BerylliumSphere(),
            new BerylliumSphere(),
            new BerylliumSphere()
        };
        // Dynamic aggregate initialization:
         a = new BerylliumSphere[] {
            new BerylliumSphere(),
            new BerylliumSphere(),
        };
        // (Trailing comma is optional in both cases)
        print("a.length = " + a.length);
        print("b.length = " + b.length);
        print("c.length = " + c.length);
        print("d.length = " + d.length);
        a = d;
        print("a.length = " + a.length);

        // Arrays of primitives:
        int[] e; // Null reference
        int[] f = new int[5];
        // The primitives inside the array are
        // automatically initialized to zero:
        print("f: " + Arrays.toString(f));
        int[] g = new int[4];
        for(int i = 0; i < g.length; i++)
            g[i] = i*i;
        int[] h = { 11, 47, 93 };
        // Compile error: variable e not initialized:
        //!print("e.length = " + e.length);
        print("f.length = " + f.length);
        print("g.length = " + g.length);
        print("h.length = " + h.length);
        e = h;
        print("e.length = " + e.length);
        e = new int[] { 1, 2 };
        print("e.length = " + e.length);
    }
} /* Output:
      b: [null, null, null, null, null]
      a.length = 2
      b.length = 5
      c.length = 4
      d.length = 3
      a.length = 3
      f: [0, 0, 0, 0, 0]
      f.length = 5
      g.length = 4
      h.length = 3
      e.length = 3
      e.length = 2
   *///:~
```

[9] Der Bezeichner `a` ist eine nicht initialisierte lokale Variable und der Compiler hindert Sie daran, etwas damit zu tun, bevor die Variable initialisiert ist. Der Bezeichner `b` wird mit einem Array für Objekte vom Typ `BerylliumSphere` initialisiert, aber es werden keine `BerylliumSphere`-Objekte im Array gespeichert. Sie können trotzdem die Länge des Array abfragen, da `b` ein gültiges Objekt referenziert. Darin liegt ein geringfügiger Nachteil: Sie können nicht abfragen, wieviele Elemente sich tatsächlich im Array befinden, da das `length`-Feld nur angibt, wieviele Elemente gespeichert werden können, das heißt die Länge des Arrays und nicht die Anzahl der darin enthaltenen Elemente. Andererseits werden die Elemente eines Arrayobjektes bei der Objekterzeugung automatisch mit `null` initialisiert, so daß Sie den Inhalt eines Speicherplatzes einfach auf `null` prüfen können. Ein Array von Elementen primitiven Typs wird in analoger Weise bei numerischem Typ automatisch mit Null, bei `char` mit `\u0000` und bei `boolean` mit `false` initialisiert.

[10] Beim Bezeichner `c` wird ein Arrayobjekt erzeugt und anschließend jeder Speicherplatz mit einem `BerylliumSphere`-Objekt besetzt. Beim Bezeichner `d` sehen Sie die „aggregierte Initialisierungssyntax“, die in nur einer Anweisung sowohl ein Arrayobjekt erzeugt (mit implizitem `new`-Operator) als auch mit `BerylliumSphere`-Objekten initialisiert.

[11] Die nächste Arrayinitialisierung können Sie sich als „dynamische aggregierte Initialisierung“ vorstellen. Die aggregierte Initialisierung bei `d` muß an der Stelle der Deklaration der Referenzvariablen `d` vorgenommen werden, aber die zweite Syntax ermöglicht Ihnen, das Arrayobjekt an einer freiwählbaren Stelle zu erzeugen und zu initialisieren. Nehmen wir beispielsweise an, `hide()` sei eine Methode die ein Array von `BerylliumSphere`-Objekten erwartet. Dann können Sie die Methode einmal so aufrufen:

```
hide(d);
```

oder das Argument dynamisch erzeugen:

```
hide(new BerylliumSphere[] {new BerylliumSphere(), new BerylliumSphere()});
```

Diese Syntax gestattet in vielen Situationen eine komfortablere Ausdrucksweise. Die Zuweisung

```
a = d;
```

zeigt, wie Sie eine in einer Referenzvariablen gespeicherte Arrayreferenz einer anderen Referenzvariablen zuweisen können, analog zu jeder anderen Objektreferenz. Nun referenzieren `a` und `b` dasselbe Objekt im dynamischen Speicher.

[12] Der zweite Teil des Beispiels zeigt, daß Arrays von Elementen primitiven Typs wie Arrays von Objekten funktionieren, ausgenommen, daß die ersteren die primitiven Wert direkt speichern.

Übungsaufgabe 1: (2) Legen Sie eine Methode an, die ein Array von `BerylliumSphere`-Objekten als Argument erwartet. Rufen Sie die Methode auf und erzeugen Sie das Argument dynamisch. Zeigen Sie, daß die gewöhnliche aggregierte Initialisierung von Arrays in diesem Fall nicht funktioniert. Ermitteln Sie die Situationen, in denen die gewöhnliche aggregierte Initialisierung funktioniert und dynamische aggregierte Initialisierung redundant ist. ■

17.3 Zurückgeben eines Arrays

[13] Stellen Sie sich vor, Sie schreiben eine Methode und wollen nicht nur einen, sondern mehrere Werte zurückgeben. In Sprachen wie C und C++ ist das schwierig, da Sie nicht einfach ein Array zurückgeben können, sondern nur einen Zeiger auf ein Array. Dadurch schleppen Sie Probleme ein, da sich der Lebenszyklus eines Arrays schwer kontrollieren läßt, so daß Speicherlecks entstehen können.

[14] In Java können Sie einfach ein Array zurückgeben. Sie brauchen sich über Ihre Verantwortung für das Array keine Gedanken zu machen. Es ist solange vorhanden wie es benötigt wird und wird anschließend von der automatischen Speicherbereinigung aufgeräumt.

[15] Das folgende Beispiel zeigt die Rückgabe eines `String`:

```
//: arrays/IceCream.java
// Returning arrays from methods.
import java.util.*;

public class IceCream {
    private static Random rand = new Random(47);
    static final String[] FLAVORS = {
        ''Chocolate'', ''Strawberry'', ''Vanilla Fudge Swirl'',
        ''Mint Chip'', ''Mocha Almond Fudge'', ''Rum Raisin'',
        ''Praline Cream'', ''Mud Pie''
    };
    public static String[] flavorSet(int n) {
        if(n > FLAVORS.length)
            throw new IllegalArgumentException(''Set too big'');
        String[] results = new String[n];
        boolean[] picked = new boolean[FLAVORS.length];
        for(int i = 0; i < n; i++) {
            int t;
            do
                t = rand.nextInt(FLAVORS.length);
            while(picked[t]);
            results[i] = FLAVORS[t];
            picked[t] = true;
        }
        return results;
    }
    public static void main(String[] args) {
        for(int i = 0; i < 7; i++)
            System.out.println(Arrays.toString(flavorSet(3)));
    }
} /* Output:
      [Rum Raisin, Mint Chip, Mocha Almond Fudge]
      [Chocolate, Strawberry, Mocha Almond Fudge]
      [Strawberry, Mint Chip, Mocha Almond Fudge]
      [Rum Raisin, Vanilla Fudge Swirl, Mud Pie]
      [Vanilla Fudge Swirl, Chocolate, Mocha Almond Fudge]
      [Praline Cream, Strawberry, Mocha Almond Fudge]
      [Mocha Almond Fudge, Strawberry, Mint Chip]
   *///:~
```

Die Methode `flavorSet()` erzeugt ein `String`-Array namens `result`. Die Länge dieses Arrays ist durch den Parameter `n` bestimmt, der beim Aufrufen der Methode bewertet wird. Die Methode wählt per Zufallsmechanismus eine Eissorte aus dem Array `FLAVORS` und speichert sie in dem von `result` referenzierten Array, das auch der Rückgabewert von `flavorSet()` ist. Die Rückgabe eines Arrays geschieht auf die gleiche Weise wie die Rückgabe eines Objektes, nämlich als Referenz. Es ist unwichtig, ob das Array in der Methode `flavorSet()` oder an einer anderen Stelle erzeugt wurde. Die automatische Speicherbereinigung kümmert sich darum, das Array aufzuräumen, wenn es nicht mehr gebraucht wird und das Array bleibt solange bestehen, als es benötigt wird.

[16] Beachten Sie, daß `flavorSet()` bei der zufälligen Wahl der Eissorte sicherstellt, daß ein bereits gewählte Sorte nicht noch einmal gewählt wird. Die `do`-Schleife wählt solange eine Sorte nach der

anderen, bis sie eine Sorte findet, die noch nicht in den von `picked` referenzierten Array markiert ist. (Selbstverständlich hätte auch ein Zeichenkettenvergleich die Antwort liefern können, ob die zufällig gewählte Eissorte bereits in dem von `result` referenzierten Array gespeichert ist.) Nach erfolgreicher Wahl wird die Eissorte gespeichert und der Auswahlvorgang beginnt erneut (`i` wird inkrementiert).

[17] Sie sehen an der Ausgabe, daß die `flavorSet()`-Methode bei jedem Aufruf die Eissorten in zufälliger Reihenfolge wählt.

Übungsaufgabe 2: (2) Schreiben Sie eine Methode, die ein `int`-Argument erwartet und ein Array entsprechender Länge zurückgibt, das `BerylliumSphere`-Objekte enthält. ■

17.4 Mehrdimensionale Arrays

[18] Sie können mühelos mehrdimensionale Arrays erzeugen. Das folgende Beispiel erzeugt ein mehrdimensionales Array von Elementen primitiven Typs. Jeder Vektor des Arrays ist durch ein Paar geschweifter Klammern gekennzeichnet:

```
//: arrays/MultidimensionalPrimitiveArray.java
// Creating multidimensional arrays.
import java.util.*;

public class MultidimensionalPrimitiveArray {
    public static void main(String[] args) {
        int[][] a = {
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
      [[1, 2, 3], [4, 5, 6]]
   *///:~
```

Jedes verschachtelte Paar geschweifter Klammern bringt Sie zur nächsten Ebene des Arrays.

[19] Das Beispiel bedient sich der `Arrays`-Methode `deepToString()` (seit Version 5 der Java Standard Edition (SE 5) vorhanden), um das zweidimensionale Array in eine `String`-Darstellung umzuwandeln, wie Sie an der Ausgabe sehen.

[20] Sie können ein mehrdimensionales Arrays auch per `new` allokieren. Das folgende Beispiel erzeugt ein dreidimensionales Array:

```
//: arrays/ThreeDWithNew.java
import java.util.*;

public class ThreeDWithNew {
    public static void main(String[] args) {
        // 3-D array with fixed length:
        int[][][] a = new int[2][2][4];
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
      [[[0, 0, 0, 0], [0, 0, 0, 0]], [[0, 0, 0, 0], [0, 0, 0, 0]]]
   *///:~
```

Speicherplätze für Werte primitiven Typs werden automatisch initialisiert, wenn Sie nicht selbst Anfangswerte zuweisen. Speicherplätze für Objekte werden mit `null` initialisiert.

[21] Jeder Vektor eines Arrays kann eine individuelle Länge haben, so daß das Array keine rechteckige Form hat (*ragged array*):

```
//: arrays/RaggedArray.java
import java.util.*;

public class RaggedArray {
    public static void main(String[] args) {
        Random rand = new Random(47);
        // 3-D array with varied-length vectors:
        int[][][] a = new int[rand.nextInt(7)][][];
        for(int i = 0; i < a.length; i++) {
            a[i] = new int[rand.nextInt(5)][];
            for(int j = 0; j < a[i].length; j++)
                a[i][j] = new int[rand.nextInt(5)];
        }
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
      [[], [[0], [0], [0, 0, 0, 0]], [[], [0, 0], [0, 0]], [[0, 0, 0], [0],
      [0, 0, 0, 0]], [[0, 0, 0], [0, 0, 0], [0], []], [[0], [], [0]]]
   *///:~
```

Der erste `new`-Ausdruck erzeugt ein Array dessen Länge in der ersten Dimension zufällig gewählt wird, die beiden übrigen Dimensionen bleiben unbestimmt. Der zweite `new`-Ausdruck erzeugt wiederum ein Array dessen Länge in der ersten Dimension (zweite Dimension von `a`) zufällig gewählt wird, die Länge der zweiten Dimension (dritte Dimension von `a`) bleibt unbestimmt und entscheidet sich erst beim dritten `new`-Ausdruck.

[22] Arrays von Referenzen auf Objekte werden genauso behandelt. Das nächste Beispiel versammelt viele `new`-Ausdrücke zwischen Paaren geschweifer Klammern:

```
//: arrays/MultidimensionalObjectArrays.java
import java.util.*;

public class MultidimensionalObjectArrays {
    public static void main(String[] args) {
        BerylliumSphere[][] spheres = {
            { new BerylliumSphere(), new BerylliumSphere() },
            { new BerylliumSphere(), new BerylliumSphere(),
              new BerylliumSphere(), new BerylliumSphere() },
            { new BerylliumSphere(), new BerylliumSphere(),
              new BerylliumSphere(), new BerylliumSphere(),
              new BerylliumSphere(), new BerylliumSphere(),
              new BerylliumSphere(), new BerylliumSphere() },
        };
        System.out.println(Arrays.deepToString(spheres));
    }
} /* Output:
      [[Sphere 0, Sphere 1], [Sphere 2, Sphere 3, Sphere 4, Sphere 5],
      [Sphere 6, Sphere 7, Sphere 8, Sphere 9, Sphere 10, Sphere 11,
      Sphere 12, Sphere 13]]
   *///:~
```

Das von `spheres` referenzierte Array ist ebenfalls nicht rechteckig. Jeder Vektor von Objektreferenzen hat eine individuelle Länge.

[23] Autoboxing funktioniert auch bei Arrayinitialisierungsausdrücken:

```
//: arrays/AutoboxingArrays.java
import java.util.*;

public class AutoboxingArrays {
    public static void main(String[] args) {
        Integer[][] a = { // Autoboxing:
            { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 },
            { 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 },
            { 51, 52, 53, 54, 55, 56, 57, 58, 59, 60 },
            { 71, 72, 73, 74, 75, 76, 77, 78, 79, 80 },
        };
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
      [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [21, 22, 23, 24, 25, 26, 27, 28, 29, 30],
      [51, 52, 53, 54, 55, 56, 57, 58, 59, 60], [71, 72, 73, 74, 75, 76, 77, 78,
      79, 80]]
   *///:~
```

Das nächste Beispiel baut ein Array von Objektreferenzen elementweise auf:

```
//: arrays/AssemblingMultidimensionalArrays.java
// Creating multidimensional arrays.
import java.util.*;

public class AssemblingMultidimensionalArrays {
    public static void main(String[] args) {
        Integer[][] a;
        a = new Integer[3][];
        for(int i = 0; i < a.length; i++) {
            a[i] = new Integer[3];
            for(int j = 0; j < a[i].length; j++)
                a[i][j] = i * j; // Autoboxing
        }
        System.out.println(Arrays.deepToString(a));
    }
} /* Output:
      [[0, 0, 0], [0, 1, 2], [0, 2, 4]]
   *///:~
```

Das Produkt `i*j` dient nur dazu, dem `Integer`-Speicherplatz einen interessanten Wert zuzuweisen.

[24] Die `Arrays`-Methode `deepToString()` funktioniert sowohl bei Arrays von Elementen primitiven Typs, als auch bei Arrays von Objektreferenzen:

```
//: arrays/MultiDimWrapperArray.java
// Multidimensional arrays of ''wrapper'' objects.
import java.util.*;

public class MultiDimWrapperArray {
    public static void main(String[] args) {
        Integer[][] a1 = { // Autoboxing
            { 1, 2, 3, },
            { 4, 5, 6, },
        };
        Double[][][] a2 = { // Autoboxing
            { { 1.1, 2.2 }, { 3.3, 4.4 } },
            { { 5.5, 6.6 }, { 7.7, 8.8 } },
            { { 9.9, 1.2 }, { 2.3, 3.4 } },
```

```
        };
        String[][] a3 = {
            { ''The'', ''Quick'', ''Sly'', ''Fox'' },
            { ''Jumped'', ''Over'' },
            { ''The'', ''Lazy'', ''Brown'', ''Dog'', ''and'', ''friend'' },
        };
        System.out.println(''a1: '' + Arrays.deepToString(a1));
        System.out.println(''a2: '' + Arrays.deepToString(a2));
        System.out.println(''a3: '' + Arrays.deepToString(a3));
    }
} /* Output:
      a1: [[1, 2, 3], [4, 5, 6]]
      a2: [[[1.1, 2.2], [3.3, 4.4]], [[5.5, 6.6], [7.7, 8.8]],
        [[9.9, 1.2], [2.3, 3.4]]]
      a3: [[The, Quick, Sly, Fox], [Jumped, Over],
        [The, Lazy, Brown, Dog, and, friend]]
   *///:~
```

Auch bei den `Integer`- und `Double`-Arrays erzeugt der Autoboxingmechanismus der SE 5 die Wrapperobjekte für Sie.

Übungsaufgabe 3: (4) Schreiben Sie eine Methode, die ein zweidimensionales Array von `double`-Elementen erzeugt. Die Längen des Arrays sind durch die Argumente der Methode bestimmt, die Initialisierungswerte werden aus einem durch Anfangs- und Endwert definierten Bereich gewählt, die ebenfalls als Argumente übergeben werden. Schreiben Sie eine zweite Methode, welche das von der ersten Methode erzeugte Array ausgibt. Testen Sie die Methode in der `main()`-Methode, indem Sie Arrays in verschiedenen Längen erzeugen und ausgeben. ■

Übungsaufgabe 4: (2) Wiederholen Sie Übungsaufgabe 3 mit einem dreidimensionalen Array. ■

Übungsaufgabe 5: (1) Zeigen Sie, daß die Speicherplätze eines mehrdimensionalen Arrays von Elementen nicht-primitiven Typs automatisch mit `null` initialisiert werden. ■

Übungsaufgabe 6: (1) Schreiben Sie eine Methode, die zwei `int`-Argumente erwartet, welche die beiden Längen eines zweidimensionalen Arrays sind. Die Methode soll ein entsprechendes zweidimensionales Array erzeugen und mit `BerylliumSphere`-Objekten initialisieren. ■

Übungsaufgabe 7: (1) Wiederholen Sie Übungsaufgabe 6 mit einem dreidimensionalen Array. ■

17.5 Arrays aus generische Typen

[25] In der Regel lassen sich Arrays und generische Typen nicht gut miteinander kombinieren. Sie können kein Array aus einem generischen Typ erzeugen:

```
Peel<Banana>[] peels = new Peel<Banana>[10]; // Illegal
```

Die Information welcher Parametertyp gewählt wurdem geht durch Typsauslöschung verloren. Arrays müssen aber den exakten Typ ihrer Elemente kennen, um Typsicherheit erzwingen zu können.

[26] Der Arraytyp selbst ist dagegen parametrisierbar:

```
//: arrays/ParameterizedArrayType.java
class ClassParameter<T> {
    public T[] f(T[] arg) { return arg; }
}

class MethodParameter {
```

```
    public static <T> T[] f(T[] arg) { return arg; }
}

public class ParameterizedArrayType {
    public static void main(String[] args) {
        Integer[] ints = { 1, 2, 3, 4, 5 };
        Double[] doubles = { 1.1, 2.2, 3.3, 4.4, 5.5 };
        Integer[] ints2 = new ClassParameter<Integer>().f(ints);
        Double[] doubles2 = new ClassParameter<Double>().f(doubles);
        ints2 = MethodParameter.f(ints);
        doubles2 = MethodParameter.f(doubles);
    }
} ///:~
```

Beachten Sie den Komfort einer parametrisierten Methode gegenüber einer parametrisierten Klasse: Sie brauchen nicht für jeden benötigten Parametertyp ein Objekt zu erzeugen und können die Methode als statisch deklarieren. Natürlich können Sie nicht immer eine parametrisierte Methode anstelle einer parametrisierten Klasse wählen, aber die Methode kann die bessere Wahl sein.

[27] Es ist nicht ganz richtig, daß Sie kein Array aus einem generischen Typ erzeugen können. Der Compiler gestattet Ihnen nicht, ein *Arrayobjekt* aus einem generischen Typ zu erzeugen, wohl aber eine Referenzvariable für ein solches Array anzulegen:

```
List<String>[] ls;
```

Diese Deklaration durchläuft den Compiler ohne Beschwerde. Sie können zwar kein Arrayobjekt aus einem generischen Typ erzeugen, wohl aber aus dem nicht-generischen Typ und anschließend umwandeln:

```
//: arrays/ArrayOfGenerics.java
// It is possible to create arrays of generics.
import java.util.*;

public class ArrayOfGenerics {
    @SuppressWarnings("unchecked")
    public static void main(String[] args) {
        List<String>[] ls;
        List[] la = new List[10];
        ls = (List<String>[]) la; // "Unchecked" warning
        ls[0] = new ArrayList<String>();
        // Compile-time checking produces an error:
        //! ls[1] = new ArrayList<Integer>();

        // The problem: List<String> is a subtype of Object
        Object[] objects = ls; // So assignment is OK
        // Compiles and runs without complaint:
        objects[1] = new ArrayList<Integer>();

        // However, if your needs are straightforward it is
        // possible to create an array of generics, albeit
        // with an "unchecked" warning:
        List<BerylliumSphere>[] spheres = (List<BerylliumSphere>[]) new List[10];
        for(int i = 0; i < spheres.length; i++)
            spheres[i] = new ArrayList<BerylliumSphere>();
    }
} ///:~
```

Die Deklaration der Referenzvariablen vom Typ `List<String>[]` bewirkt gewisse Prüfungen zur Übersetzungszeit, wie die „unchecked"-Warnung dokumentiert. Das Problem besteht darin, daß Arrays kovariant sind, das heißt eine Referenzvariable vom Typ `List<String>[]` ist auch vom Typ

`Object[]`, so daß Sie ein `ArrayList<Integer>`-Objekt als Arrayelement zuweisen können, ohne daß zur Übersetzungs- oder Laufzeit ein Fehler gemeldet wird.

[28] Wenn Sie auf die aufwärts gerichtete Tyumwandlung nach `Object[]` verzichten und nur relativ unkomplizierte Anforderungen stellen, können Sie ein Array aus einem generischen Typ erzeugen, das eine einfache Typprüfung zur Übersetzungszeit ausführt.

[29] Im allgemeinen sind generische Typen an den Ein- und Austrittspunkten einer Klasse oder Methode wirksam. Im Inneren macht die Typauslöschung generische Typen für gewöhnlich unbrauchbar. Sie können beispielsweise kein Arrayobjekt aus dem Typparameter eines generischen Typs erzeugen:

```
//: arrays/ArrayOfGenericType.java
// Arrays of generic types won't compile.

public class ArrayOfGenericType<T> {
    T[] array; // OK
    @SuppressWarnings("unchecked")
    public ArrayOfGenericType(int size) {
        //! array = new T[size]; // Illegal
        array = (T[]) new Object[size]; // "unchecked" Warning
    }
    // Illegal:
    //! public <U> U[] makeArray() { return new U[10]; }
} ///:~
```

Auch hier steht die Typauslöschung im Weg: Das Beispiel versucht Arrayobjekte von Typen zu erzeugen, die ausgelöscht wurden und somit unbekannte Typen sind. Beachten Sie, daß Sie ein `Object`-Arrayobjekt erzeugen und umwandeln können, wobei allerdings, wenn Sie die Annotation `@SuppressWarnings` nicht angeben, zur Übersetzungszeit eine „unchecked"-Warnung ausgegeben wird, da das Array eigentlich keine Elemente vom Typ `T` enthält und die übergebenen Elemente auch nicht auf diesem Typ prüft. Bei einem `String`-Array erzwingt Java sowohl zur Übersetzungs- als auch zur Laufzeit, daß nur `String`-Objekte in das Array eingesetzt werden. Ein `Object`-Array kann dagegen alles außer Elementen primitiven Typs enthalten.

Übungsaufgabe 8: (1) Zeigen Sie die Behauptungen im letzten Absatz. ■

Übungsaufgabe 9: (3) Schreiben Sie die für das `Peel<Banana>`-Beispiel benötigten Klassen und zeigen Sie, daß der Compiler die Erzeugung des Arrayobjektes verweigert. Lösen Sie das Problem, indem Sie ein `ArrayList`-Objekt verwenden. ■

Übungsaufgabe 10: (2) Ändern Sie das Beispiel *ArrayOfGenericType.java* so, daß es Container statt Arrays verwendet und zeigen Sie, daß sich die Warnungen zur Übersetzungszeit elimieren lassen. ■

17.6 Erzeugen von Testdaten

[30] Beim Experimentieren mit Arrays und Programmen im allgemeinen, ist es nützlich, auf einfachem Wege Arrays mit Testdaten erzeugen zu können. Die Beispiele in diesem Kapitel füllen ein Array mit Werten primitiven Typs beziehungsweise Objektreferenzen.

17.6.1 Die Arrays-Methode fill()

[31] Die Klasse `Arrays` aus der Standardbibliothek von Java hat eine einfache statische `fill()`-Methode, die lediglich einen einzigen Wert beziehungsweise eine einzige Referenz in alle Speicherplätze kopiert:

```
//: arrays/FillingArrays.java
// Using Arrays.fill()
import java.util.*;
import static net.mindview.util.Print.*;

public class FillingArrays {
    public static void main(String[] args) {
        int size = 6;
        boolean[] a1 = new boolean[size];
        byte[] a2 = new byte[size];
        char[] a3 = new char[size];
        short[] a4 = new short[size];
        int[] a5 = new int[size];
        long[] a6 = new long[size];
        float[] a7 = new float[size];
        double[] a8 = new double[size];
        String[] a9 = new String[size];
        Arrays.fill(a1, true);
        print("a1 = " + Arrays.toString(a1));
        Arrays.fill(a2, (byte) 11);
        print("a2 = " + Arrays.toString(a2));
        Arrays.fill(a3, 'x');
        print("a3 = " + Arrays.toString(a3));
        Arrays.fill(a4, (short) 17);
        print("a4 = " + Arrays.toString(a4));
        Arrays.fill(a5, 19);
        print("a5 = " + Arrays.toString(a5));
        Arrays.fill(a6, 23);
        print("a6 = " + Arrays.toString(a6));
        Arrays.fill(a7, 29);
        print("a7 = " + Arrays.toString(a7));
        Arrays.fill(a8, 47);
        print("a8 = " + Arrays.toString(a8));
        Arrays.fill(a9, "Hello");
        print("a9 = " + Arrays.toString(a9));
        // Manipulating ranges:
        Arrays.fill(a9, 3, 5, "World");
        print("a9 = " + Arrays.toString(a9));
    }
} /* Output:
      a1 = [true, true, true, true, true, true]
      a2 = [11, 11, 11, 11, 11, 11]
      a3 = [x, x, x, x, x, x]
      a4 = [17, 17, 17, 17, 17, 17]
      a5 = [19, 19, 19, 19, 19, 19]
      a6 = [23, 23, 23, 23, 23, 23]
      a7 = [29.0, 29.0, 29.0, 29.0, 29.0, 29.0]
      a8 = [47.0, 47.0, 47.0, 47.0, 47.0, 47.0]
      a9 = [Hello, Hello, Hello, Hello, Hello, Hello]
      a9 = [Hello, Hello, Hello, World, World, Hello]
   *///:~
```

Sie können entweder das gesamte Array füllen oder nur einen Teilbereich, wie die beiden letzten `fill()`-Aufrufe zeigen. Da Sie die `fill()`-Methode nur mit einem einzigen Wert aufrufen können, ist das Ergebnis nicht besonders nützlich.

17.6.2 Datengeneratoren

[32–33] Das in Kapitel 16 eingeführte Interface *`Generator`*/Generatorkonzept gestattet flexibles Erzeugen von Arrays mit interessanteren Daten. Durch die Wahl eines entsprechenden Generators können Sie Daten jedes Typs erzeugen (ein Beispiel für das *Strategy*-Entwurfsmuster; jede Implementierung des *`Generator`*-Interfaces stellt eine andere Strategie dar[1]).

[34] Dieser Abschnitt präsentiert einige Generatorklassen. Wie Sie bereits im vorigen Kapitel gesehen haben, können Sie mühelos eigene Generatorklassen hinzufügen.

[35] Das erste Beispiel präsentiert einen Satz von zählenden Generatoren für die Wrapperklassen der primitiven Typen und die Klasse `String`. Die Generatorklasse sind als innere Klassen der Klasse `CountingGenerator` angelegt, um die Namen der Klassen der generierten Objekte verwenden zu können. Die Generatorklasse für `Integer`-Objekte wird beispielsweise per `new CountingGenerator.Integer()` erzeugt:

```
//: net/mindview/util/CountingGenerator.java
// Simple generator implementations.
package net.mindview.util;

public class CountingGenerator {
    public static class
        Boolean implements Generator<java.lang.Boolean> {
        private boolean value = false;
        public java.lang.Boolean next() {
            value = !value; // Just flips back and forth
            return value;
        }
    }
    public static class
        Byte implements Generator<java.lang.Byte> {
        private byte value = 0;
        public java.lang.Byte next() { return value++; }
    }
    static char[] chars = (''abcdefghijklmnopqrstuvwxyz'' +
                           ''ABCDEFGHIJKLMNOPQRSTUVWXYZ'').toCharArray();
    public static class
        Character implements Generator<java.lang.Character> {
        int index = -1;
        public java.lang.Character next() {
            index = (index + 1) % chars.length;
            return chars[index];
        }
    }
    public static class
        String implements Generator<java.lang.String> {
        private int length = 7;
        Generator<java.lang.Character> cg = new Character();
```

[1] Die Abgrenzung zwischen den Entwurfsmustern *Strategy* und *Command* ist hier unscharf. Sie könnten auch argumentieren, daß eine Generatorklasse das *Command*-Entwurfsmuster repräsentiert. Aus meiner Perspektive besteht die Aufgabe aber darin, ein Array zu füllen, wobei eine Generatorklasse einen Teil dieser Aufgabe erfüllt, also einer Strategie mehr ähnelt als einem Kommando.

```
        public String() {}
        public String(int length) { this.length = length; }
        public java.lang.String next() {
            char[] buf = new char[length];
            for(int i = 0; i < length; i++)
                buf[i] = cg.next();
            return new java.lang.String(buf);
        }
    }
    public static class
        Short implements Generator<java.lang.Short> {
        private short value = 0;
        public java.lang.Short next() { return value++; }
    }
    public static class
        Integer implements Generator<java.lang.Integer> {
        private int value = 0;
        public java.lang.Integer next() { return value++; }
    }
    public static class
        Long implements Generator<java.lang.Long> {
        private long value = 0;
        public java.lang.Long next() { return value++; }
    }
    public static class
        Float implements Generator<java.lang.Float> {
        private float value = 0;
        public java.lang.Float next() {
            float result = value;
            value += 1.0;
            return result;
        }
    }
    public static class
        Double implements Generator<java.lang.Double> {
        private double value = 0.0;
        public java.lang.Double next() {
            double result = value;
            value += 1.0;
            return result;
        }
    }
} ///:~
```

[36] Jede innere Generatorklasse „zählt“ in gewisser Weise. Bei `CountingGenerator.Character` sind es die Groß- und Kleinbuchstaben. Die Klasse `CountingGenerator.String` verwendet `CountingGenerator.Character`, um ein `char`-Array mit Buchstaben zu füllen, das anschließend in ein `String`-Objekt umgewandelt und zurückgegeben wird. Die Länge des `char`-Arrays wird durch das Argument des Konstruktors bestimmt. Beachten Sie, daß `CountingGenerator.String` eine Referenzvariable vom Typ *`Generator<java.lang.Character>`* anstelle von `CountingGenerator.Character` verwendet. Im Beispiel *RandomGenerator.java* auf Seite 598 wird das `CountingGenerator.Character`-Objekt durch ein `RandomGenerator.String`-Objekt ersetzt.

[37] Das folgende Beispiel wendet den Reflexionsmechanismus auf geschachtelte Generatorklassen an, läßt sich also zum Testen von Klassen der obigen Struktur verwenden:

```
//: arrays/GeneratorsTest.java
```

```
import net.mindview.util.*;

public class GeneratorsTest {
    public static int size = 10;
    public static void test(Class<?> surroundingClass) {
        for(Class<?> type : surroundingClass.getClasses()) {
            System.out.print(type.getSimpleName() + ": ");
            try {
                Generator<?> g = (Generator<?>) type.newInstance();
                for(int i = 0; i < size; i++)
                    System.out.printf(g.next() + " ");
                System.out.println();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    public static void main(String[] args) {
        test(CountingGenerator.class);
    }
} /* Output:
      Double: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
      Float: 0.0 1.0 2.0 3.0 4.0 5.0 6.0 7.0 8.0 9.0
      Long: 0 1 2 3 4 5 6 7 8 9
      Integer: 0 1 2 3 4 5 6 7 8 9
      Short: 0 1 2 3 4 5 6 7 8 9
      String: abcdefg hijklmn opqrstu vwxyzAB CDEFGHI JKLMNOP QRSTUVW XYZabcd
        efghijk lmnopqr
      Character: a b c d e f g h i j
      Byte: 0 1 2 3 4 5 6 7 8 9
      Boolean: true false true false true false true false true false
   *///:~
```

Die Testklasse `GeneratorsTest` setzt voraus, daß die getestete Klasse geschachtelte Generatorklassen enthält, die jeweils über einen Standardkonstruktor (einen argumentlosen Konstruktor) verfügen. Die Methode `getClasses()` gibt ein Array von Klassenobjekten aller geschachtelten Klassen zurück. Die `test()`-Methode erzeugt ein Objekt der jeweiligen Generatorklasse und gibt das Ergebnis von zehn Aufrufen der `next()`-Methode aus.

[38] Das nächste Beispiel zeigt eine Anzahl innerer Generatorklassen, die einen Zufallszahlengenerator verwenden. Da der `Random`-Konstruktor mit einem konstanten Wert aufgerufen wird, ist die Ausgabe der Generatoren in diesem Beispiel reproduzierbar:

```
//: net/mindview/util/RandomGenerator.java
// Generators that produce random values.
package net.mindview.util;
import java.util.*;

public class RandomGenerator {
    private static Random r = new Random(47);
    public static class
        Boolean implements Generator<java.lang.Boolean> {
        public java.lang.Boolean next() {
            return r.nextBoolean();
        }
    }
    public static class
        Byte implements Generator<java.lang.Byte> {
```

```
        public java.lang.Byte next() {
            return (byte)r.nextInt();
        }
    }
    public static class
        Character implements Generator<java.lang.Character> {
        public java.lang.Character next() {
            return
                CountingGenerator.chars[r.nextInt(CountingGenerator.chars.length)];
        }
    }
    public static class
        String extends CountingGenerator.String {
        // Plug in the random Character generator:
        { cg = new Character(); } // Instance initializer
        public String() {}
        public String(int length) { super(length); }
    }
    public static class
        Short implements Generator<java.lang.Short> {
        public java.lang.Short next() {
            return (short) r.nextInt();
        }
    }
    public static class
        Integer implements Generator<java.lang.Integer> {
        private int mod = 10000;
        public Integer() {}
        public Integer(int modulo) { mod = modulo; }
        public java.lang.Integer next() {
            return r.nextInt(mod);
        }
    }
    public static class
        Long implements Generator<java.lang.Long> {
        private int mod = 10000;
        public Long() {}
        public Long(int modulo) { mod = modulo; }
        public java.lang.Long next() {
            return new java.lang.Long(r.nextInt(mod));
        }
    }
    public static class
        Float implements Generator<java.lang.Float> {
        public java.lang.Float next() {
            // Trim all but the first two decimal places:
            int trimmed = Math.round(r.nextFloat() * 100);
            return ((float) trimmed) / 100;
        }
    }
    public static class
        Double implements Generator<java.lang.Double> {
        public java.lang.Double next() {
            long trimmed = Math.round(r.nextDouble() * 100);
            return ((double) trimmed) / 100;
        }
    }
```

```
} ///:~
```

Die Klasse `RandomGenerator.String` ist von `CountingGenerator.String` abgeleitet, wobei das `Character`-Generatorobjekt einfach ausgewechselt wird.

[39] Damit die erzeugten Zahlen nicht zu groß werden, beschränkt `RandomGenerator.Integer` den Zufallsgenerator in der `next()`-Methode auf 10000. Sie können per Konstruktor auch eine andere Schranke einstellen. Bei `RandomGenerator.Long` gilt dieselbe Beschränkung. Die Klassen `RandomGenerator.Float` und `RandomGenerator.Double` brechen den erzeugten Wert nach der zweiten Nachkommastelle ab.

[40] Wie verwenden die Klasse `GeneratorsTest` nochmals, um `RandomGenerator` zu testen:

```
//: arrays/RandomGeneratorsTest.java
import net.mindview.util.*;

public class RandomGeneratorsTest {
    public static void main(String[] args) {
        GeneratorsTest.test(RandomGenerator.class);
    }
} /* Output:
      Double: 0.73 0.53 0.16 0.19 0.52 0.27 0.26 0.05 0.8 0.76
      Float: 0.53 0.16 0.53 0.4 0.49 0.25 0.8 0.11 0.02 0.8
      Long: 7674 8804 8950 7826 4322 896 8033 2984 2344 5810
      Integer: 8303 3141 7138 6012 9966 8689 7185 6992 5746 3976
      Short: 3358 20592 284 26791 12834 -8092 13656 29324 -1423 5327
      String: bkInaMe sbtWHkj UrUkZPg wsqPzDy CyRFJQA HxxHvHq XumcXZJ
        oogoYWM NvqeuTp nXsgqia
      Character: x x E A J J m z M s
      Byte: -60 -17 55 -14 -5 115 39 -37 79 115
      Boolean: false true false false true true true true true true
   *///:~
```

Sie können die Anzahl der erzeugten Werte über das öffentliche Feld `GeneratorsTest.size` einstellen.

17.6.3 Arraygeneratoren

[41] Wir brauchen zwei Hilfsklassen, um per Generator ein Arrayobjekt füllen zu können. Die eine verwendet einen Generator, um ein Array eines von `Object` abgeleiteten Typs zu erzeugen. Die andere erwartet ein Array eines primitiven Wrappertyps und erzeugt ein Array von primitiven Werten.

[42] Die erste Hilfsklasse bietet zwei Optionen in Form einer überladenen statischen Methode namens `array()`. Die erste Version erwartet ein existierendes Array und füllt es mit Hilfe eines Generators. Die zweite Version erwartet ein Klassenobjekt, einen Generator und die gewünschte Anzahl von Elementen und erzeugt ein neues Array, welches per Generator gefüllt wird. Beachten Sie, daß die folgende Klasse `Generated` nur Arrays eines von `Object` abgeleiteten Typs erzeugt, nicht aber Arrays für primitive Werte:

```
//: net/mindview/util/Generated.java
package net.mindview.util;
import java.util.*;

public class Generated {
    // Fill an existing array:
    public static <T> T[] array(T[] a, Generator<T> gen) {
```

```
        return new CollectionData<T>(gen, a.length).toArray(a);
    }
    // Create a new array:
    @SuppressWarnings("unchecked")
    public static <T> T[] array(Class<T> type, Generator<T> gen, int size) {
        T[] a = (T[]) java.lang.reflect.Array.newInstance(type, size);
        return new CollectionData<T>(gen, size).toArray(a);
    }
} ///:~
```

Die Klasse `CollectionData` wird in Kapitel 18 definiert. Sie erzeugt ein Objekt vom Typ *`Collection`*, das mit Elementen gefüllt ist, die der Generator `gen` erzeugt. Das zweite Argument des Konstruktors gibt die Anzahl der Elemente an. Jeder Untertyp von *`Collection`* besitzt eine `toArray()`-Methode, die hier verwendet wird, um das als Argument übergebene Array mit Elementen zu füllen.

[43] Die zweite Version der `array()`-Methode verwendet den Reflexionsmechanismus, um dynamisch ein neues Array des entsprechenden Typs und der angeforderten Länge zu erzeugen.

[44] Wir können die Klasse `Generated` mit Hilfe einer der `CountingGenerator`-Klassen aus Unterabschnitt 17.6.2 testen:

```
//: arrays/TestGenerated.java
import java.util.*;
import net.mindview.util.*;

public class TestGenerated {
    public static void main(String[] args) {
        Integer[] a = { 9, 8, 7, 6 };
        System.out.println(Arrays.toString(a));
        a = Generated.array(a,new CountingGenerator.Integer());
        System.out.println(Arrays.toString(a));
        Integer[] b =
            Generated.array(Integer.class, new CountingGenerator.Integer(), 15);
        System.out.println(Arrays.toString(b));
    }
} /* Output:
      [9, 8, 7, 6]
      [0, 1, 2, 3]
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
   *///:~
```

Das von `a` referenzierte Array ist bereits initialisiert, wird aber von der `Generated`-Methode `array()` überschrieben (das Array bleibt dabei an Ort und Stelle). Die Initialisierung von `b` zeigt, daß die `array()`-Methode auch ein neues Array erzeugen und füllen kann.

[45] Generische Typen arbeiten nicht mit primitiven Parametertypen. Wir wollen aber die Generatoren verwenden, um Arrays mit Werten primitiven Typs zu füllen. Wir lösen das Problem durch eine Hilfklasse, die ein Array von Wrapperobjekten entgegennimmt und in ein Array des entsprechenden primitiven Typs umwandelt. Ohne diese Hilfsklasse müßten wir für jeden primitiven Typ eine eigene Generatorklasse schreiben:

```
//: net/mindview/util/ConvertTo.java
package net.mindview.util;

public class ConvertTo {
   public static boolean[] primitive(Boolean[] in) {
       boolean[] result = new boolean[in.length];
       for(int i = 0; i < in.length; i++)
           result[i] = in[i]; // Autounboxing
```

```
        return result;
    }
    public static char[] primitive(Character[] in) {
        char[] result = new char[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static byte[] primitive(Byte[] in) {
        byte[] result = new byte[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static short[] primitive(Short[] in) {
        short[] result = new short[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static int[] primitive(Integer[] in) {
        int[] result = new int[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static long[] primitive(Long[] in) {
        long[] result = new long[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static float[] primitive(Float[] in) {
        float[] result = new float[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
    public static double[] primitive(Double[] in) {
        double[] result = new double[in.length];
        for(int i = 0; i < in.length; i++)
            result[i] = in[i];
        return result;
    }
} ///:~
```

[46] Die `primitive()`-Methoden erzeugen jeweils ein Array passender Länge des entsprechenden primitiven Typs und kopieren anschließend die Elemente des von `in` referenzierten Wrapperarrays in das erstere Array. Beachten das Autounboxing in den Ausdrücken:

```
result[i] = in[i];
```

Das folgende Beispiel verwendet die Hilfsklasse `ConvertTo` mit beiden Versionen der `Generated`-Methode `array()`:

```
//: arrays/PrimitiveConversionDemonstration.java
import java.util.*;
import net.mindview.util.*;
```

```
public class PrimitiveConversionDemonstration {
    public static void main(String[] args) {
        Integer[] a =
            Generated.array(Integer.class, new CountingGenerator.Integer(), 15);
        int[] b = ConvertTo.primitive(a);
        System.out.println(Arrays.toString(b));
        boolean[] c = ConvertTo.primitive(
            Generated.array(Boolean.class, new CountingGenerator.Boolean(), 7));
        System.out.println(Arrays.toString(c));
    }
} /* Output:
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
      [true, false, true, false, true, false, true]
   *///:~
```

Abschließend testet das folgende Programm die Umwandlungsklassen aus diesem Unterabschnitt mit den `RandomGenerator`-Klassen:

```
//: arrays/TestArrayGeneration.java
// Test the tools that use generators to fill arrays.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class TestArrayGeneration {
    public static void main(String[] args) {
        int size = 6;
        boolean[] a1 = ConvertTo.primitive(
            Generated.array(Boolean.class, new RandomGenerator.Boolean(), size));
        print("a1 = " + Arrays.toString(a1));
        byte[] a2 = ConvertTo.primitive(
            Generated.array(Byte.class, new RandomGenerator.Byte(), size));
        print("a2 = " + Arrays.toString(a2));
        char[] a3 = ConvertTo.primitive(
            Generated.array(Character.class, new RandomGenerator.Character(), size));
        print("a3 = " + Arrays.toString(a3));
        short[] a4 = ConvertTo.primitive(
            Generated.array(Short.class, new RandomGenerator.Short(), size));
        print("a4 = " + Arrays.toString(a4));
        int[] a5 = ConvertTo.primitive(
            Generated.array(Integer.class, new RandomGenerator.Integer(), size));
        print("a5 = " + Arrays.toString(a5));
        long[] a6 = ConvertTo.primitive(
            Generated.array(Long.class, new RandomGenerator.Long(), size));
        print("a6 = " + Arrays.toString(a6));
        float[] a7 = ConvertTo.primitive(
            Generated.array(Float.class, new RandomGenerator.Float(), size));
        print("a7 = " + Arrays.toString(a7));
        double[] a8 = ConvertTo.primitive(
            Generated.array(Double.class, new RandomGenerator.Double(), size));
        print("a8 = " + Arrays.toString(a8));
    }
} /* Output:
      a1 = [true, false, true, false, false, true]
      a2 = [104, -79, -76, 126, 33, -64]
      a3 = [Z, n, T, c, Q, r]
      a4 = [-13408, 22612, 15401, 15161, -28466, -12603]
      a5 = [7704, 7383, 7706, 575, 8410, 6342]
```

```
    a6 = [7674, 8804, 8950, 7826, 4322, 896]
    a7 = [0.01, 0.2, 0.4, 0.79, 0.27, 0.45]
    a8 = [0.16, 0.87, 0.7, 0.66, 0.87, 0.59]
*///:~
```

Dieser Test überprüft außerdem, daß die Umwandlungen in den `primitive()`-Methoden der Klasse `ConvertTo` korrekt funktionieren.

Übungsaufgabe 11: (2) Zeigen Sie, daß Autoboxing mit Arrays nicht funktioniert. ■

Übungsaufgabe 12: (1) Erzeugen Sie per `CountingGenerator` ein initialisiertes Array von `double`-Werten und geben Sie es aus. ■

Übungsaufgabe 13: (2) Erzeugen Sie per `CountingGenerator.Character` ein `String`-Objekt. ■

Übungsaufgabe 14: (6) Erzeugen Sie zu jedem primitiven Typ ein Array und füllen Sie es per `CountingGenerator`. Geben Sie jedes Array aus. ■

Übungsaufgabe 15: (2) Ändern Sie das Beispiel *ContainerComparison.java*, indem Sie eine Generatorklasse für `BerylliumSphere`-Objekte anlegen. Ändern Sie die `main()`-Methode, so daß Sie die neue Generatorklasse über die `Generated`-Methode `array()` nutzt. ■

Übungsaufgabe 16: (3) Schreiben Sie, ausgehend vom Beispiel *CountingGenerator.java*, eine Klasse `SkipGenerator`, die mit der per Konstruktor übergebenen Schrittweite neue Werte erzeugt. Ändern Sie das Beispiel *TestArrayGeneration.java*, um zeigen zu können, daß Ihre neue Klasse funktioniert. ■

Übungsaufgabe 17: (5) Schreiben und testen Sie einen Generator für die Klasse `BigDecimal` und sorgen Sie dafür, daß der Generator mit den Methoden der Klasse `Generated` aufgerufen werden kann. ■

17.7 Hilfsmethoden für Arrays

[47] Die Klasse `java.util.Arrays` definiert verschiedene statische Hilfsmethoden für Arrays. Es gibt sechs grundlegende Methoden: `equals()` vergleicht zwei Arrays auf Gleichheit (es gibt eine Variante namens `deepEquals()` für mehrdimensionale Arrays). `fill()` wurde bereits in Unterabschnitt 17.6.1 behandelt. `sort()` sortiert ein Array. `binarySearch()` sucht nach einem Element in einem sortierten Array. `toString()` gibt eine `String`-Darstellung des Arrays zurück. `hashCode()` erzeugt den Hashwert eines Arrays (Sie lernen in Kapitel 18 was das bedeutet). Alle diese Methoden sind für jeden primitiven Typ sowie für `Object` überladen. Darüber hinaus erwartet die statische `Arrays`-Methode `asList()` eine Folge von Elementen oder ein Array und wandelt seine Eingabe in einen Container vom Typ *`List`* um. Diese Methode wurde in Kapitel 12 behandelt.

[48] Bevor wir die `Arrays`-Methoden diskutieren, betrachten wir noch eine nützliche Methode, die nicht zur Klasse `Arrays` gehört.

17.7.1 Kopieren eines Arrays: Die System-Methode arraycopy()

[49] Die Standardbibliothek von Java enthält die statische Methode `System.arraycopy()`, welche ein Array viel schneller kopiert, als wenn Sie eine `for`-Schleife anlegen, um die Kopie von Hand auszuführen. Die Methode ist überladen, um alle Typen annehmen zu können. Das folgende Beispiel führt die `arraycopy()`-Methode an `int`-Arrays vor:

```
//: arrays/CopyingArrays.java
// Using System.arraycopy()
import java.util.*;
import static net.mindview.util.Print.*;

public class CopyingArrays {
    public static void main(String[] args) {
        int[] i = new int[7];
        int[] j = new int[10];
        Arrays.fill(i, 47);
        Arrays.fill(j, 99);
        print("i = " + Arrays.toString(i));
        print("j = " + Arrays.toString(j));
        System.arraycopy(i, 0, j, 0, i.length);
        print("j = " + Arrays.toString(j));
        int[] k = new int[5];
        Arrays.fill(k, 103);
        System.arraycopy(i, 0, k, 0, k.length);
        print("k = " + Arrays.toString(k));
        Arrays.fill(k, 103);
        System.arraycopy(k, 0, i, 0, k.length);
        print("i = " + Arrays.toString(i));
        // Objects:
        Integer[] u = new Integer[10];
        Integer[] v = new Integer[5];
        Arrays.fill(u, new Integer(47));
        Arrays.fill(v, new Integer(99));
        print("u = " + Arrays.toString(u));
        print("v = " + Arrays.toString(v));
        System.arraycopy(v, 0, u, u.length/2, v.length);
        print("u = " + Arrays.toString(u));
    }
} /* Output:
      i = [47, 47, 47, 47, 47, 47, 47]
      j = [99, 99, 99, 99, 99, 99, 99, 99, 99, 99]
      j = [47, 47, 47, 47, 47, 47, 47, 99, 99, 99]
      k = [47, 47, 47, 47, 47]
      i = [103, 103, 103, 103, 103, 47, 47]
      u = [47, 47, 47, 47, 47, 47, 47, 47, 47, 47]
      v = [99, 99, 99, 99, 99]
      u = [47, 47, 47, 47, 47, 99, 99, 99, 99, 99]
   *///:~
```

Die Argumente von `arraycopy()` sind das Quellarray, die Startposition des Kopiervorgangs im Quellarray, das Zielarray, die Startposition im Zielarray, von der an die kopierten Elemente eingesetzt werden und die Anzahl der zu kopierenden Elemente. Das Verletzen der Arraygrenzen ruft selbstverständlich eine Ausnahme hervor.

[50] Das Beispiel zeigt, daß sowohl Arrays von primitiven Werten als auch Arrays von Objektreferenzen kopiert werden können. Beim Kopieren eines Arrays von Referenzen werden allerdings nur die Referenzen kopiert; die Objekte selbst, werden nicht dupliziert. Dieser Kopiermodus wird als *flache Kopie* bezeichnet (in den Online-Anhängen zu diesem Buch unter der Webadresse *http://www-.mindview.net* finden Sie weitere Einzelheiten).

[51] Die `System`-Methode `arraycopy()` führt kein Autoboxing beziehungsweise Autounboxing durch. Die Arrays müssen exakt demselben Typ angehören.

Übungsaufgabe 18: (3) Erzeugen und füllen Sie ein Array von `BerylliumSphere`-Objekten. Kopieren Sie dieses Array und zeigen Sie, daß es sich um eine flache Kopie handelt. ■

17.7.2 Vergleichen von Arrays: Die Arrays-Methode equals()

[52] Die Klasse `Arrays` hat eine `equals()`-Methode, um ganze Arrays miteinander vergleichen zu können. Die Methode ist für jeden primitiven Typ sowie für `Object` überladen. Gleichheit ist gegeben, wenn die Arrays dieselbe Anzahl von Elementen haben und jedes Element mit dem korrespondierenten Element im anderen Array übereinstimmt. (Bei Werten primitiven Typs wird die `equals()`-Methode der entsprechenden Wrapperklasse verwendet, beispielsweise `Integer.equals()` für `int`). Ein Beispiel:

```
//: arrays/ComparingArrays.java
// Using Arrays.equals()
import java.util.*;
import static net.mindview.util.Print.*;

public class ComparingArrays {
    public static void main(String[] args) {
        int[] a1 = new int[10];
        int[] a2 = new int[10];
        Arrays.fill(a1, 47);
        Arrays.fill(a2, 47);
        print(Arrays.equals(a1, a2));
        a2[3] = 11;
        print(Arrays.equals(a1, a2));
        String[] s1 = new String[4];
        Arrays.fill(s1, "Hi");
        String[] s2 = { new String("Hi"), new String("Hi"),
                        new String("Hi"), new String("Hi") };
        print(Arrays.equals(s1, s2));
    }
} /* Output:
      true
      false
      true
   *///:~
```

Ursprünglich stimmen die von `a1` und `a2` referenzierten Arrays exakt überein und die `equals()`-Methode liefert `true`. Wird ein Element geändert, so lautet das Ergebnis der Gleichheitsprüfung `false`. Im letzten Fall referenzieren alle Einträge von `s1` ein und dasselbe Objekt, `s2` dagegen fünf verschiedene Objekte. Die Gleichheitsprüfung von Arrays bezieht sich aber (per `Object.equals()`) auf den Inhalt, weshalb das Ergebnis `true` lautet.

Übungsaufgabe 19: (2) Schreiben Sie eine Klasse mit einem `int`-Feld, welches per Konstruktor initialisiert wird. Erzeugen Sie zwei Arrays von Objekten dieser Klasse mit für beide Arrays übereinstimmender Initialisierung und zeigen Sie, daß die `Arrays`-Methode `equals()` die beiden Arrays für verschieden hält. Ergänzen Sie Ihre Klasse um eine `equals()`-Methode, um das Problem zu beheben. ■

Übungsaufgabe 20: (4) Demonstrieren Sie die Methode `deepEquals()` für mehrdimensionale Arrays. ■

17.7.3 Vergleichen von Elementen: Die Interfaces Comparable und Comparator

[53] Das Sortieren setzt die Vergleichbarkeit der Elemente bezüglich ihres eigentlichen Typs voraus. Eine Lösung besteht natürlich darin, für jeden Typ eine individuelle Sortiermethode zu schreiben, wobei sich ein solcher Quelltext nicht für neue Typen wiederverwenden läßt.

[54] Eines der primären Ziele beim Design eines Programms besteht darin, die veränderlichen Dinge von den unveränderlichen Dingen zu trennen. In unserem Fall ist der allgemeine Sortieralgorithmus der unverändliche Teil, während sich das Vergleichskriterium für die Elemente von einem Aufruf zum nächsten ändert. Wir verwenden das *Strategy*-Entwurfsmuster[2], statt das Vergleichskriterium in viele verschiedene Sortiermethoden einzusetzen. Der veränderliche Teil ist dabei in einer separaten Klasse gekapselt (dem Strategieobjekt). Das Strategieobjekt wird dem stets gleichen Algorithmus übergeben, der erst durch die Strategie seine Funktionstüchtigkeit erhält. Auf diese Weise können Sie verschiedene Klasse anlegen, um verschiedene Vergleichsmöglichkeiten zu implementieren und Objekte dieser Klassen in ein und dasselbe Sortierverfahren einsetzen.

[55] Java kennt zwei Möglichkeiten, um Vergleichsfunktionalität zur Verfügung zu stellen. Die erste besteht darin, einer Klasse durch Implementieren des Interfaces *`java.lang.Comparable`* eine „natürliche“ Vergleichsmethode zu verleihen. Die Objekte einer Klasse, die das Interface *`Comparable`* implementiert, heißen **komparabel**. Das Interface ist sehr einfach und deklariert nur eine einzige Methode, nämlich `compareTo()`. Diese Methode erwartet ein anderes Objekt desselben Typs als Argument und liefert einen negativen Wert, wenn das aktuelle Objekt kleiner als das Argument ist, Null, wenn das aktuelle Objekt mit dem Argument übereinstimmt und einen positiven Wert, wenn das aktuelle Objekt größer als das Argument ist.

[56] Die Klasse `CompType` im folgenden Beispiel implementiert das Interface *`Comparable`* und führt die Vergleichbarkeit der Objekte mit Hilfe der `Arrays`-Methode `sort()` vor:

```
//: arrays/CompType.java
// Implementing Comparable in a class.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CompType implements Comparable<CompType> {
    int i;
    int j;
    private static int count = 1;
    public CompType(int n1, int n2) {
        i = n1;
        j = n2;
    }
    public String toString() {
        String result = ”[i = ” + i + ”, j = ” + j + ”]”;
        if(count++ % 3 == 0)
            result += ”\n”;
        return result;
    }
    public int compareTo(CompType rv) {
        return (i < rv.i ? -1 : (i == rv.i ? 0 : 1));
    }
    private static Random r = new Random(47);
    public static Generator<CompType> generator() {
        return new Generator<CompType>() {
```

[2] Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995). Siehe auch: *Thinking in Patterns, Problem-Solving Techniques using Java.*

```
                public CompType next() {
                    return new CompType(r.nextInt(100),r.nextInt(100));
                }
            };
        }
        public static void main(String[] args) {
            CompType[] a = Generated.array(new CompType[12], generator());
            print("before sorting:");
            print(Arrays.toString(a));
            Arrays.sort(a);
            print("after sorting:");
            print(Arrays.toString(a));
        }
    } /* Output:
          before sorting:
          [[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
          , [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
          , [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
          , [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
          ]
          after sorting:
          [[i = 9, j = 78], [i = 11, j = 22], [i = 16, j = 40]
          , [i = 20, j = 58], [i = 22, j = 7], [i = 51, j = 89]
          , [i = 58, j = 55], [i = 61, j = 29], [i = 68, j = 0]
          , [i = 88, j = 28], [i = 93, j = 61], [i = 98, j = 61]
          ]
      *///:~
```

Bei der Definition der Vergleichsmethode müssen Sie entscheiden, was es bedeutet, zwei Objekte dieser Klasse miteinander zu vergleichen. In diesem Beispiel gehen nur die `i`-Werte in das Vergleichskriterium ein, die `j`-Werte werden nicht beachtet.

[57] Die Methode `generator()` erzeugt mittels einer anonymen inneren Klasse ein Objekt einer Klasse, die das Interface *`Generator`* implementiert. Die `next()`-Methode erzeugt ein `CompType`-Objekt, dessen Felder mit Zufallszahlen initialisiert werden. In der `main()`-Methode wird das Generatorobjekt verwendet, um ein `CompType`-Array zu füllen, das anschließend sortiert wird. Wäre das Interface *`Comparable`* nicht implementiert worden, so würde beim Aufrufen der `sort()`-Methode zur Laufzeit eine Ausnahme vom Typ `ClassCastException` hervorgerufen werden, da die `sort()`-Methode ihr Argument in den Typ *`Comparable`* umwandelt.

[58] Stellen Sie sich nun vor, Sie müssen mit Objekten einer Klasse arbeiten, die das Interface *`Comparable`* nicht implementiert beziehungsweise *`Comparable`* zwar implementiert, aber die Funktionalität paßt nicht zu Ihren Anforderungen und Sie würden für diesen Typ lieber ein anderes Vergleichskriterium verwenden. Sie können das Problem lösen, indem Sie eine separate Klasse anlegen, die das in Kapitel 12 kurz vorgestellte Interface *`Comparator`* implementiert. Dieser Ansatz ist wiederum ein Beispiel für das *Strategy*-Entwurfsmuster. Das Interface deklariert die beiden Methoden `equals()` und `compare()`. Sie müssen die `equals()`-Methode aber nur bei speziellen Anforderungen hinsichtlich der Performanz implementieren, da Sie jedesmal, wenn Sie eine Klasse anlegen, implizit von der Basisklasse `Object` ableiten, die eine `equals()`-Methode definiert. Es genügt, die Standardversion von `equals()` aus der Klasse `Object` zu verwenden, um den durch das Interface auferlegten Vertrag zu erfüllen.

[59] Die Hilfsklasse `Collections` (die wir im nächsten Kapitel genauer betrachten) enthält die Methode `reverseOrder()`, die einen Komparator zurück, der die natürliche Sortierreihenfolge umgekehrt:

```
//: arrays/Reverse.java
// The Collections.reverseOrder() Comparator
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Reverse {
    public static void main(String[] args) {
        CompType[] a = Generated.array(new CompType[12], CompType.generator());
        print("before sorting:");
        print(Arrays.toString(a));
        Arrays.sort(a, Collections.reverseOrder());
        print("after sorting:");
        print(Arrays.toString(a));
    }
} /* Output:
      before sorting:
      [[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
      , [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
      , [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
      , [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
      ]
      after sorting:
      [[i = 98, j = 61], [i = 93, j = 61], [i = 88, j = 28]
      , [i = 68, j = 0], [i = 61, j = 29], [i = 58, j = 55]
      , [i = 51, j = 89], [i = 22, j = 7], [i = 20, j = 58]
      , [i = 16, j = 40], [i = 11, j = 22], [i = 9, j = 78]
      ]
   *///:~
```

Sie können auch einen eigenen Komparator schreiben. Der folgende Komparator vergleicht `CompType`-Objekte anhand ihrer `j`-Werte:

```
//: arrays/ComparatorTest.java
// Implementing a Comparator for a class.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class CompTypeComparator implements Comparator<CompType> {
    public int compare(CompType o1, CompType o2) {
        return (o1.j < o2.j ? -1 : (o1.j == o2.j ? 0 : 1));
    }
}

public class ComparatorTest {
    public static void main(String[] args) {
        CompType[] a = Generated.array(new CompType[12], CompType.generator());
        print("before sorting:");
        print(Arrays.toString(a));
        Arrays.sort(a, new CompTypeComparator());
        print("after sorting:");
        print(Arrays.toString(a));
    }
} /* Output:
      before sorting:
      [[i = 58, j = 55], [i = 93, j = 61], [i = 61, j = 29]
      , [i = 68, j = 0], [i = 22, j = 7], [i = 88, j = 28]
      , [i = 51, j = 89], [i = 9, j = 78], [i = 98, j = 61]
```

```
    , [i = 20, j = 58], [i = 16, j = 40], [i = 11, j = 22]
    ]
    after sorting:
    [[i = 68, j = 0], [i = 22, j = 7], [i = 11, j = 22]
    , [i = 88, j = 28], [i = 61, j = 29], [i = 16, j = 40]
    , [i = 58, j = 55], [i = 20, j = 58], [i = 93, j = 61]
    , [i = 98, j = 61], [i = 9, j = 78], [i = 51, j = 89]
    ]
  *///:~
```

Übungsaufgabe 21: (3) Versuchen Sie, das `BerylliumSphere`-Array aus Übungsaufgabe 18 zu sortieren. Implementieren Sie das Interface *`Comparator`*, um das Problem zu beheben. Legen Sie einen weiteren Komparator an, um die Arrayelemente in umgekehrter Reihenfolge zu sortieren. ■

17.7.4 Sortieren eines Arrays: Die Arrays-Methode sort()

[60] Die eingebauten Sortiermethoden gestatten Ihnen, ein beliebiges Array von Elementen primitives Typs oder Objektreferenzen zu sortieren, sofern im letzteren Fall die Klasse der Objekte entweder das Interface *`Comparable`* implementiert oder einen Komparator besitzt.[3] Das folgende Beispiel erzeugt `String`-Objekte mit zufälligem Inhalt und sortiert sie:

```
//: arrays/StringSorting.java
// Sorting an array of Strings.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class StringSorting {
    public static void main(String[] args) {
        String[] sa =
            Generated.array(new String[20], new RandomGenerator.String(5));
        print("Before sort: " + Arrays.toString(sa));
        Arrays.sort(sa);
        print("After sort: " + Arrays.toString(sa));
        Arrays.sort(sa, Collections.reverseOrder());
        print("Reverse sort: " + Arrays.toString(sa));
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        print("Case-insensitive sort: " + Arrays.toString(sa));
    }
} /* Output:
      Before sort: [YNzbr, nyGcF, OWZnT, cQrGs, eGZMm, JMRoE, suEcU, OneOE,
        dLsmw, HLGEa, hKcxr, EqUCB, bkIna, Mesbt, WHkjU, rUkZP, gwsqP, zDyCy,
        RFJQA, HxxHv]
      After sort: [EqUCB, HLGEa, HxxHv, JMRoE, Mesbt, OWZnT, OneOE, RFJQA,
        WHkjU, YNzbr, bkIna, cQrGs, dLsmw, eGZMm, gwsqP, hKcxr, nyGcF, rUkZP,
        suEcU, zDyCy]
      Reverse sort: [zDyCy, suEcU, rUkZP, nyGcF, hKcxr, gwsqP, eGZMm, dLsmw,
        cQrGs, bkIna, YNzbr, WHkjU, RFJQA, OneOE, OWZnT, Mesbt, JMRoE, HxxHv,
        HLGEa, EqUCB]
      Case-insensitive sort: [bkIna, cQrGs, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr,
        HLGEa, HxxHv, JMRoE, Mesbt, nyGcF, OneOE, OWZnT, RFJQA, rUkZP, suEcU,
        WHkjU, YNzbr, zDyCy]
  *///:~
```

[3] Überraschenderweise wurde das Sortieren von `String`-Objekten in den Java-Versionen 1.0 und 1.1 nicht unterstützt.

Beachten Sie die *lexikographische Sortierung* der Zeichenketten: Zuerst werden die „Worte“ ausgegeben, die mit einem Großbuchstaben beginnen. Daran schließen sich die Worte an, die mit einem Kleinbuchstaben beginnen. (Telefonbücher verwenden typischerweise diesen Sortiermodus.) Wenn Sie die Worte ohne Berücksichtigung der Groß-/Kleinschreibung sortieren möchten, übergeben Sie der `sort()`-Methode die Konstante `CASE_INSENSITIVE_ORDER` (siehe oben).

[61] Der in der Standardbibliothek von Java verwendete Sortieralgorithmus ist optimal an den Typ der sortierten Elemente angepaßt, das heißt Quicksort für Elemente primitiven Typs und ein stabiler Mergesort-Algorithmus für Objekte. Sie brauchen sich über die Performanz keine Gedanken zu machen, es sei denn, Ihr Profiler stellt fest, daß Ihr Sortierverfahren einen Engpaß bildet.

17.7.5 Suche in einem sortieren Array: Die Arrays-Methode binarySearch()

[62] Ist ein Array sortiert, so gestattet die `Arrays`-Methode `binarySearch()` schnelles Suchen nach einem bestimmten Element. Auf ein unsortiertes Array angewandt liefert `binarySearch()` dagegen kein vorhersagbares Ergebnis. Das folgende Beispiel verwendet einen Generator vom Typ `RandomGenerator.Integer`, um ein Array zu füllen und einen Suchwert zu wählen:

```
//: arrays/ArraySearching.java
// Using Arrays.binarySearch().
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ArraySearching {
    public static void main(String[] args) {
        Generator<Integer> gen = new RandomGenerator.Integer(1000);
        int[] a = ConvertTo.primitive(Generated.array(new Integer[25], gen));
        Arrays.sort(a);
        print("Sorted array: " + Arrays.toString(a));
        while(true) {
            int r = gen.next();
            int location = Arrays.binarySearch(a, r);
            if(location >= 0) {
                print("Location of " + r + " is " + location +
                      ", a[" + location + "] = " + a[location]);
                break; // Out of while loop
            }
        }
    }
} /* Output:
      Sorted array: [128, 140, 200, 207, 258, 258, 278, 288, 322, 429, 511, 520,
522, 551, 555, 589, 693, 704, 809, 861, 861, 868, 916, 961, 998]
      Location of 322 is 8, a[8] = 322
   *///:~
```

Die `while`-Schleife erzeugt solange zufällige Suchwerte, bis einer davon im Array gefunden wird.

[63] Die `Arrays`-Methode `binarySearch()` gibt einen positiven Wert zurück, wenn sie das gesuchte Element findet. Andernfalls gibt ein negativer Rückgabewert die Stelle an, an der das gesuchte Element unter Aufrechterhaltung der Sortierung eingeordnet werde müßte. Der zurückgelieferte Wert ist

```
-(insertion point) - 1
```

Der Einsetzungspunkt ist der Index des ersten Elementes, welches größer als das gesuchte Element ist beziehungsweise `a.size()`, falls alle Elemente des Arrays kleiner als das gesuchte Element sind.

[64] Enthält ein Array ein Element mehr als einmal, so besteht keine Garantie dafür, welches Exemplar gefunden wird. Der Suchalgorithmus unterstützt Duplikate entwurfsbedingt nicht, toleriert aber ihre Anwesenheit. Wenn Sie eine sortierte Liste ohne Duplikate brauchen, können Sie ein `TreeSet`-Objekt verwenden, wobei die Sortierung der Elemente erhalten bleibt oder ein `LinkedHashSet`-Objekt, das die Einsetzungsreihenfolge erhält.. Die Objekte dieser Klassen kümmern sich automatisch um alle Einzelheiten. Sie sollten diese Klassen nur bei Performanzengpässen durch ein von Hand gepflegtes Array ersetzen.

[65] Wenn Sie ein Array von Objektreferenz per Komparator sortieren (bei Elementen primitiven Typs nicht möglich), müssen Sie bei der binären Suche mittels `binarySearch()` denselben Kompatator übergeben (es gibt eine entsprechende überladene Version der Methode). Das Beispiel *StringSorting.java* von Seite 610 läßt sich so ändern, daß es eine binäre Suche ausführt:

```
//: arrays/AlphabeticSearch.java
// Searching with a Comparator.
import java.util.*;
import net.mindview.util.*;

public class AlphabeticSearch {
    public static void main(String[] args) {
        String[] sa =
            Generated.array(new String[30], new RandomGenerator.String(5));
        Arrays.sort(sa, String.CASE_INSENSITIVE_ORDER);
        System.out.println(Arrays.toString(sa));
        int index =
            Arrays.binarySearch(sa, sa[10], String.CASE_INSENSITIVE_ORDER);
        System.out.println("Index: "+ index + "\n"+ sa[index]);
    }
} /* Output:
      [bkIna, cQrGs, cXZJo, dLsmw, eGZMm, EqUCB, gwsqP, hKcxr, HLGEa, HqXum, HxxHv,
        JMRoE, JmzMs, Mesbt, MNvqe, nyGcF, ogoYW, OneOE, OWZnT, RFJQA, rUkZP, sgqia,
        slJrL, suEcU, uTpnX, vpfFv, WHkjU, xxEAJ, YNzbr, zDyCy]
      Index: 10
      HxxHv
   *///:~
```

Der Komparator wird `binarySearch()` als drittes Argument übergeben. In diesem Beispiel wird das gesuchte Element aus dem Array gewählt, so daß der Erfolg sicher ist.

Übungsaufgabe 22: (2) Zeigen Sie, daß das Ergebnis von `binarySearch()` auf einem unsortierten Array nicht vorhersagbar ist. ■

Übungsaufgabe 23: (2) Erzeugen Sie ein Array von `Integer`-Elementen, füllen Sie es mit zufällig gewählten `int`-Werten (per Autoboxing) und sortieren Sie es mittels Komparator in umgekehrter Reihenfolge. ■

Übungsaufgabe 24: (3) Zeigen Sie, daß die Objekte der Klasse aus Übungsaufgabe 19 gesucht werden können, also vergleichbar sind. ■

17.8 Zusammenfassung

[66] In diesem Kapitel haben Sie gesehen, daß Java systemnahe Arrays fester Länge in angemessener Weise unterstützt. Diese Art von Arrays gewichtet, analog zu C und C++, Performanz höher als Flexibilität. In der ersten Java-Version waren systemnahe Arrays fester Länge absolut notwendig, nicht nur weil sich die Designer von Java entschlossen hatten, primitive Typen in den Sprachumfang

aufzunehmen (ebenfalls aus Performanzgründen), sondern auch weil Container in dieser frühen Version nur in minimalem Umfang unterstützt wurden. Bei den frühen Java-Versionen war es stets sinnvoll, Arrays zu wählen.

[67] In den folgenden Java-Versionsn wurde die Unterstützung von Containern erheblich verbessert und heute neigen Container dazu, Arrays in jeder Hinsicht, mit Ausnahme der Performanz, in den Schatten zu stellen, wobei aber auch die Performanz der Container deutlich verbessert wurde. Wie bereits an mehreren Stellen in diesem Buch erwähnt, liegen Performanzprobleme aber in der Regel nicht dort, wo Sie sie sich vorstellen.

[68] Durch Autoboxing und generische Typen lassen sich Werte primitiven Typs mühelos in Containern speichern, wodurch Sie zusätzlich ermuntert werden, systemnahe Arrays durch Container zu ersetzen. Nachdem generische Typen typsichere Container ermöglichen, sind Arrays in dieser Hinsicht nicht länger im Vorteil.

[69] Generische Typen verhalten sich Arrays gegenüber ziemlich feindselig, wie bereits in diesem Kapitel beschrieben und wie Sie selbst sehen werden, wenn Sie sie benutzen. Selbst wenn es Ihnen die Verknüpfung von Arrays und generischen Typen gelingt (siehe Kapitel **??**), werden während der Übersetzung häufig „unchecked“-Warnungen ausgegeben.

[70] Designer der Programmiersprache Java haben mir gelegentlich direkt geraten, Container anstelle von Arrays zu verwenden, als wir bestimmte Beispiele diskutiert haben (ich hatte Arrays gewählt, um gewisse Dinge zu zeigen, hatte also keine andere Wahl).

[71] Alle diese Dinge deuten darauf hin, daß Sie Container gegenüber Arrays bevorzugen sollten, wenn Sie mit einer aktuellen Java-Version programmieren. Nur wenn die Performanz erwiesenermaßen leidet und sich ein Array anstelle eines Container tatsächlich als vorteilhaft erweist, sollten Sie Arrays wählen.

[72] Dies ist eine ziemlich kühne Behauptung, aber es gibt Sprachen, die überhaupt keine systemnahen Arrays fester Länge haben. Diese Sprachen verfügen nur über längenveränderliche Container mit erheblich mehr Funktionalität als die Arrays von C, C++ oder Java. Python (*www.python.org*) hat beispielsweise einen Typ `list`, der einfache Arraysyntax unterstützt und viel mehr Funktionalität hat. Sie können sogar Klassen daraus ableiten:

```
aList = [1, 2, 3, 4, 5]
print type(aList) # <type 'list'>
print aList # [1, 2, 3, 4, 5]
print aList[4] # 5 Basic list indexing
aList.append(6) # lists can be resized
aList += [7, 8] # Add a list to a list
print aList # [1, 2, 3, 4, 5, 6, 7, 8]
aSlice = aList[2:4]
print aSlice # [3, 4]

class MyList(list): # Inherit from list
    # Define a method, 'this' pointer is explicit:
    def getReversed(self):
        reversed = self[:] # Copy list using slices
        reversed.reverse() # Built-in list method
        return reversed

list2 = MyList(aList) # No 'new' needed for objects creation
print type(list2) # <class '__main__.MyList'>
print list2.getReversed() # [8, 7, 6, 5, 4, 3, 2, 1]
#:~
```

[73] Die grundlegende Python-Syntax wurde in Abschnitt 16.16 erläutert. Hier wird eine Liste

erzeugt, indem eine kommaseparierte Folge von Objekten in eckige Klammern gesetzt wird. Das Resultat ist ein Objekt des Laufzeittyps `list` (die Ausgaben der `print()`-Anweisungen sind in derselben Zeile als Kommentar angegeben). Das Ausdrucken einer Liste liefert dasselbe Ergebnis wie die `Arrays`-Methode `toString()` bei Java.

[74] Der `:`-Operator gestattet zusammen mit dem `[]`-Operator eine Teilfolge aus einer Liste auszuwählen. Der Typ `list` hat noch viele weitere eingebaute Fähigkeiten.

[75] `MyList` ist eine Klassendefinition, wobei die Basisklassen im folgenden Klammernpaar angegeben werden. Im Körper der Klasse leitet `def` eine Methodendefinition ein, wobei als erstes Argument der Methode stets automatisch und explizit das Äquivalent der Selbstreferenz `this` übergeben wird (bei Python konventionsgemäß `self`; kein Schlüsselwort). Beachten Sie die automatische Ableitung des Konstruktors.

[76] In Python ist zwar tatsächlich *alles* Objekt (inklusive der Typen für Ganz- und Fließkommazahlen), aber es gibt eine Hintertür: Zur Optimierung der Performanz Ihres Programms können Sie kritische Abschnitte als Erweiterungen in C oder C++ schreiben beziehungsweise ein spezielles Werkzeug namens Pyrex verwenden, das dazu dient Ihr Programm zu beschleunigen. Auf diese Weise erhalten Sie die Objektreinheit ohne auf Performanzverbesserungen verzichten zu müssen.

[77] PHP (*http://www.php.net*) geht sogar noch einen Schritt weiter und hat nur einen einzigen Arraytyp, der sich sowohl als indiziertes als auch als assoziatives Array (`Map`) verwenden läßt.

[78] Es ist interessant, darüber zu spekulieren, ob die Designer von Java nach den vergangenen Jahren in der Java-Evolution primitive Typen und systemnahe Arrays in die Sprache integrieren würden, wenn sie noch einmal von vorne anfangen würden. Ohne primitive Typen und Arrays wäre es möglich, eine echte rein objektorientierte Sprache zu entwickeln (anderslautenden Behauptungen zum Trotz ist Java, gerade durch die systemnahen Überreste, keine rein objektorientierte Sprache). Das Argument „Effizienz“ ist scheinbar immer zwingend, aber im Laufe der Zeit hat sich eine Evolution weg von diesem Gedanken und hin zum Einsatz weniger systemnaher Komponenten wie Containern abgezeichnet. Könnten Container, wie bei anderen Sprachen, darüber hinaus in den Kern der Sprache integriert werden, so hätte der Compiler bessere Optimierungsmöglichkeiten.

[79] *Green-fields speculation aside, we are certainly stuck with arrays, and you will see them when reading code. Containers, however, are almost always a better choice.*

Übungsaufgabe 25: (3) Schreiben Sie das Beispiel *PythonLists.py* in Java neu. ■

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 18

Die Containerbibliothek in allen Einzelheiten

Inhaltsübersicht

[0] In Kapitel 12 „Einführung in die Containerbibliothek“ wurden die Grundzüge und die wichtigste Funktionalität der Containerbibliothek von Java vorgestellt. Dieser Umfang reicht aus, um in der Arbeit mit Containern auf den richtigen Weg zu kommen. Dieses Kapitel untersucht diese wichtige Bibliothek in allen Einzelheiten.

[1] Wenn Sie die Containerbibliothek in vollem Umfang einsetzen möchten, müssen Sie mehr wissen, als in Kapitel 12 beschrieben wird. Dieses Kapitel setzt allerdings fortgeschrittene Konzepte wie generische Typen voraus und ist dementsprechend weiter hinten in diesem Buch eingeordnet.

[2] Nach einem umfassenderen Überblick über Container lernen Sie wie Hashalgorithmen funktionieren und welche Eigenschaften die Methoden `hashCode()` und `equals()` der Elemente erfüllen müssen, um in einem hashbasierten Container bestimmungsgemäß zu arbeiten.

18.1 Übersicht über die Containerbibliothek

[3] Abbildung [Seite 439 im Buch] in Abschnitt 12.14 zeigte eine vereinfachte Übersicht über die Containerbibliothek von Java. Abbildung [Seite 792 im Buch] zeigt eine umfassendere Übersicht mit abstrakten Klassen und veralteten Komponenten (ohne Implementierungen des *`Queue`*-Interfaces).

[4] Neue Klassen und Interfaces in Version 5 der Java Standard Edition (SE 5):

- Das Interface *`Queue`* (die Klasse `LinkedList` wurde geändert um dieses Interface zu implementieren, siehe Abschnitt 12.11) sowie seine Implementierung `PriorityQueue` und verschiedene Implementierungen des Interfaces *`BlockingQueue`*, siehe Unterabschnitte 22.5.4, 22.7.3 und 22.7.4 sowie Abschnitte 22.8 und 22.9 (Seiten 22.9 und 22.9.2).
- Das Interface *`ConcurrentMap`* sowie seine Implementierung `ConcurrentHashMap`, die ebenfalls der Threadprogrammierung dienen und in Unterabschnitt 22.9.2 gezeigt werden (Seiten 22.9.2 und 22.9.2.2).
- Die ebenfalls in der Threadprogrammierung verwendeten Klassen `CopyOnWriteArrayList` und `CopyOnWriteArraySet`, siehe Unterabschnitte 22.7.7 und 22.9.2 sowie Abschnitt 22.10 (Seite 1001).
- Die speziellen Implementierungen `EnumSet` und `EnumMap` der Interfaces *`Set`* und *`Map`* für Aufzählungstypen in Kapitel 20.
- Verschiedene Hilfsmethoden in der Klasse `Collections`.

[5] Die lang-gestrichelten Rahmen repräsentieren abstrakte Klassen (Sie sehen einige Klassen, deren Namen mit „`Abstract`“ beginnen). Die abstrakten Klassen mögen auf den ersten Blick störend

wirken, sind aber Hilfsklassen, die ein bestimmtes Interface teilweise implementieren. Wenn Sie beispielsweise das Interface *Set* selbst implementieren möchten, beginnen Sie nicht bei *Set* selbst und legen sämtliche Methoden an, sondern leiten Ihre Klasse von `AbstractSet` ab und konzentrieren sich auf die restliche Implementierung. Die Containerbibliothek bietet allerdings genügend Funktionalität, um fast jedes Bedürfnis befriedigen zu können, so daß Sie die abstrakten Klassen in der Regel nicht zu beachten brauchen.

18.2 Erzeugen von Testdaten

[6] Das Problem, den Inhalt eines Containers auszugeben, ist zwar gelöst, das Füllen eines Containers mit Testdaten leidet aber unter derselben Schwachstelle wie die Hilfsklasse `java.util.Arrays`. Das Äquivalent von `Arrays` in der Containerbibliothek ist die Klasse `Collections` mit einer Reihe von statischen Hilfsmethoden, darunter `fill()`. Wie die gleichnamige `Arrays`-Methode verteilt `Collections.fill()` eine einzige Objektreferenz auf alle Speicherplätze des Containers. Außerdem akzeptiert die `fill()`-Methode nur Container vom Typ *List* (allerdings kann der von `fill()` modifizierte Container einem Konstruktor beziehungsweise einer `addAll()`-Methode übergeben werden):

```
//: containers/FillingLists.java
// The Collections.fill() & Collections.nCopies() methods.
import java.util.*;

class StringAddress {
    private String s;
    public StringAddress(String s) { this.s = s; }
    public String toString() {
        return super.toString() + ” ” + s;
    }
}

public class FillingLists {
    public static void main(String[] args) {
        List<StringAddress> list =
            new ArrayList<StringAddress>(
                Collections.nCopies(4, new StringAddress(”Hello”)));
        System.out.println(list);
        Collections.fill(list, new StringAddress(”World!”));
        System.out.println(list);
    }
} /* Output: (Sample)
      [StringAddress@82ba41 Hello, StringAddress@82ba41 Hello,
        StringAddress@82ba41 Hello, StringAddress@82ba41 Hello]
      [StringAddress@923e30 World!, StringAddress@923e30 World!,
        StringAddress@923e30 World!, StringAddress@923e30 World!]
   *///:~
```

Dieses Beispiel zeigt zwei Möglichkeiten, um einen Container vom Typ *Collection* mit Referenzen auf ein einziges Objekt zu füllen. Die erste, `Collections.nCopies()`, erzeugt einen Container vom Typ *List*, der dem Konstruktor der Klasse `ArrayList` übergeben wird und einen Container dieses Typs füllt.

[7] Die `toString()`-Methode der Klasse `StringAddress` ruft die `Object`-Version von `toString()` auf, die den Klassennamen, gefolgt von der vorzeichenlosen Hexadezimaldarstellung des Hashwertes des Objektes (von der `hashCode()`-Methode generiert) zurückgibt. Sie erkennen an der Ausgabe, daß alle Referenzen auf dasselbe Objekt verweisen. Das gilt auch nach dem Aufrufen der zweiten Methode, `Collections.fill()`. Der Nutzwert der `fill()`-Methode ist zusätzlich dadurch eingeschränkt,

daß sie nur vorhandene Elemente des *List*-Containers ersetzen, aber keine weiteren Elemente hinzufügen kann.

18.2.1 Ein Generator für Elemente von Containern des Typs Collection

[8–10] Fast alle Containertypen unter dem Interface *Collection* verfügen über einen Konstruktor, der wiederum einen Container vom Typ *Collection* erwartet, mit dessen Inhalt der neue Container gefüllt wird. Um mühelos Testdaten erzeugen zu können, genügt es, eine Klasse zu schreiben, deren Konstruktor ein Argument vom Typ *Generator* (das Generatorkonzept und das Interface *Generator* wurden in Abschnitt 16.3 eingeführt und in Abschnitt 17.6.2 weiter untersucht) sowie die Anzahl zu erzeugender Elemente (`quantity`) erwartet:

```
//: net/mindview/util/CollectionData.java
// A Collection filled with data using a generator object.
package net.mindview.util;
import java.util.*;

public class CollectionData<T> extends ArrayList<T> {
    public CollectionData(Generator<T> gen, int quantity) {
        for(int i = 0; i < quantity; i++)
            add(gen.next());
    }
    // A generic convenience method:
    public static <T> CollectionData<T>
        list(Generator<T> gen, int quantity) {
        return new CollectionData<T>(gen, quantity);
    }
} ///:~
```

Die Klasse `CollectionData` verwendet einen Generator, um soviele Objekte als benötigt in den Container einzusetzen. Der erzeugte Container vom Typ `ArrayList` kann dem Konstruktor einer beliebigen Containerklasse unter dem Interface *Collection* übergeben werden, der die Testdaten in den neuen Container kopiert. Die im Interface *Collection* deklarierte Methode `addAll()` kann ebenfalls verwendet werden, um einen vorhandenen Container dieses Typs zu füllen. Die generische Methode `fill()` erspart Tippaufwand beim Verwenden dieser Klasse. Die Klasse `CollectionData` ist ein Beispiel für das *Adapter*-Entwurfsmuster:[1] Sie verbindet einen Generator mit dem Konstruktor einer Containerklasse unter dem Interface *Collection*.

[11] Das folgende Beispiel initialisiert einen Container vom Typ `LinkedHashSet`:

```
//: containers/CollectionDataTest.java
import java.util.*;
import net.mindview.util.*;

class Government implements Generator<String> {
    String[] foundation = (''strange women lying in ponds '' +
                           ''distributing swords is no basis for a system of '' +
                           ''government'').split(('' '');
    private int index;
    public String next() { return foundation[index++]; }
}

public class CollectionDataTest {
    public static void main(String[] args) {
```

[1] Dieses entspricht dem *Adapter*-Entwurfsmuster eventuell nicht im strengen Sinn, das heißt der Definition im GoF-Buch (Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995)), trifft aber den Kerngedanken.

```
        Set<String> set =
            new LinkedHashSet<String>(
                new CollectionData<String>(new Government(), 15));
        // Using the convenience method:
        set.addAll(CollectionData.list(new Government(), 15));
        System.out.println(set);
    }
} /* Output:
      [strange, women, lying, in, ponds, distributing, swords, is, no, basis,
        for, a, system, of, government]
   *///:~
```

Die Elemente werden in der Reihenfolge ausgegeben, in der sie eingesetzt wurden, da die Klasse `LinkedHashSet` eine verkettete Liste implementiert, welche die Einsetzungsreihenfolge erhält.

[12] Alle in Abschnitt 17.6.2 definierten Generatorklassen können nun mittels der Adapterklasse `CollectionData` verwendet werden. Das folgende Beispiel wählt die Generatorklassen `RandomGenerator.String` und `RandomGenerator.Integer`:

```
//: containers/CollectionDataGeneration.java
// Using the Generators defined in the Arrays chapter.
import java.util.*;
import net.mindview.util.*;

public class CollectionDataGeneration {
    public static void main(String[] args) {
        System.out.println(new ArrayList<String>(
            CollectionData.list( // Convenience method
                                 new RandomGenerator.String(9), 10)));
        System.out.println(new HashSet<Integer>(
            new CollectionData<Integer>(
                new RandomGenerator.Integer(), 10)));
    }
} /* Output:
      [YNzbrnyGc, FOWZnTcQr, GseGZMmJM, RoEsuEcUO, neOEdLsmw,
        HLGEahKcx, rEqUCBbkI, naMesbtWH, kjUrUkZPg, wsqPzDyCy]
      [573, 4779, 871, 4367, 6090, 7882, 2017, 8037, 3455, 299]
   *///:~
```

Die Anzahl der Buchstaben in den von `RandomGenerator.String` erzeugten `String`-Objekte wird über das zweite Konstruktorargument eingestellt.

18.2.2 Ein Paar-Generator für Elemente von Containern des Typs Map

[13] Wir verwenden denselben Ansatz, um Container vom Typ *`Map`* mit Testdaten zu füllen. Dazu benötigen wir eine `Pair`-Klasse, da die `next()`-Methode des Generators ein Paar (einen Schlüssel und einen Wert) liefern muß:

```
//: net/mindview/util/Pair.java
package net.mindview.util;

public class Pair<K,V> {
    public final K key;
    public final V value;
    public Pair(K k, V v) {
        key = k;
        value = v;
```

```
    }
} ///:~
```

Die Felder `key` und `value` sind als öffentliche und finale Felder deklariert, so daß die Klasse `Pair` nur-lesbare Transferobjekte (Entwurfsmuster *Data-Transfer-Object*) liefert.

[14] Die Adapterklasse `MapData` implementiert verschiedene Kombinationen von Generatoren, iterablen Klassen und konstanten Werten, mit denen Sie einen Hilfscontainer vom Typ *`Map`* erzeugen können:

```
//: net/mindview/util/MapData.java
// A Map filled with data using a generator object.
package net.mindview.util;
import java.util.*;

public class MapData<K,V> extends LinkedHashMap<K,V> {
    // A single Pair Generator:
    public MapData(Generator<Pair<K,V>> gen, int quantity) {
        for(int i = 0; i < quantity; i++) {
            Pair<K,V> p = gen.next();
            put(p.key, p.value);
        }
    }
    // Two separate Generators:
    public MapData(Generator<K> genK, Generator<V> genV, int quantity) {
        for(int i = 0; i < quantity; i++) {
            put(genK.next(), genV.next());
        }
    }
    // A key Generator and a single value:
    public MapData(Generator<K> genK, V value, int quantity){
        for(int i = 0; i < quantity; i++) {
            put(genK.next(), value);
        }
    }
    // An Iterable and a value Generator:
    public MapData(Iterable<K> genK, Generator<V> genV) {
        for(K key : genK) {
            put(key, genV.next());
        }
    }
    // An Iterable and a single value:
    public MapData(Iterable<K> genK, V value) {
        for(K key : genK) {
            put(key, value);
        }
    }
    // Generic convenience methods:
    public static <K,V> MapData<K,V>
        map(Generator<Pair<K,V>> gen, int quantity) {
        return new MapData<K,V>(gen, quantity);
    }
    public static <K,V> MapData<K,V>
        map(Generator<K> genK, Generator<V> genV, int quantity) {
        return new MapData<K,V>(genK, genV, quantity);
    }
    public static <K,V> MapData<K,V>
        map(Generator<K> genK, V value, int quantity) {
```

```
        return new MapData<K,V>(genK, value, quantity);
    }
    public static <K,V> MapData<K,V>
        map(Iterable<K> genK, Generator<V> genV) {
        return new MapData<K,V>(genK, genV);
    }
    public static <K,V> MapData<K,V>
        map(Iterable<K> genK, V value) {
        return new MapData<K,V>(genK, value);
    }
} ///:~
```

Mit der Adapterklasse `MapData` haben Sie die Wahl zwischen einem einzigen Generator vom Typ *`Generator<Pair<K,V>>`*, einem Generator und einem konstanten Wert, einer iterablen Klasse (darunter jeder Container vom Typ *`Collection`*) und einem Generator oder einer iterablen Klassen und einem konstanten Wert. Die generischen `map()`-Methoden ersparen Tippaufwand beim Erzeugen eines `MapData`-Objektes.

[15] Das folgende Beispiel zeigt Anwendungen der Adapterklasse `MapData`. Die Generatorklasse `Letters` implementiert das Interface *`Iterable`*, liefert also einen Iterator und kann somit zum Testen der `map()`-Methoden in `MapData` verwendet werden, die ein Argument vom Typ *`Iterable`* erwarten:

```
//: containers/MapDataTest.java
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Letters implements Generator<Pair<Integer,String>>, Iterable<Integer> {
    private int size = 9;
    private int number = 1;
    private char letter = 'A';
    public Pair<Integer,String> next() {
        return new Pair<Integer,String>(number++, "" + letter++);
    }
    public Iterator<Integer> iterator() {
        return new Iterator<Integer>() {
            public Integer next() { return number++; }
            public boolean hasNext() { return number < size; }
            public void remove() {
                throw new UnsupportedOperationException();
            }
        };
    }
}

public class MapDataTest {
    public static void main(String[] args) {
        // Pair Generator:
        print(MapData.map(new Letters(), 11));
        // Two separate generators:
        print(MapData.map(new CountingGenerator.Character(),
                          new RandomGenerator.String(3), 8));
        // A key Generator and a single value:
        print(MapData.map(new CountingGenerator.Character(), "Value", 6));
        // An Iterable and a value Generator:
        print(MapData.map(new Letters(), new RandomGenerator.String(3)));
        // An Iterable and a single value:
        print(MapData.map(new Letters(), "Pop"));
```

```
    }
} /* Output:
      {1=A, 2=B, 3=C, 4=D, 5=E, 6=F, 7=G, 8=H, 9=I, 10=J, 11=K}
      {a=YNz, b=brn, c=yGc, d=FOW, e=ZnT, f=cQr, g=Gse, h=GZM}
      {a=Value, b=Value, c=Value, d=Value, e=Value, f=Value}
      {1=mJM, 2=RoE, 3=suE, 4=cUO, 5=neO, 6=EdL, 7=smw, 8=HLG}
      {1=Pop, 2=Pop, 3=Pop, 4=Pop, 5=Pop, 6=Pop, 7=Pop, 8=Pop}
   *///:~
```

Dieses Beispiel nutzt auch die Generatorklassen aus Abschnitt 17.6.2. Sie können mit den Hilfsklassen `CollectionData` und `MapData` aus diesem und dem vorigen Unterabschnitt einen Hilfscontainer vom Typ *`Collection`* oder *`Map`* mit beliebigem Inhalt erzeugen und den eigentlichen Container per Konstruktor, *`Map`*`.putAll()` oder *`Collection`*`addAll()` mit dem Inhalt des Hilfscontainers initialisieren.

18.2.3 Verwendung der abstrakten Klassen (Teilimplementierungen)

[16] Eine alternative Lösung des Problems, Testdaten für einen Container zu erzeugen, besteht darin, das Interface *`Collection`* beziehungsweise *`Map`* individuell zu implementieren. Zu jedem Containerinterface im Package `java.util` existiert eine abstrakte Klasse, die eine teilweise Implementierung des jeweiligen Containertyps liefert, so daß Sie lediglich die erforderlichen Methoden anlegen müssen, um den gewünschten Container zu erhalten. Ist der resultierende Container nur-lesbar, wie bei Testdaten typischerweise der Fall, so ist die Anzahl der Methoden, die Sie implementieren müssen minimal.

[17] Obwohl in diesem Fall nicht notwendig, bietet der folgende Lösungsvorschlag eine Gelegenheit, um das Entwurfsmuster *Flyweight* vorzuführen. Dieses Entwurfsmuster wird angewendet, wenn die gewöhnliche Lösung zu viele Objekte benötigt oder wenn das Erzeugen der normalen Objekte zu viel Arbeitsspeicher beansprucht. Das Entwurfsmuster *Flyweight* lagert einen Teil des Objektes aus, so daß ein Teil oder sogar das gesamte Objekt aus einer effizienteren externen Tabelle (oder mit Hilfe eines anderen speicherfreundlichen Verfahrens) abgefragt werden kann.

[18] Das folgende Beispiel soll insbesondere zeigen, wie vergleichsweise einfach sich eine individuelle Implementierung des *`Map`*- beziehungsweise *`Collection`*-Interfaces mit Hilfe der abstrakten Klassen im Package `java.util` bewerkstelligen läßt. Wenn Sie eine nur-lesbare Containerklasse vom Typ *`Map`* schreiben möchten, genügt es, sie von `AbstractMap` abzuleiten und die `entrySet()`-Methode zu implementieren. Wenn Sie eine nur-lesbare Containerklasse vom Typ *`Set`* brauchen, leiten Sie sie von `AbstractSet` ab und legen die beiden Methoden `iterator()` und `size()` an.

[19] Das folgende Beispiel verwendet die Länder der Welt und ihre Hauptstädte als Testdaten.[2] Die Methode `capitals()` liefert einen Container vom Typ *`Map`*, der Ländernamen auf die Namen der Hauptstädte abbildet. Die Methode `names()` gibt einen Container vom Typ *`List`* zurück, der die Ländernamen enthält. Beide Methoden sind überladen und besitzen je eine Version, die ein `int`-Argument erwartet, welches die Anzahl der Elemente einer Teilliste angibt:

```
//: net/mindview/util/Countries.java
// ''Flyweight'' Maps and Lists of sample data.
package net.mindview.util;
import java.util.*;
import static net.mindview.util.Print.*;

public class Countries {
    public static final String[][] DATA = {
```

[2] Die Daten stammen aus dem Internet. Im Laufe der Zeit haben Leser verschiedene Korrekturen geschickt.

```
// Africa
{''ALGERIA'',''Algiers''}, {''ANGOLA'',''Luanda''},
{''BENIN'',''Porto-Novo''}, {''BOTSWANA'',''Gaberone''},
{''BURKINA FASO'',''Ouagadougou''},
{''BURUNDI'',''Bujumbura''},
{''CAMEROON'',''Yaounde''}, {''CAPE VERDE'',''Praia''},
{''CENTRAL AFRICAN REPUBLIC'',''Bangui''},
{''CHAD'',''N'djamena''}, {''COMOROS'',''Moroni''},
{''CONGO'',''Brazzaville''}, {''DJIBOUTI'',''Dijibouti''},
{''EGYPT'',''Cairo''}, {''EQUATORIAL GUINEA'',''Malabo''},
{''ERITREA'',''Asmara''}, {''ETHIOPIA'',''Addis Ababa''},
{''GABON'',''Libreville''}, {''THE GAMBIA'',''Banjul''},
{''GHANA'',''Accra''}, {''GUINEA'',''Conakry''},
{''BISSAU'',''Bissau''},
{''COTE D'IVOIR (IVORY COAST)'',''Yamoussoukro''},
{''KENYA'',''Nairobi''}, {''LESOTHO'',''Maseru''},
{''LIBERIA'',''Monrovia''}, {''LIBYA'',''Tripoli''},
{''MADAGASCAR'',''Antananarivo''}, {''MALAWI'',''Lilongwe''},
{''MALI'',''Bamako''}, {''MAURITANIA'',''Nouakchott''},
{''MAURITIUS'',''Port Louis''}, {''MOROCCO'',''Rabat''},
{''MOZAMBIQUE'',''Maputo''}, {''NAMIBIA'',''Windhoek''},
{''NIGER'',''Niamey''}, {''NIGERIA'',''Abuja''},
{''RWANDA'',''Kigali''},
{''SAO TOME E PRINCIPE'',''Sao Tome''},
{''SENEGAL'',''Dakar''}, {''SEYCHELLES'',''Victoria''},
{''SIERRA LEONE'',''Freetown''}, {''SOMALIA'',''Mogadishu''},
{''SOUTH AFRICA'',''Pretoria/Cape Town''},
{''SUDAN'',''Khartoum''},
{''SWAZILAND'',''Mbabane''}, {''TANZANIA'',''Dodoma''},
{''TOGO'',''Lome''}, {''TUNISIA'',''Tunis''},
{''UGANDA'',''Kampala''},
{''DEMOCRATIC REPUBLIC OF THE CONGO (ZAIRE)'',
 ''Kinshasa''},
{''ZAMBIA'',''Lusaka''}, {''ZIMBABWE'',''Harare''},
// Asia
{''AFGHANISTAN'',''Kabul''}, {''BAHRAIN'',''Manama''},
{''BANGLADESH'',''Dhaka''}, {''BHUTAN'',''Thimphu''},
{''BRUNEI'',''Bandar Seri Begawan''},
{''CAMBODIA'',''Phnom Penh''},
{''CHINA'',''Beijing''}, {''CYPRUS'',''Nicosia''},
{''INDIA'',''New Delhi''}, {''INDONESIA'',''Jakarta''},
{''IRAN'',''Tehran''}, {''IRAQ'',''Baghdad''},
{''ISRAEL'',''Jerusalem''}, {''JAPAN'',''Tokyo''},
{''JORDAN'',''Amman''}, {''KUWAIT'',''Kuwait City''},
{''LAOS'',''Vientiane''}, {''LEBANON'',''Beirut''},
{''MALAYSIA'',''Kuala Lumpur''}, {''THE MALDIVES'',''Male''},
{''MONGOLIA'',''Ulan Bator''},
{''MYANMAR (BURMA)'',''Rangoon''},
{''NEPAL'',''Katmandu''}, {''NORTH KOREA'',''P'yongyang''},
{''OMAN'',''Muscat''}, {''PAKISTAN'',''Islamabad''},
{''PHILIPPINES'',''Manila''}, {''QATAR'',''Doha''},
{''SAUDI ARABIA'',''Riyadh''}, {''SINGAPORE'',''Singapore''},
{''SOUTH KOREA'',''Seoul''}, {''SRI LANKA'',''Colombo''},
{''SYRIA'',''Damascus''},
{''TAIWAN (REPUBLIC OF CHINA)'',''Taipei''},
{''THAILAND'',''Bangkok''}, {''TURKEY'',''Ankara''},
{''UNITED ARAB EMIRATES'',''Abu Dhabi''},
```

```
{''VIETNAM'',''Hanoi''}, {''YEMEN'',''Sana'a''},
// Australia and Oceania
{''AUSTRALIA'',''Canberra''}, {''FIJI'',''Suva''},
{''KIRIBATI'',''Bairiki''},
{''MARSHALL ISLANDS'',''Dalap-Uliga-Darrit''},
{''MICRONESIA'',''Palikir''}, {''NAURU'',''Yaren''},
{''NEW ZEALAND'',''Wellington''}, {''PALAU'',''Koror''},
{''PAPUA NEW GUINEA'',''Port Moresby''},
{''SOLOMON ISLANDS'',''Honaira''}, {''TONGA'',''Nuku'alofa''},
{''TUVALU'',''Fongafale''}, {''VANUATU'',''< Port-Vila''},
{''WESTERN SAMOA'',''Apia''},
// Eastern Europe and former USSR
{''ARMENIA'',''Yerevan''}, {''AZERBAIJAN'',''Baku''},
{''BELARUS (BYELORUSSIA)'',''Minsk''},
{''BULGARIA'',''Sofia''}, {''GEORGIA'',''Tbilisi''},
{''KAZAKSTAN'',''Almaty''}, {''KYRGYZSTAN'',''Alma-Ata''},
{''MOLDOVA'',''Chisinau''}, {''RUSSIA'',''Moscow''},
{''TAJIKISTAN'',''Dushanbe''}, {''TURKMENISTAN'',''Ashkabad''},
{''UKRAINE'',''Kyiv''}, {''UZBEKISTAN'',''Tashkent''},
// Europe
{''ALBANIA'',''Tirana''}, {''ANDORRA'',''Andorra la Vella''},
{''AUSTRIA'',''Vienna''}, {''BELGIUM'',''Brussels''},
{''BOSNIA'',''-''}, {''HERZEGOVINA'',''Sarajevo''},
{''CROATIA'',''Zagreb''}, {''CZECH REPUBLIC'',''Prague''},
{''DENMARK'',''Copenhagen''}, {''ESTONIA'',''Tallinn''},
{''FINLAND'',''Helsinki''}, {''FRANCE'',''Paris''},
{''GERMANY'',''Berlin''}, {''GREECE'',''Athens''},
{''HUNGARY'',''Budapest''}, {''ICELAND'',''Reykjavik''},
{''IRELAND'',''Dublin''}, {''ITALY'',''Rome''},
{''LATVIA'',''Riga''}, {''LIECHTENSTEIN'',''Vaduz''},
{''LITHUANIA'',''Vilnius''}, {''LUXEMBOURG'',''Luxembourg''},
{''MACEDONIA'',''Skopje''}, {''MALTA'',''Valletta''},
{''MONACO'',''Monaco''}, {''MONTENEGRO'',''Podgorica''},
{''THE NETHERLANDS'',''Amsterdam''}, {''NORWAY'',''Oslo''},
{''POLAND'',''Warsaw''}, {''PORTUGAL'',''Lisbon''},
{''ROMANIA'',''Bucharest''}, {''SAN MARINO'',''San Marino''},
{''SERBIA'',''Belgrade''}, {''SLOVAKIA'',''Bratislava''},
{''SLOVENIA'',''Ljuijana''}, {''SPAIN'',''Madrid''},
{''SWEDEN'',''Stockholm''}, {''SWITZERLAND'',''Berne''},
{''UNITED KINGDOM'',''London''}, {''VATICAN CITY'',''--''},
// North and Central America
{''ANTIGUA AND BARBUDA'',''Saint John's''},
{''BAHAMAS'',''Nassau''},
{''BARBADOS'',''Bridgetown''}, {''BELIZE'',''Belmopan''},
{''CANADA'',''Ottawa''}, {''COSTA RICA'',''San Jose''},
{''CUBA'',''Havana''}, {''DOMINICA'',''Roseau''},
{''DOMINICAN REPUBLIC'',''Santo Domingo''},
{''EL SALVADOR'',''San Salvador''},
{''GRENADA'',''Saint George's''},
{''GUATEMALA'',''Guatemala City''},
{''HAITI'',''Port-au-Prince''},
{''HONDURAS'',''Tegucigalpa''}, {''JAMAICA'',''Kingston''},
{''MEXICO'',''Mexico City''}, {''NICARAGUA'',''Managua''},
{''PANAMA'',''Panama City''}, {''ST. KITTS'',''-''},
{''NEVIS'',''Basseterre''}, {''ST. LUCIA'',''Castries''},
{''ST. VINCENT AND THE GRENADINES'',''Kingstown''},
{''UNITED STATES OF AMERICA'',''Washington, D.C.''},
```

```
        // South America
        {''ARGENTINA'',''Buenos Aires''},
        {''BOLIVIA'',''Sucre (legal)/La Paz(administrative)''},
        {''BRAZIL'',''Brasilia''}, {''CHILE'',''Santiago''},
        {''COLOMBIA'',''Bogota''}, {''ECUADOR'',''Quito''},
        {''GUYANA'',''Georgetown''}, {''PARAGUAY'',''Asuncion''},
        {''PERU'',''Lima''}, {''SURINAME'',''Paramaribo''},
        {''TRINIDAD AND TOBAGO'',''Port of Spain''},
        {''URUGUAY'',''Montevideo''}, {''VENEZUELA'',''Caracas''},
    };
    // Use AbstractMap by implementing entrySet()
    private static class FlyweightMap extends AbstractMap<String,String> {
        private static class Entry implements Map.Entry<String,String> {
            int index;
            Entry(int index) { this.index = index; }
            public boolean equals(Object o) {
                return DATA[index][0].equals(o);
            }
            public String getKey() { return DATA[index][0]; }
            public String getValue() { return DATA[index][1]; }
            public String setValue(String value) {
                throw new UnsupportedOperationException();
            }
            public int hashCode() {
                return DATA[index][0].hashCode();
            }
        }
        // Use AbstractSet by implementing size() & iterator()
        static class EntrySet extends AbstractSet<Map.Entry<String,String>> {
            private int size;
            EntrySet(int size) {
                if(size < 0)
                    this.size = 0;
                // Can't be any bigger than the array:
                else if(size > DATA.length)
                    this.size = DATA.length;
                else
                    this.size = size;
            }
            public int size() { return size; }
            private class Iter implements Iterator<Map.Entry<String,String>> {
                // Only one Entry object per Iterator:
                private Entry entry = new Entry(-1);
                public boolean hasNext() {
                    return entry.index < size - 1;
                }
                public Map.Entry<String,String> next() {
                    entry.index++;
                    return entry;
                }
                public void remove() {
                    throw new UnsupportedOperationException();
                }
            }
            public Iterator<Map.Entry<String,String>> iterator() {
                return new Iter();
            }
```

```
        }
        private static Set<Map.Entry<String,String>> entries =
            new EntrySet(DATA.length);
        public Set<Map.Entry<String,String>> entrySet() {
            return entries;
        }
    }
    // Create a partial map of 'size' countries:
    static Map<String,String> select(final int size) {
        return new FlyweightMap() {
                public Set<Map.Entry<String,String>> entrySet() {
                    return new EntrySet(size);
                }
            };
    }
    static Map<String,String> map = new FlyweightMap();
    public static Map<String,String> capitals() {
        return map; // The entire map
    }
    public static Map<String,String> capitals(int size) {
        return select(size); // A partial map
    }
    static List<String> names = new ArrayList<String>(map.keySet());
    // All the names:
    public static List<String> names() { return names; }
    // A partial list:
    public static List<String> names(int size) {
        return new ArrayList<String>(select(size).keySet());
    }
    public static void main(String[] args) {
        print(capitals(10));
        print(names(10));
        print(new HashMap<String,String>(capitals(3)));
        print(new LinkedHashMap<String,String>(capitals(3)));
        print(new TreeMap<String,String>(capitals(3)));
        print(new Hashtable<String,String>(capitals(3)));
        print(new HashSet<String>(names(6)));
        print(new LinkedHashSet<String>(names(6)));
        print(new TreeSet<String>(names(6)));
        print(new ArrayList<String>(names(6)));
        print(new LinkedList<String>(names(6)));
        print(capitals().get("BRAZIL"));
    }
} /* Output:
      {ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo, BOTSWANA=Gaberone,
        BULGARIA=Sofia, BURKINA FASO=Ouagadougou, BURUNDI=Bujumbura,
        CAMEROON=Yaounde, CAPE VERDE=Praia, CENTRAL AFRICAN REPUBLIC=Bangui}
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, BURUNDI,
        CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC]
      {BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
      {ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
      {ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
      {ALGERIA=Algiers, ANGOLA=Luanda, BENIN=Porto-Novo}
      [BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
```

```
        [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
        Brasilia
    *///:~
```

Das zweidimensionale `String`-Array `DATA` ist öffentlich, also überall sichtbarbar. Die Klasse `FlyweightMap` muß die `entrySet()`-Methode implementieren, die wiederum individuelle *`Set`*- und *`Map.Entry`*-Implementierungen erforderlich macht. Hier liegt ein Teil des *Flyweight*-Entwurfsmusters: Jedes `Entry`-Objekt speichert nur seinen Index, statt des eigentlichen Schlüssel/Wert-Paares. Seine Methoden `getKey()` und `getValue()` verwenden den Index, um den Schlüssel beziehungsweise Wert des entsprechenden Schlüssel/Wert-Paares zurückzugeben. Die innere Klasse `EntrySet` sorgt dafür, daß ihr `size`-Feld nicht größer wird, als Anzahl der Elemente des von `DATA` referenzierten Containers.

[20] Der andere Teil des Entwurfsmusters ist in der inneren Klasse `Iter` implementiert. Statt für jedes Schlüssel/Wert-Paar in `DATA` ein `Entry`-Objekt zu erzeugen, gibt es nur ein `Entry`-Objekt *pro Iterator*. Das `Entry`-Objekt funktioniert wie ein Fenster auf die Daten: Es enthält lediglich den Index bezüglich des statischen `DATA`-Arrays. Bei jedem Aufruf der `next()`-Methode des Iterators wird das `index`-Feld des `Entry`-Objektes inkrementiert, so daß der Iterator auf das nächste Schlüssel/Wert-Paar zeigt. Anschließend gibt der Iterator dieses Schlüssel/Wert-Paar zurück.[3]

[21] Die `select()`-Methode gibt ein `FlyweightMap`-Objekt zurück, welches ein `EntrySet`-Objekt der gewünschten Größe enthält. Letzteres wird von den überladenen Methoden `capitals()` und `names()` verwendet (siehe `main()`).

[22] Bei manchen Tests ist die beschränkte/unveränderliche Anzahl von Elementen des `DATA`-Arrays in `Countries` ein Problem. Derselbe Ansatz liefert aber auch initialisierte individuelle Container mit Datensätzen beliebiger Länge. Die folgende Klasse implementiert einen individuellen *`List`*-Container beliebiger Größe, der mit `Integer`-Elementen vorinitialisiert ist:

```
//: net/mindview/util/CountingIntegerList.java
// List of any length, containing sample data.
package net.mindview.util;
import java.util.*;

public class CountingIntegerList extends AbstractList<Integer> {
    private int size;
    public CountingIntegerList(int size) {
        this.size = size < 0 ? 0 : size;
    }
    public Integer get(int index) {
        return Integer.valueOf(index);
    }
    public int size() { return size; }
    public static void main(String[] args) {
        System.out.println(new CountingIntegerList(30));
    }
} /* Output:
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
21, 22, 23, 24, 25, 26, 27, 28, 29]
   *///:~
```

Wenn Sie eine nur-lesbare Containerklasse von `AbstractList` ableiten, müssen Sie die Methoden `get()` und `size()` implementieren. Die Aufgabe wird wiederum mit Hilfe des *Flyweight*-Entwurfsmusters gelöst: Die `get()`-Methode erzeugt bei Bedarf einen Wert, so daß der *`List`*-Container keine Elemente zu enthalten braucht.

[3] Die Container vom Typ *`Map`* im Package `java.util` kopieren Inhalt gleichartiger Container lose, das heißt mit einzelnen `getKey()`- und `getValue()`-Aufrufen. Daher funktioniert diese Vorgehensweise. Bei einer individuellen *`Map`*-Implementierung, die einfach das gesamte Schlüssel/Wert-Paar kopiert, wäre dieser Ansatz problematisch.

[23] Die nächste Klasse implementiert einen individuellen *Map*-Container beliebiger Größe, der eindeutige `Integer`-Objekte auf `String`-Objekt abbildet:

```
//: net/mindview/util/CountingMapData.java
// Unlimited-length Map containing sample data.
package net.mindview.util;
import java.util.*;

public class CountingMapData extends AbstractMap<Integer,String> {
    private int size;
    private static String[] chars =
        ''A B C D E F G H I J K L M N O P Q R S T U V W X Y Z''.split('' '');
    public CountingMapData(int size) {
        if(size < 0) this.size = 0;
        this.size = size;
    }
    private static class Entry implements Map.Entry<Integer,String> {
        int index;
        Entry(int index) { this.index = index; }
        public boolean equals(Object o) {
            return Integer.valueOf(index).equals(o);
        }
        public Integer getKey() { return index; }
        public String getValue() {
            return
                chars[index % chars.length] +
                Integer.toString(index / chars.length);
        }
        public String setValue(String value) {
            throw new UnsupportedOperationException();
        }
        public int hashCode() {
            return Integer.valueOf(index).hashCode();
        }
    }
    public Set<Map.Entry<Integer,String>> entrySet() {
        // LinkedHashSet retains initialization order:
        Set<Map.Entry<Integer,String>> entries =
            new LinkedHashSet<Map.Entry<Integer,String>>();
        for(int i = 0; i < size; i++)
            entries.add(new Entry(i));
        return entries;
    }
    public static void main(String[] args) {
        System.out.println(new CountingMapData(60));
    }
} /* Output:
      {0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0, 10=K0, 11=L0,
12=M0, 13=N0, 14=O0, 15=P0, 16=Q0, 17=R0, 18=S0, 19=T0, 20=U0, 21=V0, 22=W0, 23=X0,
24=Y0, 25=Z0, 26=A1, 27=B1, 28=C1, 29=D1, 30=E1, 31=F1, 32=G1, 33=H1, 34=I1, 35=J1,
36=K1, 37=L1, 38=M1, 39=N1, 40=O1, 41=P1, 42=Q1, 43=R1, 44=S1, 45=T1, 46=U1, 47=V1,
48=W1, 49=X1, 50=Y1, 51=Z1, 52=A2, 53=B2, 54=C2, 55=D2, 56=E2, 57=F2, 58=G2, 59=H2}
   *///:~
```

Das Beispiel verwendet einen Container vom Typ `LinkedHashSet`, statt eine individuelle *Set*-Implementierung zu liefern, implementiert das *Flyweight*-Entwurfsmuster also nicht vollständig.

Übungsaufgabe 1: (1) Legen Sie einen *List*-Container an (versuchen Sie sowohl `ArrayList` als

auch `LinkedList`) und füllen Sie ihn mit Hilfe der Klasse `Countries`. Sortieren Sie den Inhalt des Containers und geben Sie ihn aus. Rufen Sie mehrmals die Methode `Collections.shuffle()` auf und geben Sie jedes Mal den Containerinhalt aus, um zu beobachten, wie `shuffle()` die Reihenfolge der Elemente randomisiert. ■

Übungsaufgabe 2: (2) Legen Sie einen *`Map`*- und einen *`Set`*-Container an, die alle Ländernamen enthalten, die mit „A“ beginnen. ■

Übungsaufgabe 3: (1) Verwenden Sie die Klasse `Countries`, um einen *`Set`*-Container mehrmals mit denselben Daten zu füllen und verifizieren Sie, daß der Container nur ein Exemplar jedes Elementes speichert. Führen Sie diesen Versuch mit den Containertypen `HashSet`, `LinkedHashSet` und `TreeSet` durch. ■

Übungsaufgabe 4: (2) Schreiben Sie eine Initialisierungsklasse für Container vom Typ *`Collection`*, die eine per `TextFile` eingelesene Textdatei in einzelne Wörter auftrennt und diese als Initialisierungsdaten des Containers verwendet. Zeigen Sie, daß Ihre Lösung funktioniert. ■

Übungsaufgabe 5: (3) Ändern Sie das Beispiel *CountingMapData.java* zu einer vollständigen Implementierung des *Flyweight*-Entwurfsmusters, indem Sie eine individuelle `EntrySet`-Klasse anlegen (orientieren Sie sich am Beispiel *Countries.java*). ■

18.3 Das Interface Collection

[24] Die folgende Liste zeigt alle im Interface *`Collection`* deklarierten Methoden (ohne die Methoden der Klasse `Object`, die jeder Containerklasse automatisch zur Verfügung stehen), insbesondere also auch den Funktionsumfang der Containertypen *`Set`* und *`List`* (*`List`* deklariert noch zusätzliche Funktionalität). Der Containertyp *`Map`* ist keine Erweiterung von *`Collection`* und wird in Abschnitt 18.8 separat behandelt:

- `boolean add(T)` garantiert, daß der Container das Argument des generischen Typs `T` enthält. Die Methode gibt `false` zurück, wenn das Argument nicht hinzugefügt wurde. („Optionale„ Methode, siehe Abschnitt 18.4).
- `boolean addAll(Collection<? extends T>)` setzt alle als Argument übergebenen Elemente in den Container ein. Die Methode gibt `true` zurück, wenn wenigstens ein Element (neu) hinzugefügt wurde. („Optionale„ Methode)
- `void clear()` entfernt sämtliche Elemente aus dem Container. („Optionale„ Methode)
- `boolean contains(T)` gibt `true` zurück, wenn der Container das Argument vom Typ `T` enthält.
- `Boolean containsAll(Collection<?>)` gibt `true` zurück, wenn der Container alle als Argument übergebenen Elemente enthält.
- `boolean isEmpty()` gibt `true` zurück, wenn der Container keine Elemente enthält.
- *`Iterator`*<T> `iterator()` gibt einen Iterator über Elementen vom Typ `T` zurück (*`Iterator<T>`*), mit dem Sie den Container elementweise durchlaufen können.
- `Boolean remove(Object)` entfernt ein Exemplar aus dem Container, falls dieser das als Argument übergebene Element enthält. Die Methode gibt `true` zurück, wenn ein Element entfernt wurde. („Optionale“ Methode)

- `boolean removeAll(Collection<?>)` entfernt alle als Argument übergebenen Elemente aus dem Container. Die Methode gibt `true` zurück, wenn ein Element entfernt wurde. („Optionale“ Methode)
- `Boolean retainAll(Collection<?>)` beläßt nur die als Argument übergebenen Elemente im Container (die Schnittmenge, im Sinne der Mengenlehre). Die Methode gibt `true` zurück, wenn der Container verändert wurde. („Optionale“ Methode)
- `int size()` gibt die Anzahl der Elemente des Containers zurück.
- `Object toArray()` gibt ein Array zurück, welches alle Elemente des Containers enthält.
- `<T> T[] toArray(T[] a)` gibt ein Array zurück, welches alle Elemente des Containers enthält. Der Laufzeittyp des zurückgegebenen Arrays ist der Typ des als Argument übergebenen Arrays, nicht `Object`.

[25] Beachten Sie, daß das Interface *`Collection`* keine `get()`-Methode für den Zugriff auf beliebige Elemente deklariert, da *`Collection`* auch das Interface *`Set`* umfaßt, welches seinen Elementen eine eigene Ordnung aufprägt, so daß der Zugriff auf beliebige Elemente keine Bedeutung mehr hat. Das Verarbeiten der Elemente eines Containers vom Typ *`Collection`* setzt also einen Iterator voraus.

[26] Das nächste Beispiel führt alle diese Methoden vor. Obwohl diese Methoden bei jedem Container vom Typ *`Collection`* zur Verfügung stehen, wurde der Typ `ArrayList` als „gemeinsamer Nenner“ gewählt:

```
//: containers/CollectionMethods.java
// Things you can do with all Collections.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class CollectionMethods {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();
        c.addAll(Countries.names(6));
        c.add("ten");
        c.add("eleven");
        print(c);
        // Make an array from the List:
        Object[] array = c.toArray();
        // Make a String array from the List:
        String[] str = c.toArray(new String[0]);
        // Find max and min elements; this means
        // different things depending on the way
        // the Comparable interface is implemented:
        print("Collections.max(c) = " + Collections.max(c));
        print("Collections.min(c) = " + Collections.min(c));
        // Add a Collection to another Collection
        Collection<String> c2 = new ArrayList<String>();
        c2.addAll(Countries.names(6));
        c.addAll(c2);
        print(c);
        c.remove(Countries.DATA[0][0]);
        print(c);
        c.remove(Countries.DATA[1][0]);
        print(c);
        // Remove all components that are
        // in the argument collection:
```

```
        c.removeAll(c2);
        print(c);
        c.addAll(c2);
        print(c);
        // Is an element in this Collection?
        String val = Countries.DATA[3][0];
        print("c.contains(" + val + ") = " + c.contains(val));
        // Is a Collection in this Collection?
        print("c.containsAll(c2) = " + c.containsAll(c2));
        Collection<String> c3 =
            ((List<String>)c).subList(3, 5);
        // Keep all the elements that are in both
        // c2 and c3 (an intersection of sets):
        c2.retainAll(c3);
        print(c2);
        // Throw away all the elements
        // in c2 that also appear in c3:
        c2.removeAll(c3);
        print("c2.isEmpty() = " + c2.isEmpty());
        c = new ArrayList<String>();
        c.addAll(Countries.names(6));
        print(c);
        c.clear(); // Remove all elements
        print("after c.clear():" + c);
    }
} /* Output:
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven]
      Collections.max(c) = ten
      Collections.min(c) = ALGERIA
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven,
      ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      [ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven, ALGERIA,
      ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      [BENIN, BOTSWANA, BULGARIA, BURKINA FASO, ten, eleven, ALGERIA, ANGOLA,
      BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      [ten, eleven]
      [ten, eleven, ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      c.contains(BOTSWANA) = true
      c.containsAll(c2) = true
      [ANGOLA, BENIN]
      c2.isEmpty() = true
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      after c.clear():[]
   *///:~
```

Das Beispiel erzeugt mehrere `ArrayList`-Container, die verschiedene Daten enthalten und deren Referenzen aufwärts in den Typ *`Collection`* umgewandelt werden. Somit ist klar, daß keine andere Schnittstelle, als das Interface *`Collection`* verwendet wird. Die `main()`-Methode zeigt alle in *`Collection`* deklarierten Methoden anhand einfacher Beispiele.

[27] Die folgenden Abschnitte dieses Kapitels beschreiben die verschiedenen Implementierungen der Interfaces *`List`*, *`Set`* und *`Map`* und zeigen jeweils durch ein Sternchen (*) an, welche Variante Sie als Standardimplementierung wählen sollten. Die Beschreibung der veralteten Klassen `Vector`, `Stack` und `Hashtable` finden Sie in Abschnitt 18.13, am Schluß des Kapitels. Sie sollten diese Klassen nicht verwenden, finden Sie aber noch in älteren Quelltexten.

18.4 Optionale Methoden

[28] Die Methoden des Interfaces *`Collection`*, die auf unterschiedliche Art und Weise Elemente hinzufügen oder entfernen, sind sogenannte *optionale Methoden*, das heißt eine implementierende Klasse muß keine *sinnvolle* Definition einer solchen Methoden liefern.

[29] Dies ist ein ungewöhnliches Verfahren, um ein Interface zu definieren. Sie haben gelernt, daß ein Interface in einem objektorientierten Design ein Vertrag ist. Dieser Vertrag garantiert, unabhängig von der gewählten Implementierung des Interfaces, daß die deklarierten Methoden aufgerufen („einem Objekt dieses Typs entsprechende Nachrichten gesendet") werden können.[4] Eine „optionale Methode" verletzt dieses fundamentale Prinzip, da ein solche Methode *kein sinnvolles Verhalten definiert*, sondern eine Ausnahme hervorruft. Scheinbar ist hier keine Typsicherheit zur Übersetzungszeit gegeben.

[30] Es ist nicht ganz so schlimm. Auch bei einer optionalen Methode erlaubt Ihnen der Compiler, nur die in diesem Interface deklarierten Methoden aufzurufen. Es ist nicht wie bei einer dynamischen Sprache, bei der Sie auf einem beliebigen Objekt eine beliebige Methode aufrufen können und erst zur Laufzeit erfahren, ob sich eine bestimmte Methode aufrufen läßt.[5] Die meisten Methoden, die ein Argument vom Typ *`Collection`* erwarten, fragen den übergebenen Container außerdem nur ab und die Abfragemethoden im Interface *`Collection`* sind keine optionalen Methoden.

[31] Aus welchem Grund könnten Sie eine Methode als „optional" definieren? Dieser Ansatz verhindert eine explosionsartige Vermehrung der Interfaces im Design der Containerbibliothek. Andere Designansätze führen stets zu einer verwirrenden Vielzahl von Interfaces, um sämtliche Ausprägungen des eigentlichen Gegenstandes zu beschreiben. Es ist nicht einmal möglich, jeden Spezialfall in Form eines Interfaces zu erfassen, da jederzeit jemand ein neues Interface erfinden kann. Der Ansatz der „nicht unterstützten Methode" (*unsupported operation*) verwirklicht eine wichtige Motivation der Containerbibliothek von Java, nämlich daß der Umgang mit den Containern leicht zu erlernen und zu praktizieren ist. Die folgenden beiden Voraussetzungen müssen erfüllt sein, damit dieser Designansatz funktioniert:

- Ausnahmen vom Typ `UnsupportedOperationException` dürfen nur selten vorkommen, das heißt bei den meisten Klassen sollten alle Methoden sinnvoll definiert und nur in Spezialfällen eine Methode nicht unterstützt sein. Dieses Kriterium ist bei der Containerbibliothek von Java erfüllt: Die Klassen, welche Sie während 99% Ihrer Zeit verwenden (`ArrayList`, `LinkedList`, `HashSet` und `HashMap`) unterstützten alle deklarierten Methoden. Dieser Designansatz gestattet eine „Hintertür" für den Fall, daß Sie eine neue Containerklasse unter dem Interface *`Collection`* entwickeln, dabei aber nicht für jede in diesem Interface deklarierte Methode eine sinnvolle Definition schreiben, Ihre neue Klasse aber dennoch in die existierende Containerbibliothek eingliedern möchten.

- Wird eine Methode nicht unterstützt, so sollte eine vernünftige Aussicht bestehen, daß die zugehörige Ausnahme vom Typ `UnsupportedOperationException` auftritt, während Sie noch an dem Programm arbeiten und nicht erst nachdem Sie Ihrem Kunden das fertige Produkt übergeben haben. Schließlich zeigt eine solche Ausnahme einen Programmierfehler an, nämlich daß Sie die Implementierung eines Interfaces falsch verwenden.

[32] Beachten Sie, daß nicht unterstützte Methoden nur zur Laufzeit erkannt werden können, also unter die dynamische Typprüfung fallen. Falls Sie von einer statisch geprüften Sprache wie C++

[4] Ich verwende den Begriff „Interface" an dieser Stelle sowohl für die formale Definition mit dem Schlüsselwort `interface` als auch in der allgemeineren Bedeutung „der von einer Klasse oder Unterklasse unterstützten Methoden".

[5] Sie haben in Kapitel 15 gesehen, wie mächtig derartiges dynamisches Verhalten sein kann, obwohl es in der hier beschriebene Weise sonderbar und vielleicht sogar nutzlos klingt.

herkommen, scheint Java nur eine weitere statisch geprüfte Sprache zu sein. Java *hat* statische Typprüfungen, aber auch einen deutlichen Anteil von dynamischen Typprüfungen, so daß sich schwer sagen läßt, ob Java der einen oder anderen Art von Programmiersprache angehört. Wenn Sie beginnen, dies zu erkennen, werden Sie anfangen, weitere Beispiele für dynamische Typprüfungen in Java zu bemerken.

18.4.1 Nicht unterstützte Methoden

[33] Container die sich auf eine Datenstruktur fester Größe beziehen, liefern häufig Beispiele für nicht unterstützte Methoden. Sie erhalten einen solchen Container etwa, indem Sie der statischen `Arrays`-Methode `asList()` ein Array übergeben. Außerdem können Sie einen beliebigen Container (inklusive *`Map`*) durch Umwandlung mit Hilfe der statischen `unmodifiableXXX()`-Methoden in der Klasse `Collections` veranlassen, bei Aufrufen von Methoden, die den Zustand des Containers ändern, eine Ausnahme vom Typ `UnsupportedOperationException` auszuwerfen:

```
//: containers/Unsupported.java
// Unsupported operations in Java containers.
import java.util.*;

public class Unsupported {
    static void test(String msg, List<String> list) {
        System.out.println("-- " + msg + " --");
        Collection<String> c = list;
        Collection<String> subList = list.subList(1,8);
        // Copy of the sublist:
        Collection<String> c2 = new ArrayList<String>(subList);
        try { c.retainAll(c2); } catch(Exception e) {
            System.out.println("retainAll(): " + e);
        }
        try { c.removeAll(c2); } catch(Exception e) {
            System.out.println("removeAll(): " + e);
        }
        try { c.clear(); } catch(Exception e) {
            System.out.println("clear(): " + e);
        }
        try { c.add("X"); } catch(Exception e) {
            System.out.println("add(): " + e);
        }
        try { c.addAll(c2); } catch(Exception e) {
            System.out.println("addAll(): " + e);
        }
        try { c.remove("C"); } catch(Exception e) {
            System.out.println("remove(): " + e);
        }
        // The List.set() method modifies the value but
        // doesn't change the size of the data structure:
        try {
            list.set(0, "X");
        } catch(Exception e) {
            System.out.println("List.set(): " + e);
        }
    }
    public static void main(String[] args) {
        List<String> list =
            Arrays.asList("A B C D E F G H I J K L".split(" "));
        test("Modifiable Copy", new ArrayList<String>(list));
```

```
        test("Arrays.asList()", list);
        test("unmodifiableList()",
             Collections.unmodifiableList(new ArrayList<String>(list)));
    }
} /* Output:
      -- Modifiable Copy --
      -- Arrays.asList() --
      retainAll(): java.lang.UnsupportedOperationException
      removeAll(): java.lang.UnsupportedOperationException
      clear(): java.lang.UnsupportedOperationException
      add(): java.lang.UnsupportedOperationException
      addAll(): java.lang.UnsupportedOperationException
      remove(): java.lang.UnsupportedOperationException
      -- unmodifiableList() --
      retainAll(): java.lang.UnsupportedOperationException
      removeAll(): java.lang.UnsupportedOperationException
      clear(): java.lang.UnsupportedOperationException
      add(): java.lang.UnsupportedOperationException
      addAll(): java.lang.UnsupportedOperationException
      remove(): java.lang.UnsupportedOperationException
      List.set(): java.lang.UnsupportedOperationException
   *///:~
```

Da die statische `Arrays`-Methode `asList()` einen *`List`*-Container zurückgibt, der sich auf ein Array fester Länge stützt, ist es sinnvoll, nur solche Methoden zu unterstützen, die die Länge des Arrays nicht verändern. Jede Methode, die eine Änderung der Kapazität der unterliegenden Datenstruktur bewirken würde, ruft eine Ausnahme vom Typ `UnsupportedOperationException` hervor, um den Aufruf einer nicht unterstützten Methode (also eines Programmierfehlers) anzuzeigen.

[34] Beachten Sie, daß Sie den von `Arrays.asList()` zurückgegebenen Container dem Konstruktor jeder Containerklasse vom Typ *`Collection`*, der `addAll()`-Methode jedes Containers von diesem Typ oder der gleichnamigen statischen Methode der Klasse *`Collection`* übergeben können, um einen gewöhnlichen Container zu erzeugen, der das Aufrufen sämtlicher Methoden gestattet (siehe ersten Aufruf der `test()`-Methode in `main()`). Das Aufrufen einer solchen Methode liefert eine neue unterliegende Datenstruktur mit veränderbarer Kapazität.

[35] Die statischen `unmodifiableXXX()`-Methoden in der Klasse `Collections` verpacken den übergebenen Container in einem „Stellvertreterobjekt", welches bei jedem Methodenaufruf, der den Zustand des Containers auf irgendeine Weise verändern würde, eine Ausnahme vom Typ `UnsupportedOperationException` auswirft. Das Ziel der Anwendung der `unmodifiableXXX()`-Methoden besteht darin, einen „konstanten Container" zu bekommen. Das Beispiel in Unterabschnitt 18.11.2 führt die einzelnen `unmodifiableXXX()`-Methoden der Klasse `Collections` vor.

[36] Die letzte `try`-Klausel in der `test()`-Methode ruft die im Interface *`List`* deklarierte Methode `get()` auf. Hier können Sie beobachten, wie die Granularität durch nicht unterstützte Methoden gelegen kommt: Die Schnittstellen des von `Arrays.asList()` beziehungsweise `Collections.unmodifiableList()` zurückgegebenen Containers unterscheiden sich durch eine Methode. `Arrays.asList()` gibt einen *`List`*-Container mit fester Kapazität zurück, während `Collections.unmodifiableList()` einen *`List`*-Container liefert, dessen Zustand nicht verändert werden kann. Wie Sie an der Ausgabe nachvollziehen können, ist es in Ordnung, eines oder mehrere Elemente in dem von `Arrays.asList()` zurückgegebenen *`List`*-Container zu ändern, da hierbei die feste Kapazität des Containers nicht verletzt wird. Andererseits darf der von `unmodifiableList()` zurückgegebene *`List`*-Container offensichtlich keinerlei Änderung zulassen. Bei einem anderen Designansatz wären zwei weitere Interfaces erforderlich, nämlich eines mit einer funktionstüchtigen `set()`-Methode und eines ohne. Die verschiedenen unveränderlichen Containertypen unter dem In-

terface *Collection* würden ebenfalls zusätzliche Interfaces verlangen.

[37] Die Dokumentation einer Methode, die einen Container als Argument erwartet, sollte darauf hinweisen, welche optionalen Methoden implementiert sein müssen.

Übungsaufgabe 6: (2) Beachten Sie, daß das Interface *List* zusätzliche „optionale“ Methoden deklariert, die nicht in *Collection* enthalten sind. Schreiben Sie eine Variante des Beispiels *Unsupported.java*, welche diese zusätzlichen optionalen Methoden testet. ■

18.5 Das Interface List

[38] Die grundlegende Funktionalität eines Containers vom Typ *List* ist einfach zu verwenden: In der Regel rufen Sie die `add()`-Methode auf, um Elemente in den Container einzusetzen, `get()`, um einzelne Objekte abzufragen und `iterator()`, um einen Iterator über die gespeicherten Elemente anzufordern.

[39] Im folgenden Beispiel deckt jede Methode eine andere Gruppe von Operationen ab: Grundlegende Funktionalität jedes Containers vom Typ *List* (`basicTest()`), Bewegung über den Elementen eines Containers per Iterator (`iterMotion()`), Änderungen an Elementen per Iterator (`iterManipulation()`), Anzeigen der Auswirkungen von Änderungsoperationen (`testVisual()`) und Operationen, die nur dem Containertyp `LinkedList` zur Verfügung stehen (`testLinkedList()`):

```
//: containers/Lists.java
// Things you can do with Lists.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Lists {
    private static boolean b;
    private static String s;
    private static int i;
    private static Iterator<String> it;
    private static ListIterator<String> lit;
    public static void basicTest(List<String> a) {
        a.add(1, ''x''); // Add at location 1
        a.add(''x''); // Add at end
        // Add a collection:
        a.addAll(Countries.names(25));
        // Add a collection starting at location 3:
        a.addAll(3, Countries.names(25));
        b = a.contains(''1''); // Is it in there?
        // Is the entire collection in there?
        b = a.containsAll(Countries.names(25));
        // Lists allow random access, which is cheap
        // for ArrayList, expensive for LinkedList:
        s = a.get(1); // Get (typed) object at location 1
        i = a.indexOf(''1''); // Tell index of object
        b = a.isEmpty(); // Any elements inside?
        it = a.iterator(); // Ordinary Iterator
        lit = a.listIterator(); // ListIterator
        lit = a.listIterator(3); // Start at loc 3
        i = a.lastIndexOf(''1''); // Last match
        a.remove(1); // Remove location 1
        a.remove(''3''); // Remove this object
```

```
        a.set(1, ''y''); // Set location 1 to ''y''
        // Keep everything that's in the argument
        // (the intersection of the two sets):
        a.retainAll(Countries.names(25));
        // Remove everything that's in the argument:
        a.removeAll(Countries.names(25));
        i = a.size(); // How big is it?
        a.clear(); // Remove all elements
    }
    public static void iterMotion(List<String> a) {
        ListIterator<String> it = a.listIterator();
        b = it.hasNext();
        b = it.hasPrevious();
        s = it.next();
        i = it.nextIndex();
        s = it.previous();
        i = it.previousIndex();
    }
    public static void iterManipulation(List<String> a) {
        ListIterator<String> it = a.listIterator();
        it.add(''47'');
        // Must move to an element after add():
        it.next();
        // Remove the element after the newly produced one:
        it.remove();
        // Must move to an element after remove():
        it.next();
        // Change the element after the deleted one:
        it.set(''47'');
    }
    public static void testVisual(List<String> a) {
        print(a);
        List<String> b = Countries.names(25);
        print(''b = '' + b);
        a.addAll(b);
        a.addAll(b);
        print(a);
        // Insert, remove, and replace elements
        // using a ListIterator:
        ListIterator<String> x = a.listIterator(a.size()/2);
        x.add(''one'');
        print(a);
        print(x.next());
        x.remove();
        print(x.next());
        x.set(''47'');
        print(a);
        // Traverse the list backwards:
        x = a.listIterator(a.size());
        while(x.hasPrevious())
            printnb(x.previous() + '' '');
        print();
        print(''testVisual finished'');
    }
    // There are some things that only LinkedLists can do:
    public static void testLinkedList() {
        LinkedList<String> ll = new LinkedList<String>();
```

```
        ll.addAll(Countries.names(25));
        print(ll);
        // Treat it like a stack, pushing:
        ll.addFirst("one");
        ll.addFirst("two");
        print(ll);
        // Like "peeking" at the top of a stack:
        print(ll.getFirst());
        // Like popping a stack:
        print(ll.removeFirst());
        print(ll.removeFirst());
        // Treat it like a queue, pulling elements
        // off the tail end:
        print(ll.removeLast());
        print(ll);
    }
    public static void main(String[] args) {
        // Make and fill a new list each time:
        basicTest(new LinkedList<String>(Countries.names(25)));
        basicTest(new ArrayList<String>(Countries.names(25)));
        iterMotion(new LinkedList<String>(Countries.names(25)));
        iterMotion(new ArrayList<String>(Countries.names(25)));
        iterManipulation(new LinkedList<String>(Countries.names(25)));
        iterManipulation(new ArrayList<String>(Countries.names(25)));
        testVisual(new LinkedList<String>(Countries.names(25)));
        testLinkedList();
    }
} /* (Execute to see output) *////:~
```

Die Methoden `basicTest()` und `iterMotion()` rufen die *`List`*-Methoden nur auf, um die richtige Syntax zu zeigen. Der Rückgabewert wird zwar gespeichert, aber nicht ausgewertet. In einigen Fällen wird auf die Speicherung des Rückgabewertes verzichtet. Lesen Sie die Verwendung dieser Methoden in der API-Dokumentation nach, bevor Sie sie gebrauchen.

Übungsaufgabe 7: (4) Erzeugen Sie je einen Container vom Typ `ArrayList` und `LinkedList` und füllen Sie beide mit Hilfe der statischen Methode `Countries.names()`. Geben Sie den Inhalt beider Container mit Hilfe eines gewöhnlichen Iterators aus. Setzen Sie anschließend per Listeniterator den Inhalt des einen Containers in den anderen ein, wobei Sie an jeder geraden Position ein Element des ersteren Containers einsetzen. Wiederholen Sie das Einsetzen eines Containers in den anderen, wobei Sie am Ende des Containers beginnen und den Iterator rückwärts bewegen. ■

Übungsaufgabe 8: (7) Schreiben Sie eine generische Klasse `SList`, die ein Element einer einfach verketteten Liste repräsentiert, das Interface *`List`* aber nicht implementiert, um die Aufgabe nicht unnötig zu verkomplizieren. Jedes `SList`-Objekt der Liste enthält eine Referenz auf das nächste Element der Liste, nicht aber auf das vorige Element (die Klasse `LinkedList` repräsentiert im Gegensatz dazu eine doppelt verkettete Liste dar, pflegt also Verweise in beide Richtungen). Schreiben Sie einen eigenen Iterator `SListIterator`, der das Interface *`ListIterator`* nicht zu implementieren braucht. Außer `toString()` sollte die Klasse `SList` nur die `iterator()`-Methode definieren, welche eine Referenz auf ein `SListIterator`-Objekt zurückgibt. Die einzige Möglichkeit, der durch `SList` verkörperten verketteten Liste Elemente hinzuzufügen oder zu entfernen, ist das `SListIterator`-Objekt. Schreiben Sie ein Programm, das die Klasse `SList` vorführt. ■

18.6 Das Interface Set und die Anordnung der gespeicherten Elemente

[40] Die Beispiele für Container vom Typ *Set* in Kapitel 12 sind eine gute Einführung für die grundlegenden Methoden dieses Containertyps, bedienen sich aber vordefinierter Java-Typen wie `Integer` und `String`, die für die Verwendung in Containern entwickelt wurden. Sie müssen sich bei Ihren eigenen Klassen der Tatsache bewußt sein, daß ein *Set*-Container die Anordnung der gespeicherten Elemente pflegt. Das Ordnungskriterium variiert von einer Implementierung des *Set*-Interfaces zur anderen. Verschiedene Implementierungen verhalten sich also nicht nur unterschiedlich, sondern stellen auch unterschiedliche Anforderungen an den Typ der unter einem bestimmten Containertyp unter dem Interface *Set* gespeicherten Elemente:

- *Set* (Interface): Jedes Element, das Sie in einen Container vom Typ *Set* einsetzen, muß sich eindeutig von den bereits gespeicherten Elementen unterscheiden. Ein Container vom Typ *Set* enthält keine Duplikate. Die einem Container vom Typ *Set* übergebenen Elemente müssen wenigstens über eine `equals()`-Methode verfügen, um die Eindeutigkeit der Elemente gewährleisten zu können. Die Interfaces *Set* und *Collection* stimmen exakt überein. Das Interface *Set* garantiert nicht, daß die Elemente in einer bestimmten Reihenfolge gespeichert werden.
- `HashSet`*: Dieser Typ eignet sich für *Set*-Container, bei denen schnelles Nachschlagen erforderlich ist. Die Elemente müssen auch die Methode `hashCode()` zur Verfügung haben.
- `TreeSet`: Dieser Typ repräsentiert eine geordnet Menge, die sich auf eine Baumstruktur stützt. Sie können einem Container dieses Typs eine geordnete Reihe von Elementen entnehmen. Die Elemente müssen auch das Interface *Comparable* implementieren.
- `LinkedHashSet`: Dieser Typ bietet die Zugriffsgeschwindigkeit von `HashSet` und erhält zugleich die Einsetzungsreihenfolge der Elemente mit Hilfe einer verketteten Liste. Bei der Iteration über den Inhalt eines solchen Containers erscheinen die Elemente in der Reihenfolge, in der sie eingetragen wurden. Die Elemente müssen auch die Methode `hashCode()` zur Verfügung haben.

Das Sternchen (*) bei `HashSet` bedeutet, daß Sie, falls keine weiteren Rahmenbedingungen existieren, diesen Containertyp wählen sollten, weil er hinsichtlich der Zugriffsgeschwindigkeit optimiert ist.

[41] Das Definieren der Methode `hashCode()` wird in Abschnitt 18.9 beschrieben. Die `equals()`-Methode ist sowohl bei hash- als auch bei baumstrukturartiger Speicherung notwendig, die `hashCode()`-Methode dagegen nur, wenn Objekte der betrachteten Klasse in einem Container vom Typ `HashSet` (was zu erwarten ist, da dieser Typ in der Regel Ihre erste Wahl für eine *Set*-Implementierung sein sollte) oder `LinkedHashSet` gespeichert werden. Im Sinne eines guten Programmierstils sollten Sie stets auch `hashCode()` überschreiben, wenn Sie `equals()` überschreiben.

[42] Das folgende Beispiel demonstriert die Methoden, die definiert sein müssen, damit sich die Objekte einer Klasse mit Erfolg in einem Container vom Typ *Set* speichern lassen:

```
//: containers/TypesForSets.java
// Methods necessary to put your own type in a Set.
import java.util.*;

class SetType {
    int i;
    public SetType(int n) { i = n; }
```

```
    public boolean equals(Object o) {
        return o instanceof SetType && (i == ((SetType) o).i);
    }
    public String toString() { return Integer.toString(i); }
}

class HashType extends SetType {
    public HashType(int n) { super(n); }
    public int hashCode() { return i; }
}

class TreeType extends SetType implements Comparable<TreeType> {
    public TreeType(int n) { super(n); }
    public int compareTo(TreeType arg) {
        return (arg.i < i ? -1 : (arg.i == i ? 0 : 1));
    }
}

public class TypesForSets {
    static <T> Set<T> fill(Set<T> set, Class<T> type) {
        try {
            for(int i = 0; i < 10; i++)
                set.add(type.getConstructor(int.class).newInstance(i));
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        return set;
    }
    static <T> void test(Set<T> set, Class<T> type) {
        fill(set, type);
        fill(set, type); // Try to add duplicates
        fill(set, type);
        System.out.println(set);
    }
    public static void main(String[] args) {
        test(new HashSet<HashType>(), HashType.class);
        test(new LinkedHashSet<HashType>(), HashType.class);
        test(new TreeSet<TreeType>(), TreeType.class);
        // Things that don't work:
        test(new HashSet<SetType>(), SetType.class);
        test(new HashSet<TreeType>(), TreeType.class);
        test(new LinkedHashSet<SetType>(), SetType.class);
        test(new LinkedHashSet<TreeType>(), TreeType.class);
        try {
            test(new TreeSet<SetType>(), SetType.class);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
        try {
            test(new TreeSet<HashType>(), HashType.class);
        } catch(Exception e) {
            System.out.println(e.getMessage());
        }
    }
} /* Output: (Sample)
      [2, 4, 9, 8, 6, 1, 3, 7, 5, 0]
      [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
      [9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
      [9, 9, 7, 5, 1, 2, 6, 3, 0, 7, 2, 4, 4, 7, 9,
```

```
   1, 3, 6, 2, 4, 3, 0, 5, 0, 8, 8, 8, 6, 5, 1]
   [0, 5, 5, 6, 5, 0, 3, 1, 9, 8, 4, 2, 3, 9, 7,
   3, 4, 4, 0, 7, 1, 9, 6, 2, 1, 8, 2, 8, 6, 7]
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4,
   5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4,
   5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
   java.lang.ClassCastException: SetType cannot be cast to java.lang.Comparable
   java.lang.ClassCastException: HashType cannot be cast to java.lang.Comparable
*///:~
```

Das Beispiel definiert drei Klassen als Elementtypen, um zu zeigen, welche Methoden für die verschiedenen Containertypen unter dem Interface *Set* benötigt werden und duplizierten Quelltext zu vermeiden. Die Basisklasse `SetType` enthält lediglich ein `int`-Feld, dessen Inhalt über `toString()` zurückgegeben werden kann. Da jedes in einem Container vom Typ *Set* gespeicherte Element eine `equals()`-Methode besitzen muß, wird auch diese in der Basisklasse angelegt. Das Gleichheitskriterium bezieht sich auf den Wert des `int`-Feldes `i`.

[43] Die Klasse `HashType` ist von `SetType` abgeleitet und definiert eine `hashCode()`-Methode. Jedes in eine hashbasierte Implementierung von *Set* eingesetzte Element muß eine `hashCode()`-Methode besitzen.

[44] Das von der Klasse `TreeType` implementierte Interface *Comparable* ist notwendig, wenn ein Element in einem sortierten Container gespeichert werden soll, etwa vom Typ *SortedSet* (die Klasse `TreeSet` ist die einzige Implementierung dieses Interfaces). Beachten Sie, daß die `compareTo()`-Methode *nicht* das „einfach und offensichtliche“ Kriterium `i-i2` verwendet. Dieses Kriterium ist ein häufiger Programmierfehler, da es nur korrekt funktionieren würde, wenn `i` und `i2` *vorzeichenlose* `int`-Werte wären (wenn Java ein `unsigned`-Schlüsselwort besäße). Das Kriterium `i-i2` versagt, weil der *vorzeichenbehaftete* primitive Typ `int` nicht in der Lage ist, die Differenz zweier im Betrag genügend großer `int`-Werte zu speichern: Enthält `i` einen großen positiven und `j` einen großen negativen `int`-Wert, so führt die Differenz `i-j` zu einem Speicherüberlauf und der Rückgabe eines negativen Wertes.

[45] In der Regel möchten Sie, daß die `compareTo()`-Methode eine mit der `equals()`-Methode konsistente natürliche Ordnung liefert. Gibt `equals()` für einen bestimmten Vergleich `true` zurück, so sollte `compareTo()` bei demselben Vergleich Null liefern und einen von Null verschiedenen Wert, falls `equals() false` zurückgibt.

[46] Die Methoden `fill()` und `test()` in der Klasse `TypesForSets` sind als generische Methoden definiert, um die Vervielfältigung von Teilen des Quelltextes zu umgehen. Das typische Verhalten eines *Set*-Containers wird bestätigt, indem die `test()`-Methode dreimal `fill()` auf dem von `set` referenzierten Testcontainer aufruft, also Duplikate einzusetzen versucht. Die `fill()`-Methode erwartet einen *Set*-Container mit beliebigem Elementtyp sowie das Klassenobjekt des gewählten Elementtyps. Das Klassenobjekt wird benötigt, um den Konstruktor für ein `int`-Argument zu ermitteln und aufzurufen, um neue Elemente in den Container einzusetzen.

[47] Sie sehen an der Ausgabe, daß der Containertyp `HashSet` seine Elemente auf eine geheimnisvolle Weise anordnet (die in Abschnitt 18.9 erklärt wird), `LinkedHashSet` seine Elemente in der Einsetzungsreihenfolge und `TreeSet` dagegen sortiert speichert (der `compareTo()`-Methode entsprechend, hier also in absteigender Reihenfolge).

[48] Der Versuch, Elemente einzusetzen, welche die von einem *Set*-Container benötigten Methoden nicht in angemessener Weise unterstützen, geht gründlich schief. Das Eintragen von `SetType`- oder `TreeType`-Objekten, die keine eigene `hashCode()`-Methode definieren, in einen hashbasierten *Set*-Container, führt zu Duplikaten und verletzt somit eine fundamentale Eigenschaft dieses Con-

tainertyps. Das ist ziemlich beunruhigend, da nicht einmal zur Laufzeit ein Fehler gemeldet wird. Die von `Object` vererbte `hashCode()`-Methode ist zulässig, wodurch das falsche Verhalten legitimiert wird. Die einzige zuverlässige Möglichkeit, um die Korrektheit eines solchen Programmes zu gewährleisten, ist das Einbinden von Modultests (*unit tests*) in Ihren Erstellungsprozeß (zu weiteren Informationen siehe Anhang von *http://www.mindview.net/Books/BetterJava*).

[49] Wenn Sie versuchen, einem `TreeSet`-Container ein Element einer Klasse, die das Interface *`Comparable`* nicht implementiert zu übergeben, erhalten Sie ein deutliches Ergebnis. Der Container wirft beim Versuch, das Element als komparabel zu behandeln, eine Ausnahme aus.

18.6.1 Das Interface SortedSet

[50] Das Interface *`SortedSet`* beschreibt einen Containertyp, der seine Elemente garantiert in sortierter Reihenfolge enthält und gestattet dadurch zusätzliche Funktionalität, die von den folgenden in *`SortedSet`* deklarierten Methoden zur Verfügung gestellt wird:

- *`Comparator`* `comparator()` gibt die Referenz auf den für diesen Container verwendeten Komparator beziehungsweise `null` zurück, falls die natürliche Ordnung verwendet wird.
- `Object first()` gibt die Referenz auf das kleinste Element zurück.
- `Object last()` gibt die Referenz auf das größte Element zurück.
- *`SortedSet`* `subSet(fromElement, toElement)` liefert eine Sicht (*view*) auf diesen Container zwischen den Elementen `fromElement` (inklusive) und `toElement` (nicht inklusive).
- *`SortedSet`* `headSet(toElement)` liefert eine Sicht auf diesen Container über alle Elemente, die kleiner als `toElement` sind.
- *`SortedSet`* `tailSet(fromElement)` liefert eine Sicht auf diesen Container über alle Elemente, die größer oder gleich `fromElement` sind.

[51] Der folgende Beispiel zeigt diese Methoden in Aktion:

```
//: containers/SortedSetDemo.java
// What you can do with a TreeSet.
import java.util.*;
import static net.mindview.util.Print.*;

public class SortedSetDemo {
    public static void main(String[] args) {
        SortedSet<String> sortedSet = new TreeSet<String>();
        Collections.addAll(sortedSet,
                           ''one two three four five six seven eight''.split(('' '')));
        print(sortedSet);
        String low = sortedSet.first();
        String high = sortedSet.last();
        print(low);
        print(high);
        Iterator<String> it = sortedSet.iterator();
        for(int i = 0; i <= 6; i++) {
            if(i == 3) low = it.next();
            if(i == 6) high = it.next();
            else it.next();
        }
        print(low);
        print(high);
        print(sortedSet.subSet(low, high));
```

```
        print(sortedSet.headSet(high));
        print(sortedSet.tailSet(low));
    }
} /* Output:
      [eight, five, four, one, seven, six, three, two]
      eight
      two
      one
      two
      [one, seven, six, three]
      [eight, five, four, one, seven, six, three]
      [one, seven, six, three, two]
   *///:~
```

Beachten Sie, daß die Elemente eines Containers vom Typ *SortedSet* bezüglich des Vergleichskriteriums des Containers (des Komparators) sortiert werden, nicht nach der Einsetzungsreihenfolge. Die Einsetzungsreihenfolge bleibt beim Containertyp `LinkedHashSet` erhalten.

Übungsaufgabe 9: (2) Füllen Sie einen Container vom Typ `TreeSet` mit Hilfe der Generatorklasse `RandomGenerator.String`, aber mit alphabetischer Sortierung. Geben Sie den Containerinhalt aus, um die Sortierung zu verifizieren. ■

Übungsaufgabe 10: (7) Schreiben Sie eine eigene Implementierung des Interfaces *SortedSet*. Verwenden Sie einen Container vom Typ `LinkedList` als unterliegende Datenstruktur. ■

18.7 Das Interface Queue

[52] Die SE 5 liefert, abgesehen von den Warteschlangencontainern für threadbasierte Anwendungen, nur zwei Implementierungen für das Interface *Queue*, nämlich `LinkedList` und `PriorityQueue`, die sich durch ihr Ordnungsverhalten deutlicher unterscheiden als hinsichtlich ihrer Performanz. Das folgende einfache Beispiel umfaßt den größten Teil der *Queue*-Implementierungen (nicht alle Methoden lassen sich in diesem Beispiel anwenden), darunter auch die für die Threadprogrammierung vorgesehenen Containerklassen. Elemente werden an einem Ende eingesetzt und am anderen Ende entnommen:

```
//: containers/QueueBehavior.java
// Compares the behavior of some of the queues
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

public class QueueBehavior {
    private static int count = 10;
    static <T> void test(Queue<T> queue, Generator<T> gen) {
        for(int i = 0; i < count; i++)
            queue.offer(gen.next());
        while(queue.peek() != null)
            System.out.print(queue.remove() + " ");
        System.out.println();
    }
    static class Gen implements Generator<String> {
        String[] s =
            ("one two three four five six seven eight nine ten").split(" ");
        int i;
        public String next() { return s[i++]; }
```

```
    }
    public static void main(String[] args) {
        test(new LinkedList<String>(), new Gen());
        test(new PriorityQueue<String>(), new Gen());
        test(new ArrayBlockingQueue<String>(count), new Gen());
        test(new ConcurrentLinkedQueue<String>(), new Gen());
        test(new LinkedBlockingQueue<String>(), new Gen());
        test(new PriorityBlockingQueue<String>(), new Gen());
    }
} /* Output:
      one two three four five six seven eight nine ten
      eight five four nine one seven six ten three two
      one two three four five six seven eight nine ten
      one two three four five six seven eight nine ten
      one two three four five six seven eight nine ten
      eight five four nine one seven six ten three two
   *///:~
```

Mit Ausnahme der Prioritätswarteschlangen geben alle *Queue*-Container ihre Element in der Einsetzungsreihenfolge heraus.

18.7.1 Prioritätswarteschlangen

[53] Unterabschnitt 12.11.1 enthält eine einfache Einführung in das Thema „Prioritätswarteschlangen“. Eine interessantere Aufgabe ist eine Todo-Liste von Objekten, die einen Beschreibungstext sowie einen primären und einen sekundären Prioritätswert enthalten. Die Ordnung in dieser Liste wird durch Implementieren des *Comparable*-Interfaces gesteuert:

```
//: containers/ToDoList.java
// A more complex use of PriorityQueue.
import java.util.*;

class ToDoList extends PriorityQueue<ToDoList.ToDoItem> {
    static class ToDoItem implements Comparable<ToDoItem> {
        private char primary;
        private int secondary;
        private String item;
        public ToDoItem(String td, char pri, int sec) {
            primary = pri;
            secondary = sec;
            item = td;
        }
        public int compareTo(ToDoItem arg) {
            if(primary > arg.primary)
                return +1;
            if(primary == arg.primary)
                if(secondary > arg.secondary)
                    return +1;
                else if(secondary == arg.secondary)
                    return 0;
            return -1;
        }
        public String toString() {
            return Character.toString(primary) +
                secondary + ": " + item;
        }
    }
```

```
    public void add(String td, char pri, int sec) {
        super.add(new ToDoItem(td, pri, sec));
    }
    public static void main(String[] args) {
        ToDoList toDoList = new ToDoList();
        toDoList.add("Empty trash", 'C', 4);
        toDoList.add("Feed dog", 'A', 2);
        toDoList.add("Feed bird", 'B', 7);
        toDoList.add("Mow lawn", 'C', 3);
        toDoList.add("Water lawn", 'A', 1);
        toDoList.add("Feed cat", 'B', 1);
        while(!toDoList.isEmpty())
            System.out.println(toDoList.remove());
    }
} /* Output:
      A1: Water lawn
      A2: Feed dog
      B1: Feed cat
      B7: Feed bird
      C3: Mow lawn
      C4: Empty trash
   *///:~
```

Die Einträge werden bei einer Prioritätswarteschlange automatisch geordnet.

Übungsaufgabe 11: (2) Schreiben Sie eine Klasse mit einen `Integer`-Feld, welches mit Hilfe der Klasse `java.util.Random` mit einem Zufallswert zwischen 0 und 100 initialisiert wird. Implementieren Sie das Interface *`Comparable`* anhand des `Integer`-Feldes. Füllen Sie einen Container vom Typ `PriorityQueue` (eine Prioritätswarteschlange) mit Objekten Ihrer Klasse und entnehmen Sie die Elemente mit Hilfe der Methode `poll()`, um zu zeigen, daß die Elemente in der erwarteten Reihenfolge sortiert wurden. ■

18.7.2 Doppelköpfige Warteschlangen

[54] Eine doppelköpfige Warteschlange (auch „Deque“, nach der englischen Bezeichung „double-ended queue“) verhält sich wie eine Warteschlange, wobei Sie aber an beiden Enden Elemente einsetzen beziehungsweise entnehmen können. Die Klasse `LinkedList` unterstützt die Operationen einer doppelköpfigen Warteschlange durch entsprechende Methoden, die Standardbibliothek von Java enthält aber kein explizites Interface für diese Funktionalität. `LinkedList` kann somit kein *`Deque`*-Interface implementieren, wie bei *`Queue`* im vorigen Beispiel. Sie können aber selbst eine Klasse `Deque` per Komposition anlegen und einfach die relevanten Methoden der Klasse `LinkedList` exponieren:

```
//: net/mindview/util/Deque.java
// Creating a Deque from a LinkedList.
package net.mindview.util;
import java.util.*;

public class Deque<T> {
    private LinkedList<T> deque = new LinkedList<T>();
    public void addFirst(T e) { deque.addFirst(e); }
    public void addLast(T e) { deque.addLast(e); }
    public T getFirst() { return deque.getFirst(); }
    public T getLast() { return deque.getLast(); }
    public T removeFirst() { return deque.removeFirst(); }
    public T removeLast() { return deque.removeLast(); }
```

```
    public int size() { return deque.size(); }
    public String toString() { return deque.toString(); }
    // And other methods as necessary...
} ///:~
```

Wenn Sie diese `Deque`-Klasse in einem Ihrer eigenen Programme verwenden, müssen Sie eventuell zusätzliche Methoden anlegen, um die Klasse praktisch verwertbar zu machen.

[55] Das nächste Beispiel ist ein einfacher Test für die obige `Deque`-Klasse:

```
//: containers/DequeTest.java
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DequeTest {
    static void fillTest(Deque<Integer> deque) {
        for(int i = 20; i < 27; i++)
            deque.addFirst(i);
        for(int i = 50; i < 55; i++)
            deque.addLast(i);
    }
    public static void main(String[] args) {
        Deque<Integer> di = new Deque<Integer>();
        fillTest(di);
        print(di);
        while(di.size() != 0)
            printnb(di.removeFirst() + " ");
        print();
        fillTest(di);
        while(di.size() != 0)
            printnb(di.removeLast() + " ");
    }
} /* Output:
      [26, 25, 24, 23, 22, 21, 20, 50, 51, 52, 53, 54]
      26 25 24 23 22 21 20 50 51 52 53 54
      54 53 52 51 50 20 21 22 23 24 25 26
   *////:~
```

Es ist weniger wahrscheinlich, daß Sie an beiden Enden Elemente einsetzen und entnehmen. Daher wird die doppelköpfige Warteschlange seltener verwendet als die gewöhnliche Version.

18.8 Das Interface Map

[56] Wie Sie in den Abschnitten 12.2 und 12.10 gelernt haben, besteht das Konzept des Containertyps *`Map`* („assoziatives Array“) darin, Schlüssel mit Werten zu Schlüssel/Wert-Paaren zu verknüpfen, so daß Sie einen Wert mit Hilfe seines Schlüssels nachschlagen (abfragen) können. Die Standardbibliothek von Java liefert verschiedene grundlegende Implementierungen des Interfaces *`Map`*: `HashMap`, `TreeMap`, `LinkedHashMap`, `WeakHashMap`, `ConcurrentHashMap` und `IdentityHashMap`. Jede dieser Klassen implementiert zwar das Interface *`Map`*, unterscheidet sich aber von den übrigen Implementierungen durch Verhalten, Effizienz, die Reihenfolge in der die Schlüssel/Wert-Paare gespeichert und herausgegeben werden, die Verweildauer der Paare im Container, das Verhalten des Containers bei Beanspruchung durch mehrere Threads sowie das Gleichheitskriterium für die Schlüssel. Die Anzahl der Implementierungen des Interfaces *`Map`* ist ein Maß für die Bedeutsamkeit dieses Containertyps.

[57] Sie gewinnen ein tieferes Verständnis für den Containertyp *`Map`*, wenn Sie sich ansehen, wie

ein assoziatives Array programmiert wird. Das folgende Beispiel zeigt eine stark vereinfachte Implementierung:

```
//: containers/AssociativeArray.java
// Associates keys with values.
import static net.mindview.util.Print.*;

public class AssociativeArray<K,V> {
    private Object[][] pairs;
    private int index;
    public AssociativeArray(int length) {
        pairs = new Object[length][2];
    }
    public void put(K key, V value) {
        if(index >= pairs.length)
            throw new ArrayIndexOutOfBoundsException();
        pairs[index++] = new Object[] { key, value };
    }
    @SuppressWarnings(''unchecked'')
    public V get(K key) {
        for(int i = 0; i < index; i++)
            if(key.equals(pairs[i][0]))
                return (V) pairs[i][1];
        return null; // Did not find key
    }
    public String toString() {
        StringBuilder result = new StringBuilder();
        for(int i = 0; i < index; i++) {
            result.append(pairs[i][0].toString());
            result.append('' : '');
            result.append(pairs[i][1].toString());
            if(i < index - 1)
                result.append(''\n'');
        }
        return result.toString();
    }
    public static void main(String[] args) {
        AssociativeArray<String,String> map =
            new AssociativeArray<String,String>(6);
        map.put(''sky'', ''blue'');
        map.put(''grass'', ''green'');
        map.put(''ocean'', ''dancing'');
        map.put(''tree'', ''tall'');
        map.put(''earth'', ''brown'');
        map.put(''sun'', ''warm'');
        try {
            map.put(''extra'', ''object''); // Past the end
        } catch(ArrayIndexOutOfBoundsException e) {
            print(''Too many objects!'');
        }
        print(map);
        print(map.get(''ocean''));
    }
} /* Output:
      Too many objects!
      sky : blue
      grass : green
      ocean : dancing
```

```
        tree : tall
        earth : brown
        sun : warm
        dancing
    *///:~
```

Die wesentlichen Methoden eines assoziativen Arrays sind `put()` und `get()`. Die `toString()`-Methode wurde überschrieben, um die Schlüssel/Wert-Paare ansprechend ausgeben zu können. Die `main()`-Methode initialisiert ein Objekt der Klasse `AssociativeArray` mit einigen Einträgen und gibt den resultierenden Containerinhalt aus, gefolgt von einem Aufruf der `get()`-Methode, der einen einzelnen Wert liefert.

[58] Die `get()`-Methode erwartet den nachzuschlagenden Schlüssel und liefert den damit verknüpften Wert beziehungsweise `null`, falls der Schlüssel nicht existiert. Die `get()`-Methode implementiert den wohl ineffizientesten Ansatz, um nach dem Wert zu suchen: Das „Verfahren“ beginnt beim ersten Arrayelement und ruft die `equals()`-Methode auf, um die Schlüssel zu vergleichen. Der Schwerpunkt dieses Beispiels liegt aber auf der Einfachheit, nicht der Effizienz.

[59] Das obige Beispiel ist somit lehrreich, aber nicht effizient. Die begrenzte Kapazität macht den Ansatz außerdem unflexibel. Die *`Map`*-Container im Package `java.util` haben diese Probleme nicht und können anstelle von `AssociativeArray` in das obige Beispiel eingesetzt werden.

Übungsaufgabe 12: (1) Ersetzen Sie im Beispiel *AssociativeArray.java* die Klasse `AssociativeArray` nacheinander durch Container der Typen `HashMap`, `TreeMap` und `LinkedHashMap`. ■

Übungsaufgabe 13: (4) Erweitern Sie das Beispiel *AssociativeArray.java* zu einem Programm, welches die Häufigkeiten von Wörtern zählt, indem es `String`-Objekte auf `Integer`-Objekte abbildet. Verwenden Sie die in Unterabschnitt 19.7.1 beschriebene Klasse `TextFile`, um eine Textdatei einzulesen und den Inhalt an Leerraum sowie Interpunktionszeichen aufzutrennen sowie um die Vorkommen der einzelnen Wörter in der Textdatei zu zählen. ■

18.8.1 Performanz

[60] Performanz ist ein fundamentales Anliegen der Container vom Typ *`Map`* und die lineare Suche nach einem Schlüssel in der obigen `get()`-Methode ist sehr langsam. Die Klasse `HashMap` implementiert ein viel schnelleres Suchverfahren. Statt der langsamen Suche nach einem Schlüssel, basiert dieses Verfahren auf einem sogenannten *Hashwert*. Der Hashwert eines Objektes ist ein aus bestimmten Eigenschaften des fraglichen Objektes ermittelter „relativ eindeutiger“ `int`-Wert. Die Wurzelklasse `Object` vererbte ihre `hashCode()`-Methode an alle Java-Klassen, wodurch jedes Objekt in der Lage ist, einen Hashwert zu liefern. Ein Container vom Typ `HashMap` verwendet den Hashwert eines Objektes zur schnellen Suche nach dem Schlüssel. Daraus erwächst eine dramatische Verbesserung der Performanz.[6]

[61] Die folgende Liste zeigt die grundlegenden Implementierung des Interfaces *`Map`*. Das Sternchen (*) bei `HashMap` bedeutet, daß Sie, falls keine weiteren Rahmenbedingungen existieren, diesen Con-

[6] Falls diese Beschleunigung Ihren Anforderungen an die Performanz nicht genügt, können Sie das Nachschlagen in der Tabelle zusätzlich schneller machen, indem Sie eine eigene Implementierung des Interfaces *`Map`* schreiben und an Ihre Objekttypen anpassen, um den Zeitaufwand durch die Typwandlungen von und nach `Object` zu vermeiden. Geschwindigkeitsenthusiasten können, um noch höhere Niveaus an Performanz zu erreichen, Donald Knuths Buch *The Art of Computer Programming, Volume 3: Sorting and Searching* (2nd ed.) zu Rate ziehen und überlaufende Bucketlisten durch Arrays ersetzen, welche zwei Vorteile haben: Einerseits lassen sich Arrays hinsichtlich ihrer Speicherungseigenschaften auf der Festplatte optimieren. Andererseits sparen Arrays den größten Teil der Zeit, die zum Erzeugen sowie zum Aufräumen von Datensätzen durch die automatische Speicherbereinigung benötigt wird.

tainertyp wählen sollten, weil er hinsichtlich der Zugriffsgeschwindigkeit optimiert ist. Die übrigen Implementierungen verkörpern andere Eigenschaften und sind weniger schnell:

- `HashMap`*: Die Implementierung basiert auf einer Hashtabelle (verwenden Sie `HashMap` anstelle von `Hashtable`) und gestattet Einsetzen und Nachschlagen von Schlüssel/Wert-Paaren in konstanter Zeit. Die Performanz kann über den Konstruktor (Kapazität und Ladefaktor (*load factor*) der Hashtabelle) justiert werden.
- `LinkedHashMap`: Wie `HashMap`, aber Sie erhalten die Schlüssel/Wert-Paare bei Iteration über den Containerinhalt in der Einsetzungsreihenfolge beziehungsweise in der am längsten nicht verwendeten (*least-recently-used*, LRU) Reihenfolge. Mit Ausnahme der Iteration ist `LinkedHashMap` nur geringfügig langsamer als `HashMap`. Bei Iteration ist `LinkedHashMap` aufgrund der verketteten Liste zur Aufrechterhaltung der Einsetzungsreihenfolge schneller als `HashMap`.
- `TreeMap`: Die Implementierung basiert auf einem Rot-Schwarz-Baum. Schlüssel und Schlüssel/Wert-Paare werden sortiert ausgegeben (die Ordnung richtet sich nach der *`Comparable`*-Implementierung der Schlüssel beziehungsweise dem Komparator des Containers.) Der Nutzen des Containertyps `TreeMap` besteht in der sortierten Ausgabe der Schlüssel beziehungsweise Schlüssel/Wert-Paare. Die Klasse `TreeMap` ist lediglich eine Implementierung des Interfaces *`Map`* mit einer `subMap()`-Methode, die einen Teil der Baumstruktur zurückgibt.
- `WeakHashMap`: Ein *`Map`*-Container mit *schwachen Schlüsseln*, der die Freigabe von Objekten gestattet, die mit Schlüsseln im Container verknüpft sind. Das Design dieses Containertyps ist an die Lösung gewisser Probleme angepaßt. Existiert außerhalb des Containers keine Referenz auf das mit einem bestimmten Schlüssel verknüpfte Objekt, so darf dieser Schlüssel der automatischen Speicherbereinigung übergeben werden.
- `ConcurrentHashMap`: Threadsicherer *`Map`*-Container ohne synchronisierungsbedingte Sperre (siehe Unterabschnitt 22.9.2, insbesondere Unterunterabschnitt 22.9.2.2).
- `IdentityHashMap`: Ein *`Map`*-Container, der den `==`-Operator statt der `equals()`-Methode zum Vergleichen der Schlüssel verwendet. Nur zur Lösung spezieller Probleme, nicht aber zum allgemeinen Gebrauch.

Hashtabellen sind die häufigste Variante der Speicherung von Elementen in einem Container vom Typ *`Map`*. In Abschnitt 18.9 lernen Sie mehr darüber, wie Hashalgorithmen funktionieren.

[62] Die Anforderungen an die Schlüssel eines Containers vom Typ *`Map`* sind deckungsgleich mit den Anforderungen an die Elemente eines Containers vom Typ *`Set`* (siehe Beispiel *TypesForSets.java*, Seite 638). Jeder Schlüssel muß eine `equals()`-Methode besitzen, falls der Schlüssel in einem hashbasierten *`Map`*-Container verwendet wird, auch eine passende `hashCode()`-Methode. Wird der Schlüssel in einem `TreeMap`-Container verwendet, so muß seine Klasse auch das Interface *`Comparable`* implementieren.

[63] Das folgende Beispiel zeigt die im Interface *`Map`* deklarierten Methoden. Der Container wird mit Hilfe der auf Seite 628f definierten Hilfsklasse `CountingMapData` initialisiert:

```
//: containers/Maps.java
// Things you can do with Maps.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class Maps {
    public static void printKeys(Map<Integer,String> map) {
        printnb("Size = " + map.size() + ", ");
```

```
        printnb("Keys: ");
        print(map.keySet()); // Produce a Set of the keys
    }
    public static void test(Map<Integer,String> map) {
        print(map.getClass().getSimpleName());
        map.putAll(new CountingMapData(25));
        // Map has 'Set' behavior for keys:
        map.putAll(new CountingMapData(25));
        printKeys(map);
        // Producing a Collection of the values:
        printnb("Values: ");
        print(map.values());
        print(map);
        print("map.containsKey(11): " + map.containsKey(11));
        print("map.get(11): " + map.get(11));
        print("map.containsValue(\"F0\"): " + map.containsValue("F0"));
        Integer key = map.keySet().iterator().next();
        print("First key in map: " + key);
        map.remove(key);
        printKeys(map);
        map.clear();
        print("map.isEmpty(): " + map.isEmpty());
        map.putAll(new CountingMapData(25));
        // Operations on the Set change the Map:
        map.keySet().removeAll(map.keySet());
        print("map.isEmpty(): " + map.isEmpty());
    }
    public static void main(String[] args) {
        test(new HashMap<Integer,String>());
        test(new TreeMap<Integer,String>());
        test(new LinkedHashMap<Integer,String>());
        test(new IdentityHashMap<Integer,String>());
        test(new ConcurrentHashMap<Integer,String>());
        test(new WeakHashMap<Integer,String>());
    }
} /* Output:
      HashMap
      Size = 25, Keys: [15, 8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4, 19,
      11, 18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
      Values: [P0, I0, X0, Q0, H0, W0, J0, V0, G0, B0, O0, Y0, E0, T0, L0,
      S0, D0, M0, R0, C0, N0, U0, K0, F0, A0]
      {15=P0, 8=I0, 23=X0, 16=Q0, 7=H0, 22=W0, 9=J0, 21=V0, 6=G0, 1=B0,
      14=O0, 24=Y0, 4=E0, 19=T0, 11=L0, 18=S0, 3=D0, 12=M0, 17=R0, 2=C0,
      13=N0, 20=U0, 10=K0, 5=F0, 0=A0}
      map.containsKey(11): true
      map.get(11): L0
      map.containsValue("F0"): true
      First key in map: 15
      Size = 24, Keys: [8, 23, 16, 7, 22, 9, 21, 6, 1, 14, 24, 4, 19, 11,
      18, 3, 12, 17, 2, 13, 20, 10, 5, 0]
      map.isEmpty(): true
      map.isEmpty(): true
      ...
    *///:~
```

Die `printKeys()`-Methode zeigt eine *`Collection`*-Sicht auf einen Container vom Typ *`Map`*. Die Methode `keySet()` gibt eine Referenz auf einen *`Set`*-Container zurück, der sich auf die Schlüssel des

Map-Containers bezieht. Die verbesserten Ausgabemöglichkeiten für Container in der SE 5 gestatten, das Ergebnis der `values()`-Methode, nämlich einen Container vom Typ *Collection*, mühelos auszugeben. (Beachten Sie, daß die Schlüssel eindeutig sein müssen, während unter den Werten Duplikate erlaubt sind.) Da sich der Schlüssel- beziehungsweise Wertcontainer auf den ursprünglichen *Map*-Container stützt, schlagen sich Änderungen am Schlüssel- beziehungsweise Wertcontainer im *Map*-Container nieder und umgekehrt.

[64] Das restliche Programm zeigt ein einfaches Beispiel für jede im Interface *Map* deklarierte Methode und testet anschließend alle grundlegenden Implementierungen.

Übungsaufgabe 14: (3) Zeigen Sie, daß sich die Klasse `java.util.Properties` in das obige Programm einsetzen läßt. ■

18.8.2 Das Interface SortedMap

[65] Das Interface *SortedMap* (die Klasse `TreeMap` ist die einzige verfügbare Implementierung) beschreibt einen Containertyp, der seine Schlüssel garantiert in sortierter Reihenfolge speichert und gestattet dadurch zusätzliche Funktionalität, die von den folgenden in *SortedMap* deklarierten Methoden zur Verfügung gestellt wird:

- *Comparator* `comparator()` gibt die Referenz auf den für diesen Container verwendeten Komparator beziehungsweise `null` zurück, falls die natürliche Ordnung verwendet wird.
- `T firstKey()` gibt die Referenz auf den kleinsten Schlüssel zurück.
- `T lastKey()` gibt die Referenz auf den größten Schlüssel zurück.
- *SortedMap* `subMap(fromKey, toKey)` liefert eine Sicht (*view*) auf diesen Container zwischen den Schlüsseln `fromKey` (inklusive) und `toKey` (nicht inklusive).
- *SortedMap* `headMap(toKey)` liefert eine Sicht auf diesen Container über alle Schlüssel, die kleiner als `toKey` sind.
- *SortedMap* `tailMap(fromKey)` liefert eine Sicht auf diesen Container über alle Schlüssel, die größer oder gleich `fromKey` sind.

[66] Das folgende Beispiel ähnelt *SortedSetDemo.java* auf Seite 641 und zeigt die zusätzliche Funktionalität der Klasse `TreeMap`:

```
//: containers/SortedMapDemo.java
// What you can do with a TreeMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class SortedMapDemo {
    public static void main(String[] args) {
        TreeMap<Integer,String> sortedMap =
            new TreeMap<Integer,String>(new CountingMapData(10));
        print(sortedMap);
        Integer low = sortedMap.firstKey();
        Integer high = sortedMap.lastKey();
        print(low);
        print(high);
        Iterator<Integer> it = sortedMap.keySet().iterator();
        for(int i = 0; i <= 6; i++) {
            if(i == 3) low = it.next();
```

```
            if(i == 6) high = it.next();
            else it.next();
        }
        print(low);
        print(high);
        print(sortedMap.subMap(low, high));
        print(sortedMap.headMap(high));
        print(sortedMap.tailMap(low));
    }
} /* Output:
      {0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
      0
      9
      3
      7
      {3=D0, 4=E0, 5=F0, 6=G0}
      {0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0}
      {3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0, 9=J0}
   *///:~
```

Die Schlüssel/Wert-Paare werden in der Reihenfolge gespeichert, die sich aus der Sortierung der Schlüssel ergibt. Da die Elemente eines `TreeMap`-Containers geordnet sind, ist das Konzept „Position“ sinnvoll, so daß Sie das erste beziehungsweise letzte Element und Teilmengen abfragen können.

18.8.3 Die Klasse LinkedHashMap

[67] Der Containertyp `LinkedHashMap` verwendet zur Verbesserung der Zugriffsgeschwindigkeit einen Hashalgorithmus, gibt seine Schlüssel/Wert-Paare aber beim Traversieren des Containerinhaltes in der Einsetzungsreihenfolge heraus (`System.out.println()` iteriert über den Containerinhalt, zeigt also das Ergebnis des Traversierens an). Außerdem kann ein `LinkedHashMap`-Container per Konstruktor so konfiguriert werden, daß die Elemente nach dem LRU-Verfahren (*least-recently-used*: „am längsten nicht verwendet“) sortiert werden, wobei Elemente, die keinen Zugriff mehr erfahren haben (und somit Kandidaten zum Entfernen sind) vorne in der Liste stehen. Das folgende einfache Beispiel zeige beide Fähigkeiten:

```
//: containers/LinkedHashMapDemo.java
// What you can do with a LinkedHashMap.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class LinkedHashMapDemo {
    public static void main(String[] args) {
        LinkedHashMap<Integer,String> linkedMap =
            new LinkedHashMap<Integer,String>(new CountingMapData(9));
        print(linkedMap);
        // Least-recently-used order:
        linkedMap = new LinkedHashMap<Integer,String>(16, 0.75f, true);
        linkedMap.putAll(new CountingMapData(9));
        print(linkedMap);
        for(int i = 0; i < 6; i++) // Cause accesses:
            linkedMap.get(i);
        print(linkedMap);
        linkedMap.get(0);
        print(linkedMap);
    }
```

```
} /* Output:
      {0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
      {0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 6=G0, 7=H0, 8=I0}
      {6=G0, 7=H0, 8=I0, 0=A0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0}
      {6=G0, 7=H0, 8=I0, 1=B0, 2=C0, 3=D0, 4=E0, 5=F0, 0=A0}
   *///:~
```

Die Ausgabe zeigt, daß die Schlüssel/Wert-Paare tatsächlich in der Einsetzungsreihenfolge durchlaufen werden, auch bei der LRU-Version. Nachdem in der LRU-Version Zugriffe auf die ersten sechs Elemente stattgefunden haben, wandern die drei letzten Elemente an den Anfang der Liste. Nach dem Zugriff auf den Eintrag mit dem Schlüssel „0“ wird das entsprechende Paar ans Ende der Liste gesetzt.

18.9 Hashalgorithmen und Hashwerte

[68–69] Die Beispiele in Kapitel 12 verwenden bei Containern vom Typ `HashMap` Objekte vordefinierter Klassen als Schlüssel. Diese Beispiele funktionieren, weil die vordefinierten Klassen über die gesamte notwendige „Verdrahtung“ verfügen, damit sich ihre Objekte als Schlüssel korrekt verhalten. Das Vergessen dieser erforderlichen Verdrahtung ist ein häufiger Fehler beim Schreiben eigener Klassen, deren Objekte als Schlüssel eines Containers vom Typ `HashMap` dienen sollen. Wir verwenden ein Wettervorhersagesystem als Beispiel, das `Groundhog`-Objekte (*groundhog*: Murmeltier) auf `Prediction`-Objekte abbildet. Die Aufgabe ist scheinbar ziemlich einfach: Sie schreiben zwei Klassen und verwenden `Groundhog` als Schlüssel beziehungsweise `Prediction` als Werte:

```
//: containers/Groundhog.java
// Looks plausible, but doesn't work as a HashMap key.

public class Groundhog {
    protected int number;
    public Groundhog(int n) { number = n; }
    public String toString() {
        return "Groundhog #" + number;
    }
} ///:~

//: containers/Prediction.java
// Predicting the weather with groundhogs.
import java.util.*;

public class Prediction {
    private static Random rand = new Random(47);
    private boolean shadow = rand.nextDouble() > 0.5;
    public String toString() {
        if(shadow)
            return "Six more weeks of Winter!";
        else
            return "Early Spring!";
    }
} ///:~

//: containers/SpringDetector.java
// What will the weather be?
import java.lang.reflect.*;
import java.util.*;
import static net.mindview.util.Print.*;
```

```
public class SpringDetector {
    // Uses a Groundhog or class derived from Groundhog:
    public static <T extends Groundhog>
        void detectSpring(Class<T> type) throws Exception {
        Constructor<T> ghog = type.getConstructor(int.class);
        Map<Groundhog,Prediction> map = new HashMap<Groundhog,Prediction>();
        for(int i = 0; i < 10; i++)
            map.put(ghog.newInstance(i), new Prediction());
        print("map = " + map);
        Groundhog gh = ghog.newInstance(3);
        print("Looking up prediction for " + gh);
        if(map.containsKey(gh))
            print(map.get(gh));
        else
            print("Key not found: " + gh);
    }
    public static void main(String[] args) throws Exception {
        detectSpring(Groundhog.class);
    }
} /* Output:
      map = {Groundhog #3=Early Spring!, Groundhog #7=Early Spring!,
        Groundhog #5=Early Spring!, Groundhog #9=Six more weeks of Winter!,
        Groundhog #8=Six more weeks of Winter!,
        Groundhog #0=Six more weeks of Winter!,
        Groundhog #6=Early Spring!, Groundhog #4=Six more weeks of Winter!,
        Groundhog #1=Six more weeks of Winter!, Groundhog #2=Early Spring!}
      Looking up prediction for Groundhog #3
      Key not found: Groundhog #3
   *///:~
```

[70] Jedes `Groundhog`-Objekt hat eine Kennzahl, so daß Sie ein `Prediction`-Objekt im `HashMap`-Container nachschlagen können, in dem Sie beispielsweise das `Prediction`-Objekt anfordern, welches mit `Groundhog`-Objekt Nummer 3 verknüpft ist. Die Klasse `Prediction` hat ein `boolean`-Feld, das mit Hilfe von `java.util.Random` initialisiert wird sowie eine `toString()`-Methode, die den Inhalt dieses Feldes interpretiert. Die Methode `detectSpring()` nutzt den Reflexionsmechanismus, um Objekte der Klasse `Groundhog` oder einer hiervon abgeleiteten Klasse zu erzeugen. Dieser Ansatz zahlt sich auf Seite 654 aus, wenn wir eine neue Klasse von `Groundhog` ableiten, um das hier gezeigte Problem zu lösen.

[71] Die `detectSpring()`-Methode füllt einen `HashMap`-Container mit `Groundhog`/`Prediction`-Paaren. Der Containerinhalt wird einmal ausgegeben, damit Sie sehen, daß der Container gefüllt wurde. Anschließend wird ein `Groundhog`-Objekt mit der Kennzahl 3 als Schlüssel verwendet, um die Wettervorhersage dieses Murmeltiers abzurufen. (Sie sehen an der Ausgabe des Containerinhalts, daß ein entsprechendes `Groundhog`/`Prediction`-Paar existiert.)

[72] Das Programm ist zwar einfach, funktioniert aber nicht: Der Schlüssel für das `Groundhog`-Objekt mit der Kennzahl 3 ist nicht auffindbar. Das Problem besteht darin, daß die Klasse `Groundhog` automatisch von der universellen Wurzelklasse `Object` abgeleitet wird und somit die `hashCode()`-Methode von `Object` verwendet, um die Hashwerte der einzelnen Objekte zu berechnen. Die `Object`-Version der `hashCode()`-Methode liefert die Speicheradresse ihres Objektes. Folglich haben die beiden `Groundhog`-Objekte mit der Kennzahl 3 verschiedene Hashwerte.

[73] Vielleicht überlegen Sie sich gerade, daß es ausreicht, die `hashCode()`-Methode mit passender Funktionalität zu überschreiben. Aber das genügt nicht, bis Sie auch die `equals()`-Methode überschreiben, die ebenfalls von `Object` vererbt wird. Der `HashMap`-Container braucht die `equals()`-Methode, um zu ermitteln, ob der übergebene Schlüssel mit einem Schlüssel in der Hashtabelle

übereinstimmt.

[74] Die `equals()`-Methode ist eine Äquivalenzrelation, hat also die folgenden fünf Eigenschaften:

- *Reflexivität*. Für beliebige `x` gilt: `x.equals(x)` liefert `true`.
- *Symmetrie*. Für beliebige `x`, `y` gilt: `x.equals(y)` liefert genau dann `true`, wenn `y.equals(x)` `true` liefert.
- *Transitivität*. Für beliebige `x`, `y`, `z` gilt: Liefern `x.equals(y)` und `y.equals(z)` `true`, so auch `x.equals(z)`.
- *Konsistenz*. Für beliebige `x`, `y` gilt: Beliebig viele Aufrufe von `x.equals(y)` liefern konsistent entweder `true` oder `false`, vorausgesetzt, daß sich keine für den Vergleich per `equals()`-Methode ausgewertete Eigenschaften eines der beiden Objekte verändert hat.
- Für alle von `null` verschiedenen `x` gilt: `x.equals(null)` liefert `false`.

[75] Die `Object`-Version der `equals()`-Methode vergleicht, wie `hashCode()`, nur die Speicheradressen der Objekte, so daß zwei `Groundhog`-Objekte mit der Kennzahl 3 voneinander verschiedene Objekte sind. Wenn Sie also Objekte einer eigene Klasse als Schlüssel eines `HashMap`-Containers verwenden wollen, müssen Sie die Methoden `hashCode()` und `equals()` überschreiben. Das folgende Beispiel zeigt eine Lösung für das Murmeltierproblem:

```
//: containers/Groundhog2.java
// A class that's used as a key in a HashMap
// must override hashCode() and equals().

public class Groundhog2 extends Groundhog {
    public Groundhog2(int n) { super(n); }
    public int hashCode() { return number; }
    public boolean equals(Object o) {
        return o instanceof Groundhog2 &&
            (number == ((Groundhog2) o).number);
    }
} ///:~

//: containers/SpringDetector2.java
// A working key.

public class SpringDetector2 {
    public static void main(String[] args) throws Exception {
        SpringDetector.detectSpring(Groundhog2.class);
    }
} /* Output:
      map = {Groundhog #2=Early Spring!, Groundhog #4=Six more weeks of Winter!,
        Groundhog #9=Six more weeks of Winter!,
        Groundhog #8=Six more weeks of Winter!, Groundhog #6=Early Spring!,
        Groundhog #1=Six more weeks of Winter!, Groundhog #3=Early Spring!,
        Groundhog #7=Early Spring!, Groundhog #5=Early Spring!,
        Groundhog #0=Six more weeks of Winter!}
      Looking up prediction for Groundhog #3
      Early Spring!
   *///:~
```

[76] Die `Groundhog2`-Version von `hashCode()` gibt die Kennzahl als Hashwert zurück. In diesem Beispiel trägt der Programmierer die Verantwortung, dafür zu sorgen, daß keine zwei `Groundhog2`-Objekte mit gleichen Kennzahlen existieren. Anders als die `equals()`-Methode, welche streng ermitteln muß, ob zwei Objekte äquivalent sind oder nicht, wird von der `hashCode()`-Methode *nicht* verlangt, daß sie eindeutige Werte liefert (Sie werden diesen Aspekt später in diesem Kapitel besser

verstehen). Die `equals()`-Methode bezieht sich in diesem Beispiel auf die Kennzahl des `Groundhog2`-Objektes. Existieren in diesem `HashMap`-Container zwei `Groundhog2`-Objekte mit derselben Kennzahl als Schlüssel, ~~it will fail~~.

[77] Die `equals()`-Methode prüft scheinbar nur, ob ihr Argument ein Objekt der Klasse `Groundhog2` ist (mit Hilfe des `instanceof`-Operators, siehe Kapitel 15). Der `instanceof`-Operator bewirkt aber stillschweigend noch eine zweite Prüfung, nämlich ob ihr Argument `null` referenziert (der `instanceof`-Operator gibt `false` zurück, wenn sei linkes Argument `null` ist). Hat das Argument den richtigen Typ und ist nicht `null`, so vergleicht `equals()` die Inhalte der `number`-Felder der beiden Objekte. Wie Sie an der Ausgabe erkennen können, verhält sich das Programm nun korrekt.

[78] Wenn Sie Objekte einer eigenen Klasse in einem Container vom Typ `HashSet` speichern wollen, müssen Sie die gleichen Anforderungen einhalten, wie beim Containertyp `HashMap`.

18.9.1 Aufgabe der Methode hashCode()

[79] Das Murmeltierbeispiel dient nur zum Aufwärmen, bevor wir das Problem korrekt lösen. Es zeigt, daß eine hashbasierte Datenstruktur wie `HashSet`, `HashMap`, `LinkedHashSet` oder `LinkedHashMap`, ohne Überschreiben der Methoden `equals()` und `hashCode()` ihrer Schlüsselklasse, wahrscheinlich nicht richtig mit den Schlüsseln umgeht. Eine gute Lösung des Problems setzt allerdings voraus, daß Sie die Abläufe in einer hashbasierten Datenstruktur verstehen.

[80] Vergegenwärtigen Sie sich zuerst die Motivation „hinter“ einem Hashalgorithmus: Sie wollen ein Objekt mit Hilfe eines anderen Objektes nachschlagen. Das läßt sich allerdings auch mit einem `TreeMap`-Container bewerkstelligen oder Sie können das Interface *`Map`* selbst implementieren. Das folgende Beispiel wählt im Gegensatz zu einer hashbasierten Implementierung eine Lösung mit zwei `ArrayList`-Containern. Anders als *AssociativeArray.java* auf Seite 646 liefert das folgende Beispiel eine vollständige Implementierung des *`Map`*-Interfaces, darunter auch die Methode `entrySet()`:

```
//: containers/SlowMap.java
// A Map implemented with ArrayLists.
import java.util.*;
import net.mindview.util.*;

public class SlowMap<K,V> extends AbstractMap<K,V> {
    private List<K> keys = new ArrayList<K>();
    private List<V> values = new ArrayList<V>();
    public V put(K key, V value) {
        V oldValue = get(key); // The old value or null
        if(!keys.contains(key)) {
            keys.add(key);
            values.add(value);
        } else
            values.set(keys.indexOf(key), value);
        return oldValue;
    }
    public V get(Object key) { // key is type Object, not K
        if(!keys.contains(key))
            return null;
        return values.get(keys.indexOf(key));
    }
    public Set<Map.Entry<K,V>> entrySet() {
        Set<Map.Entry<K,V>> set= new HashSet<Map.Entry<K,V>>();
        Iterator<K> ki = keys.iterator();
        Iterator<V> vi = values.iterator();
```

```
        while(ki.hasNext())
            set.add(new MapEntry<K,V>(ki.next(), vi.next()));
        return set;
    }
    public static void main(String[] args) {
        SlowMap<String,String> m = new SlowMap<String,String>();
        m.putAll(Countries.capitals(15));
        System.out.println(m);
        System.out.println(m.get("BULGARIA"));
        System.out.println(m.entrySet());
    }
} /* Output:
      {CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE VERDE=Praia,
        ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN REPUBLIC=Bangui,
        BOTSWANA=Gaberone, BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
        EGYPT=Cairo, ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti}
      Sofia
      [CAMEROON=Yaounde, CHAD=N'djamena, CONGO=Brazzaville, CAPE VERDE=Praia,
        ALGERIA=Algiers, COMOROS=Moroni, CENTRAL AFRICAN REPUBLIC=Bangui,
        BOTSWANA=Gaberone, BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia,
        EGYPT=Cairo, ANGOLA=Luanda, BURKINA FASO=Ouagadougou, DJIBOUTI=Dijibouti]
  *///:~
```

Die `put()`-Methode setzt die Schlüssel und Werte in die entsprechenden `ArrayList`-Container ein. In Übereinstimmung mit dem Interface *`Map`* gibt `put()` den zuvor mit diesem Schlüssel verknüpften Wert zurück oder `null`, falls der Schlüssel nicht existiert.

[81] Die `get()`-Methode folgt ebenfalls der Spezifikation durch das Interface *`Map`*, indem sie `null` zurückgibt, falls der `SlowMap`-Container den übergebenen Schlüssel nicht enthält. Existiert der Schlüssel, so wird er verwendet, um den ganzzahligen Index seiner Position in dem von `keysList` referenzierten `ArrayList`-Container abzufragen. Der Index dient wiederum zum Abfragen des gespeicherten Wertes an der korrespondierenden Position in dem von `valuesList` referenzierten `ArrayList`-Container. Beachten Sie, daß das Argument `key` der `get()`-Methode, vielleicht entgegen Ihrer Erwartung, vom Typ `Object` ist, nicht vom Typ `K`. (Die `get()`-Methode im Beispiel *AssociativeArray.java* auf Seite 646 erwartet dagegen ein Argument vom Typ `K`.) Der Grund liegt bei der späten Aufnahme der generischen Typen in den Sprachumfang. Wären die generischen Typen von Anfang an Bestand der Programmiersprache gewesen, so könnte die `get()`-Methode den Typparameter als Typ ihres Argumentes wählen.

[82] Die im Interface *`Map`* deklarierte Methode `entrySet()` gibt einen *`Set`*-Container mit Elementen vom Typ *`Map.Entry`* zurück. Das Interface *`Map.Entry`* beschreibt allerdings eine implementierungsabhängige Struktur, das heißt wenn Sie eine eigene Implementierung des *`Map`*-Interfaces entwickeln, müssen Sie stets auch *`Map.Entry`* implementieren:

```
//: containers/MapEntry.java
// A simple Map.Entry for sample Map implementations.
import java.util.*;

public class MapEntry<K,V> implements Map.Entry<K,V> {
    private K key;
    private V value;
    public MapEntry(K key, V value) {
        this.key = key;
        this.value = value;
    }
    public K getKey() { return key; }
    public V getValue() { return value; }
```

```
        public V setValue(V v) {
            V result = value;
            value = v;
            return result;
        }
        public int hashCode() {
            return (key==null ? 0 : key.hashCode()) ^
                (value==null ? 0 : value.hashCode());
        }
        public boolean equals(Object o) {
            if(!(o instanceof MapEntry)) return false;
            MapEntry me = (MapEntry) o;
            return
                (key == null ?
                 me.getKey() == null : key.equals(me.getKey())) &&
                (value == null ?
                 me.getValue()== null : value.equals(me.getValue()));
        }
        public String toString() { return key + ''='' + value; }
    } ///:~
```

[83] Die Klasse `MapEntry` gespeichert die Schlüssel und Werte und gibt sie wieder heraus. Die `entrySet()`-Methode nutzt `MapEntry`, um einen *`Set`*-Container von Schlüssel/Wert-Paaren zu erzeugen. Beachten Sie, daß `entrySet()` einen `HashSet`-Container verwendet, um die Paare zu speichern und daß die Klasse `MapEntry` einfach die `hashCode()`-Methode der Schlüsselobjekte aufruft. Obwohl diese Lösung sehr einfach ist und im Rahmen des Tests in der `main()`-Methode des Beispiels *SlowMap.java* auf Seite 655 zu funktionieren scheint, ist die Implementierung nicht korrekt, da Schlüssel und Werte *kopiert* werden. Eine korrekte Implementierung liefert eine Sicht (*view*) in den *`Map`*-Container, statt einer Kopie, wobei diese Sicht, im Gegensatz zu einer Kopie, die Modifikation des ursprünglichen Containers gestattet. Übungsaufgabe 16 bietet eine Gelegenheit, diese Schwachstelle zu reparieren.

[84] Beachten Sie, daß die `equals()`-Methode der Klasse `MapEntry` sowohl Schlüssel als auch Werte prüfen muß. Die Bedeutung der Methode `hashCode()` wird im nächsten Unterabschnitt beschrieben. Die `String`-Darstellung des Inhalts des `SlowMap`-Containers erfolgt automatisch über die `toString()`-Methode der Klasse `AbstractMap`.

[85] Die `main()`-Methode der Klasse `SlowMap` lädt einen `SlowMap`-Container und zeigt seinen Inhalt an. Ein Aufruf der `get()`-Methode zeigt, daß ~~was~~ funktioniert?

Übungsaufgabe 15: (1) Wiederholen Sie Übungsaufgabe 13 (Seite 647) mit einem Container vom Typ `SlowMap`. ■

Übungsaufgabe 16: (7) Wenden Sie die Tests im Beispiel *Maps.java* (Seite 648) auf einen Container vom Typ `SlowMap` an und verifizieren Sie, daß ~~was~~ funktioniert. Korrigieren Sie alle Stellen in der Klasse `SlowMap`, die nicht korrekt funktionieren. ■

Übungsaufgabe 17: (2) Vervollständigen Sie die Implementierung des Interfaces *`Map`* durch die Klasse `SlowMap`. ■

Übungsaufgabe 18: (3) Schreiben Sie eine Klasse `SlowSet` (orientieren Sie sich dabei an der Klasse `SlowMap`). ■

18.9.2 Hashverfahren und Zugriffsgeschwindigkeit

[86] Das Beispiel *SlowMap.java* auf Seite 655 zeigt, daß es nicht allzu kompliziert ist, eine Implementierung des Interfaces `Map` zu schreiben. Andererseits ist die Klasse `SlowMap`, wie der Name bereits andeutet, nicht besonders schnell, so daß Sie sie wahrscheinlich nicht verwenden werden, wenn es eine Alternative gibt. Das Problem besteht im Nachschlagen des Schlüssels: `SlowMap` speichert die Schlüssel in keiner bestimmten Reihenfolge, so daß die Schlüsselmenge linear durchsucht werden muß. Die lineare Suche („sequentielle Suche") ist das langsamste Suchverfahren.

[87] Das Ziel aller Hashalgorithmen ist Geschwindigkeit: Hashbasierte Datenstrukturen gestatten schnelles Nachschlagen. Da die Schlüsselsuche einen Engpaß darstellt, besteht eine Lösung des Problems darin, die Schlüssel zu sortieren und die statische `Collections`-Methode `binarySearch()` zur Suche nach einem Schlüssel zu verwenden (Übungsaufgabe Nummer? führt Sie durch diesen Prozeß).

[88] Es genügt, wenn der Hashalgorithmus den Schlüssel *an irgend einer Stelle* deponiert, wo er schnell auffindbar ist. Das Array ist die schnellste Datenstruktur, in der eine Anzahl von Elementen gespeichert werden kann. Daher wird ein Array gewählt, um die Schlüssel*information* abzulegen (beachten Sie die Wortwahl „Schlüsselinformation" anstelle von „Schlüssel"). Ein Problem ist dabei allerdings, daß sich die Kapazität eines Arrays nicht verändern läßt: Ein Container vom Typ `Map` soll eine unbestimmte Anzahl von Schlüssel/Wert-Paaren speichern können. Wie läßt sich diese Anforderung mit einer festen Anzahl von Schlüsseln in einem Array in Einklang bringen?

[89] Die Lösung besteht darin, daß das Array nicht die Schlüssel selbst enthält. Aus dem Schlüsselobjekt wird eine Zahl ermittelt, aus der der Arrayindex berechnet wird (siehe Erläuterung im Anschluß an das nächste Beispiel). Diese Zahl ist der sogenannte *Hashwert* und wird von der Methode `hashCode()` geliefert (im Jargon der Informatik ist `hashCode()` eine *Hashfunktion*), die in der Klasse `Object` definiert und in Ihre eigenen Klasse voraussichtlich überschrieben ist.

[90] Das Problem der festen Kapazität des Arrays wird dadurch gelöst, daß mehr als ein Schlüssel denselben Index liefern darf, also sogenannte *Kollisionen* erlaubt werden. Somit kommt es nicht auf die Länge des Arrays an, da jeder Hashwert einen Platz im Array erhält.

[91] Der Prozeß des Nachschlagens eines Wertes beginnt also mit der Berechnung des Hashwertes aus dem Schlüssel und dessen Umwandlung in den Arrayindex. Eine sogenannte *perfekte Hashfunktion*[7] ist ein Sonderfall und liegt vor, wenn Sie garantieren können, daß es keine Kollisionen gibt (möglich, bei fester Anzahl von Werten). In allen übrigen Fällen, werden Kollisionen mittels *externer Verkettung* (*external chaining*) behandelt, das heißt das Array verweist nicht direkt auf den Wert, sondern auf eine Liste von Schlüsseln, welche linear durchsucht und mit Hilfe der `equals()`-Methode geprüft werden. Dieser Teil der Suche ist natürlich erheblich langsamer, aber bei einer guten Hashfunktion bestehen diese Listen nur aus wenigen Schlüsseln. Statt die gesamte Liste von Werten zu durchsuchen, springt der Hashalgorithmus zu einer kleinen vorsortierten Teilmenge, deren Schlüssel mit dem angeforderten Schlüssel verglichen werden müssen, um den endgültigen Wert zu finden. Dieses Verfahren ist viel schneller und begründet die Zugriffsgeschwindigkeit des Containertyps `HashMap`.

[92] Mit diesen Grundlagen über Hashalgorithmen können Sie eine einfache hashbasierte Implementierung des Interfaces `Map` schreiben:

```
//: containers/SimpleHashMap.java
// A demonstration hashed Map.
import java.util.*;
import net.mindview.util.*;
```

[7] Die seit der SE 5 definierten Klassen `EnumMap` und `EnumSet` implementieren perfekte Hashfunktionen, da ein Aufzählungstyp stets eine feste Anzahl von Werte definiert (siehe Kapitel 20).

```
public class SimpleHashMap<K,V> extends AbstractMap<K,V> {
    // Choose a prime number for the hash table
    // size, to achieve a uniform distribution:
    static final int SIZE = 997;
    // You can't have a physical array of generics,
    // but you can upcast to one:
    @SuppressWarnings("unchecked")
    LinkedList<MapEntry<K,V>>[] buckets = new LinkedList[SIZE];
    public V put(K key, V value) {
        V oldValue = null;
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null)
            buckets[index] = new LinkedList<MapEntry<K,V>>();
        LinkedList<MapEntry<K,V>> bucket = buckets[index];
        MapEntry<K,V> pair = new MapEntry<K,V>(key, value);
        boolean found = false;
        ListIterator<MapEntry<K,V>> it = bucket.listIterator();
        while(it.hasNext()) {
            MapEntry<K,V> iPair = it.next();
            if(iPair.getKey().equals(key)) {
                oldValue = iPair.getValue();
                it.set(pair); // Replace old with new
                found = true;
                break;
            }
        }
        if(!found)
            buckets[index].add(pair);
        return oldValue;
    }
    public V get(Object key) {
        int index = Math.abs(key.hashCode()) % SIZE;
        if(buckets[index] == null) return null;
        for(MapEntry<K,V> iPair : buckets[index])
            if(iPair.getKey().equals(key))
                return iPair.getValue();
        return null;
    }
    public Set<Map.Entry<K,V>> entrySet() {
        Set<Map.Entry<K,V>> set = new HashSet<Map.Entry<K,V>>();
        for(LinkedList<MapEntry<K,V>> bucket : buckets) {
            if(bucket == null) continue;
            for(MapEntry<K,V> mpair : bucket)
                set.add(mpair);
        }
        return set;
    }
    public static void main(String[] args) {
        SimpleHashMap<String,String> m = new SimpleHashMap<String,String>();
        m.putAll(Countries.capitals(25));
        System.out.println(m);
        System.out.println(m.get("ERITREA"));
        System.out.println(m.entrySet());
    }
} /* Output:
      {CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena,
        COTE D'IVOIR (IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN REPUBLIC=Bangui,
```

```
        GUINEA=Conakry, BOTSWANA=Gaberone, BISSAU=Bissau, EGYPT=Cairo,
        ANGOLA=Luanda, BURKINA FASO=Ouagadougou, ERITREA=Asmara,
        THE GAMBIA=Banjul, KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia,
        ALGERIA=Algiers, COMOROS=Moroni, EQUATORIAL GUINEA=Malabo,
        BURUNDI=Bujumbura, BENIN=Porto-Novo, BULGARIA=Sofia, GHANA=Accra,
        DJIBOUTI=Dijibouti, ETHIOPIA=Addis Ababa}
        Asmara
      [CAMEROON=Yaounde, CONGO=Brazzaville, CHAD=N'djamena,
        COTE D'IVOIR (IVORY COAST)=Yamoussoukro, CENTRAL AFRICAN REPUBLIC=Bangui,
        GUINEA=Conakry, BOTSWANA=Gaberone, BISSAU=Bissau, EGYPT=Cairo,
        ANGOLA=Luanda, BURKINA FASO=Ouagadougou, ERITREA=Asmara, THE GAMBIA=Banjul,
        KENYA=Nairobi, GABON=Libreville, CAPE VERDE=Praia, ALGERIA=Algiers,
        COMOROS=Moroni, EQUATORIAL GUINEA=Malabo, BURUNDI=Bujumbura,
        BENIN=Porto-Novo, BULGARIA=Sofia, GHANA=Accra, DJIBOUTI=Dijibouti,
        ETHIOPIA=Addis Ababa]
    *///:~
```

[93] Da die „vorsortierten Teilmengen“ in der Hashtabelle häufig als „Buckets“ bezeichnet werden, heißt die Referenzvariable, die auf die Hashtabelle verweist `buckets`. Die Anzahl der Buckets ist typischerweise eine Primzahl, um eine gleichmäßige Verteilung der Schlüssel zu begünstigen.[8] Beachten Sie, daß das Beispiel ein Array von `LinkedList`-Containern verwendet, also Kollisionen einplant: Jeder neue Eintrag wird einfach an das Ende des Containers im entsprechenden Bucket angehängt. Java erlaubt zwar nicht, ein Array von Elementen generischen Typs zu erzeugen, wohl aber, eine Referenzvariable für ein solches Array zu deklarieren. Here, it is convenient to upcast to such an array, to prevent extra casting later in the code.

[94] Die `put()`-Methode ruft die `hashCode()`-Methode des Schlüssels auf und entfernt das eventuell vorhandene Vorzeichen des Hashwertes. Damit der Hashwert in das von `buckets` referenzierte Array paßt, wird der Divisionsrest bezüglich der Kapazität des Arrays bestimmt. Enthält die so bestimmte Speicherstelle des Arrays den Wert `null`, so wurde in diesem Bucket noch kein Schlüssel/Wert-Paar gespeichert und es wird ein neuer `LinkedList`-Container erzeugt, welcher das diesem Bucket zugeteilte Paar enthält. Der normale Vorgang besteht darin, den `LinkedList`-Container des Buckets nach dem angeforderten Schlüssel zu durchsuchen und im Erfolgsfall den alten Wert in der lokalen Variablen `oldValue` zu sichern sowie durch den neuen Wert zu ersetzen. Der Schalter `found` gibt an, ob ein altes Schlüssel/Wert-Paar vorhanden war und hängt das neue Paar andernfalls an das Ende des Containers an.

[95] Die `get()`-Methode berechnet den Index in dem von `buckets` referenzierten Array auf dieselbe Weise wie die `put()`-Methode (das ist wesentlich, damit Sie garantiert zum selben Bucket kommen). Existiert ein `LinkedList`-Container, so wird er nach einem übereinstimmenden Schlüssel durchsucht.

[96] Beachten Sie, daß diese Implementierung nicht auf Performanz ausgerichtet ist, sondern nur die Vorgänge in der Hashtabelle verdeutlich soll. Der Quelltext der Klasse `java.util.HashMap` ist ein Beispiel für eine fein abgestimmte Implementierung. Die Klasse `SimpleHashMap` verwendet zur Vereinfachung bei `entrySet()` den Ansatz von `SlowMap`, einen übermäßig vereinfachten Ansatz, der bei einer universellen *`Map`*-Implementierung nicht funktioniert.

Übungsaufgabe 19: (1) Wiederholen Sie Übungsaufgabe 13 (Seite 647) mit der Klasse `Simple-`

[8] Es hat sich herausgestellt, daß Primzahlen keine ideale Anzahl von Buckets liefern. Die neueren hashbasierten Implementierungen in der Standardbibliothek von Java verwenden, nach ausgiebigen Tests, Potenzen von 2. Die Division beziehungsweise Berechnung des Divisionsrestes ist bei modernen Prozessoren eine der langsamsten Operationen. Bei einer Tabellengröße von 2^n kann der Index mit Hilfe einer Bitmaske statt einer Division bestimmt werden. Da die `get()`-Methode mit Abstand die häufigste Operation ist, verursacht der Divisionsrestoperator (`%`) einen großen Anteil der Unkosten. Dieser Anteil verschwindet bei einer Tabellengröße von 2^n (kann sich aber auf einige `hashCode()`-Methoden wie? auswirken).

`HashMap`. ■

Übungsaufgabe 20: (3) Ändern Sie die Klasse `SimpleHashMap`, so daß sie Kollisionen protokolliert. Testen Sie Ihre Lösung, indem Sie dieselben Datensätze zweimal in die Datenstruktur aufnehmen, um Kollisionen herbeizuführen. ■

Übungsaufgabe 21: (2) Ändern Sie die Klasse `SimpleHashMap`, so daß sie beim Auftreten von Kollisionen die Anzahl der Sondierungen protokolliert, das heißt wie oft die `next()`-Methode des Iterators aufgerufen werden muß, der den Inhalt eines `LinkedList`-Containers auf der Suche nach Übereinstimmungen durchläuft. ■

Übungsaufgabe 22: (2) Implementieren Sie in der Klasse `SimpleHashMap` die Methoden `clear()` und `remove()`. ■

Übungsaufgabe 23: (3) Vervollständigen Sie die Implementierung des Interfaces *`Map`* durch die Klasse `SimpleHashMap`. ■

Übungsaufgabe 24: (5) Schreiben Sie eine Klasse `SimpleHashSet` (orientieren Sie sich dabei an der Klasse `SimpleHashMap`). ■

Übungsaufgabe 25: (6) Statt einen Listeniterator zu verwenden, ändern Sie die Klasse `MapEntry` so, daß sie eine in sich geschlossene verkette Liste bildet (jedes `MapEntry`-Objekt hat einen Verweis auf das nächste `MapEntry`-Objekt in Vorwärtsrichtung). Überarbeiten Sie den Rest des Beispiels *SimpleHashMap.java*, damit das Programm mit der Änderungen korrekt funktioniert. ■

18.9.3 Überschreiben der Methode hashCode()

[97] Nachdem Sie verstanden haben, wie Hashalgorithmen funktionieren, können Sie daran gehen, eigene `hashCode()`-Methoden zu schreiben.

[98] Die Berechnung des tatsächlichen Arraysindexes aus dem Hashwert hängt von der Kapazität des jeweiligen `HashMap`-Containers ab, welche wiederum vom Füllgrad des Containers und dem Ladefaktor (der Begriff „Ladefaktor“ wird in Unterunterabschnitt 18.10.5.1 erklärt) abhängt. Der Rückgabewert Ihrer `hashCode()`-Methode durchläuft also noch eine Umwandlung, die den endgültigen Arrayindex liefert (die Klasse `SimpleHashMap` berechnet beispielsweise den Divisionsrest bezüglich der Arraylänge).

[99] Die wichtigste Eigenschaft der `hashCode()`-Methode ist, daß sie unabhängig von Zeitpunkt und Anzahl ihrer Aufrufe auf einem bestimmten Objekt, stets ein und denselben Rückgabewert liefert. Gibt Ihre `hashCode()`-Methode beim Einsetzen eines Schlüssel/Wert-Paares in einen `HashMap`-Container in der `put()`-Methode einen anderen Hashwert zurück, als beim Nachschlagen des entsprechenden Schlüssels in der `get()`-Methode, so sind Sie nicht in der Lage, den mit diesem Schlüssel verknüpften Wert abzufragen. Hängt der Rückgabewert Ihrer `hashCode()`-Methode von veränderlichen Eigenschaften ihres Objektes ab, so muß der Anwender darüber aufgeklärt werden, das eine Änderung dieser Eigenschaften zu einem anderen Arrayindex führt, weil die `hashCode()`-Methode einen abweichenden Hashwert liefert.

[100] Es ist nicht sinnvoll, den Hashwert aus eindeutigen Eigenschaften eines Objektes zu berechnen. Insbesondere liefert der Wert der Selbstreferenz `this` einen schlechten Hashwert, da Sie keinen neuen Schlüssel erzeugen können, der mit dem Schlüssel eines per `put()` gespeicherten Paares übereinstimmt. Beispiel *SpringDetector.java* auf Seite 652 hat dieses Problem gezeigt, da die `Object`-Version der `hashCode()`-Methode die Speicheradresse ihres Objektes auswertet. Verwenden Sie also Eigenschaften, die das Objekt sinnvoll identifizieren.

[101] Die Klasse `String` liefert ein Beispiel: Verwendet ein Programm mehrere `String`-Objekte, die identische Zeichenketten enthalten, so verweisen alle entsprechenden Referenzvariablen auf dieselbe Speicherstelle. Die `hashCode()`-Methode der Klasse `String` gibt bei zwei Objekten, die die Zeichenkette „hello“ enthalten, sinnvollerweise übereinstimmende Hashwerte zurück, wie das folgende Beispiel zeigt:

```
//: containers/StringHashCode.java
public class StringHashCode {
    public static void main(String[] args) {
        String[] hellos = ''Hello Hello''.split('' '');
        System.out.println(hellos[0].hashCode());
        System.out.println(hellos[1].hashCode());
    }
} /* Output: (Sample)
      69609650
      69609650
   *///:~
```

[102] Die `hashCode()`-Methode der Klasse `String` basiert offensichtlich auf dem Inhalt des `String`-Objektes.

[103] Eine effektive `hashCode()`-Methode muß sowohl schnell als auch sinnvoll sein, den Hashwert also anhand des Inhaltes eines Objektes generieren. Denken Sie daran, daß der Hashwert nicht eindeutig zu sein braucht (Geschwindigkeit hat Vorrang vor Eindeutigkeit), die Identität von Objekten aber zwischen `hashCode()` und `equals()` vollständig aufgelöst werden muß.

[104] Da der Hashwert noch in den Arrayindex umgewandelt wird, ist die Breite des Wertebereiches der `hashCode()`-Methode unwesentlich. Es kommt nur darauf an, daß `hashCode()` einen `int`-Wert zurückgibt.

[105] Eine gute `hashCode()`-Methode sollte gleichmäßig verteilte Werte liefern. Neigen die Werte zu Häufungspunkten, so sind die Hashtabellen von `HashMap`- oder `HashSet`-Containern in einigen Bereichen stärker ausgelastet und somit langsamer, als bei einer Hashfunktion die eine gleichmäßige Verteilung bewirkt.

[106] Joshua Bloch gibt in seinem Buch *Effective Java Language Programming Guide*, Addison-Wesley (2001) das folgende einfache Rezept für eine anständige `hashCode()`-Methode an:

- Speichern Sie einen konstanten, von Null verschiedenen Wert in einer `int`-Variablen namens `result`, zum Beispiel 17.
- Berechnen Sie für jedes signifikante Feld Ihres Objektes, das heißt für jedes Feld, das die `equals()`-Methode auswertet, nach dem folgenden Schema einen Hashwert `c` vom Typ `int`:
  - `boolean`: `c = (f ? 0 : 1)`
  - `byte`, `char`, `short` oder `int`: `c = (int) f`
  - `long`: `c = (int) (f ^ (f >>> 32))`
  - `float`: `c = Float.floatToIntBits(f);`
  - `double`: `long l = Double.doubleToLongBits(f); c = (int) (l ^ (l >>> 32));`
  - `Object`, wobei `equals()` die `equals()`-Methode dieses Feldes aufruft: `c = f.hashCode()`
  - Array: Wenden Sie die obigen Regeln auf jedes Element an.
- Kombinieren Sie die oben berechneten Hashwerte wie folgt: `result = 37 * result + c;`.

- Geben Sie `result` zurück.
- Testen Sie die so entstandene `hashCode()`-Methode und vergewissern Sie sich, daß identische Objekte identische Hashwerte liefern.

[107] Das nächste Beispiel befolgt diese Richtlinien:

```
//: containers/CountedString.java
// Creating a good hashCode().
import java.util.*;
import static net.mindview.util.Print.*;

public class CountedString {
    private static List<String> created =
        new ArrayList<String>();
    private String s;
    private int id = 0;
    public CountedString(String str) {
        s = str;
        created.add(s);
        // id is the total number of instances
        // of this string in use by CountedString:
        for(String s2 : created)
            if(s2.equals(s))
                id++;
    }
    public String toString() {
        return ''String: '' + s + '' id: '' + id +
            '' hashCode(): '' + hashCode();
    }
    public int hashCode() {
        // The very simple approach:
        // return s.hashCode() * id;
        // Using Joshua Bloch's recipe:
        int result = 17;
        result = 37 * result + s.hashCode();
        result = 37 * result + id;
        return result;
    }
    public boolean equals(Object o) {
        return o instanceof CountedString &&
            s.equals(((CountedString) o).s) &&
            id == ((CountedString) o).id;
    }
    public static void main(String[] args) {
        Map<CountedString,Integer> map =
            new HashMap<CountedString,Integer>();
        CountedString[] cs = new CountedString[5];
        for(int i = 0; i < cs.length; i++) {
            cs[i] = new CountedString(''hi'');
            map.put(cs[i], i); // Autobox int -> Integer
        }
        print(map);
        for(CountedString cstring : cs) {
            print(''Looking up '' + cstring);
            print(map.get(cstring));
        }
    }
```

```
} /* Output: (Sample)
      {String: hi id: 4 hashCode(): 146450=3, String: hi id: 1 hashCode(): 146447=0,
        String: hi id: 3 hashCode(): 146449=2, String: hi id: 5 hashCode(): 146451=4
        String: hi id: 2 hashCode(): 146448=1}
      Looking up String: hi id: 1 hashCode(): 146447
      0
      Looking up String: hi id: 2 hashCode(): 146448
      1
      Looking up String: hi id: 3 hashCode(): 146449
      2
      Looking up String: hi id: 4 hashCode(): 146450
      3
      Looking up String: hi id: 5 hashCode(): 146451
      4
   *///:~
```

Die Klasse `CountedString` enthält ein `String`- und ein `id`-Feld. Das `id`-Feld gibt an, wieviele `CountedString`-Objekte in ihren `String`-Feldern übereinstimmende Werte haben. Die Zählung findet im Konstruktor statt, der pro Aufruf einmal den statischen `ArrayList`-Container mit den gespeicherten `String`-Objekten durchläuft.

[108] Die Methoden `hashCode()` und `equals()` ermitteln ihre Rückgabewerte anhand der beiden Felder `s` und `id`. Würden `hashCode()` und `equals()` nur eines der Felder `s` und `id` auswerten, so würden die Methoden trotz unterschiedlicher Werte Treffer anzeigen.

[109] Die `main()`-Methode erzeugt mehrere `CountedString`-Objekte, welche dieselbe Zeichenkette enthalten, um zu zeigen, daß doppelte Zeichenketten durch das `id`-Feld eindeutige Hashwerte bekommen. Der `HashMap`-Container wird ausgegeben, damit Sie die interne Anordnung der Paare sehen (keine erkennbare Reihenfolge). Anschließend wird jeder Schlüssel einzeln abgefragt, damit Sie nachvollziehen können, daß der Mechanismus zum Nachschlagen funktioniert.

[110] Die Basisklasse `Individual` der `typeinfo.pets`-Bibliothek aus Abschnitt 15.3 liefert ein zweites Beispiel. Die Klasse wurde dort zwar verwendet, ihre Definition aber bis zu dieser Stelle verzögert, damit Sie die Implementierung verstehen können:

```
//: typeinfo/pets/Individual.java
package typeinfo.pets;

public class Individual implements Comparable<Individual> {
    private static long counter = 0;
    private final long id = counter++;
    private String name;
    public Individual(String name) { this.name = name; }
    // 'name' is optional:
    public Individual() {}
    public String toString() {
        return getClass().getSimpleName() +
            (name == null ? "" : " " + name);
    }
    public long id() { return id; }
    public boolean equals(Object o) {
        return o instanceof Individual &&
            id == ((Individual) o).id;
    }
    public int hashCode() {
        int result = 17;
        if(name != null)
            result = 37 * result + name.hashCode();
```

```
        result = 37 * result + (int) id;
        return result;
    }
    public int compareTo(Individual arg) {
        // Compare by class name first:
        String first = getClass().getSimpleName();
        String argFirst = arg.getClass().getSimpleName();
        int firstCompare = first.compareTo(argFirst);
        if(firstCompare != 0)
            return firstCompare;
        if(name != null && arg.name != null) {
            int secondCompare = name.compareTo(arg.name);
            if(secondCompare != 0)
                return secondCompare;
        }
        return (arg.id < id ? -1 : (arg.id == id ? 0 : 1));
    }
} ///:~
```

[111] Die `compareTo()`-Methode verwendet eine Hierarchie von Vergleichskriterien: Die Elemente werden zuerst nach ihrem tatsächlichen Typ sortiert, dann nach dem Inhalt ihres `name`-Feldes (falls bewertet) und schließlich nach der Einsetzungsreihenfolge. Das folgende Beispiel zeigt, wie `compareTo()` funktioniert:

```
//: containers/IndividualTest.java
import holding.MapOfList;
import typeinfo.pets.*;
import java.util.*;

public class IndividualTest {
    public static void main(String[] args) {
        Set<Individual> pets = new TreeSet<Individual>();
        for(List<? extends Pet> lp: MapOfList.petPeople.values())
            for(Pet p : lp)
                pets.add(p);
        System.out.println(pets);
    }
} /* Output:
      [Cat Elsie May, Cat Pinkola, Cat Shackleton, Cat Stanford aka Stinky el Negro,
      Cymric Molly, Dog Margrett, Mutt Spot, Pug Louie aka Louis Snorkelstein Dupree,
      Rat Fizzy, Rat Freckly, Rat Fuzzy]
   *///:~
```

Alle Haustiere haben einen Namen, so daß sie zunächst nach Tierart und anschließend nach Namen sortiert werden.

[112] Es kann kompliziert sein, passende `hashCode()`- und `equals()`-Methoden für eine neue Klasse zu schreiben. Sie finden in der Kategorie *lang* des „Jakarta Commons"-Projektes von Apache (unter der Webadresse *http://jakarta.apache.org/commons*) Hilfsmittel für diesen Zweck. Das „Jakarta Commons"-Projekt hat noch viele andere potentiell nützliche Bibliotheken und scheint die Antwort der Java-Community auf das Boost-Projekt (*http://www.boost.org*) der C++-Community zu sein.

Übungsaufgabe 26: (2) Legen Sie in der Klasse `CountedString` ein `char`-Feld an, welches vom Konstruktor initialisiert wird und ändern Sie die Methoden `hashCode()` und `equals()`, so daß sie das `char`-Feld auswerten. ■

Übungsaufgabe 27: (3) Ändern Sie die `hashCode()`-Methode im Beispiel *CountedString.java* (Seite 663), indem Sie die Kombination mit dem `id`-Feld entfernen. Zeigen Sie, daß Objekte der Klasse

`CountedString` noch immer als Schlüssel verwendet werden können. Worin besteht das Problem bei diesem Ansatz? ■

Übungsaufgabe 28: (4) Ändern Sie das Beispiel *net/mindview/util/Tuple.java* (Seite 488) in eine universell verwendbare Klasse, indem Sie für jede Tupelklasse die Methoden `hashCode()` und `equals()` anlegen sowie das Interface *`Comparable`* implementieren. ■

18.10 Entscheidung für eine Implementierung

[113] Sie wissen nun, daß die Containerbibliothek von Java die vier fundamentalen Containertypen *`Map`*, *`List`*, *`Set`* und *`Queue`* definiert, zu denen es jeweils mehr als eine Implementierung gibt. Wie können Sie entscheiden, welche Implementierung Sie wählen sollten, wenn Sie die Funktionalität eines dieser vier Containertypen brauchen?

[114] Jede einzelne Implementierung hat ihre eigenen Fähigkeiten, Stärken und Schwächen. Beispielsweise können Sie der Abbildung am Kapitelanfang entnehmen, daß die „Eigenschaft“ der Klassen `Hashtable`, `Vector` und `Stack` darin besteht, daß Sie veraltet sind, einen älteren Quelltext also nicht funktionsuntüchtig machen (am besten verwenden Sie diese drei Klassen in neuem Quelltext nicht mehr).

[115] Die verschiedenen Containertypen unter dem Interface *`Queue`* unterscheiden sich nur durch die Art und Weise, in der sie Werte annehmen und herausgeben (die Bedeutung dieser Klassen zeigt sich in Kapitel 22).

[116] Die Unterscheidung zwischen Containertypen richtet sich häufig nach der Datenstruktur, auf die sich der Containertyp stützt, das heißt die Datenstruktur, die das jeweilige Interface physikalisch implementiert. Beispielsweise haben die Klassen `ArrayList` und `LinkedList` dieselben grundlegenden Methoden des Containertyps *`List`*, da beide Klassen das Interface *`List`* implementieren. Die Klasse `ArrayList` bezieht sich aber auf ein Array, während `LinkedList` als doppelt verkettete Liste implementiert ist, also aus einzelnen Objekten besteht, die sowohl Daten enthalten, als auch Referenzen auf das vorige und das nächste Objekt. Wenn Sie viele Einsetzungs- oder Löschoperationen in der Mitte des Containers benötigen, ist `LinkedList` die passende Wahl. (Die Klasse `LinkedList` verfügt über zusätzliche Funktionalität, die aus der Klasse `AbstractSequentialList` stammt.) Andernfalls ist ein Container vom Typ `ArrayList` typischerweise schneller.

[117] Das Interface *`Set`* wird, um ein weiteres Beispiel zu geben, von den Klassen `TreeSet`, `HashSet` und `LinkedHashSet` implementiert,[9] die jeweils ein eigenes Verhalten haben: Die Klasse `HashSet` ist die Standardimplementierung und bietet schnellen Zugriff, während die Klasse `LinkedHashSet` die Einsetzungsreihenfolge ihrer Elemente erhält. Die Klasse `TreeSet` stützt sich auf einen `TreeMap`-Container und liefert eine stets sortierte (*constantly sorted*) Menge. Welche Implementierung Sie wählen, hängt von Ihren Anforderungen ab.

[118] Zuweilen verfügen verschiedene Implementierungen eines bestimmten Containerinterfaces über dieselben Methoden, unterscheiden sich aber in deren Performanz. In diesem Fall sollten Sie die Implementierung danach auswählen, wie häufig Sie eine bestimmte Methoden aufrufen und wie schnell der Methodenaufruf verarbeitet werden muß. In einem solchen Fall stellen sich die Unterschiede zwischen verschiedenen Implementierungen im Rahmen eines Performanztests heraus.

[9] Sowie von den Klassen `EnumMap` und `CopyOnWriteArraySet`, die aber Spezialfälle sind. Es gibt unter Umständen weitere spezielle Implementierungen der Containerinterfaces, aber dieser Abschnitt betrachtet die allgemeineren Fälle.

18.10.1 Eine kleine Bibliothek für Performanztests

[119] Ich habe die Grundfunktionalität des Testablaufs in einer kleinen Bibliothek erfaßt, um das Kopieren von Quelltext zu vermeiden und Konsistenz zwischen den Tests zu ermöglichen. Die Bibliothek besteht aus einer Basisklasse, von der Sie eine Anzahl anonymer innerer Klassen ableiten können, je eine pro Test. Jede dieser inneren Klassen wird im Rahmen des Testablaufs aufgerufen. Dieser Ansatz gestattet Ihnen, mühelos neue Tests hinzuzufügen oder ältere Tests zu entfernen.

[120] Die Testbibliothek ist ein weiteres Beispiel für das *Templatemethod*-Entwurfsmuster. Obwohl Sie dem typischen Ansatz dieses Entwurfsmuster folgen und die `Test`-Methode `test()` für jeden einzelnen Test überschreiben, liegt der Kern des Quelltextes (der Teil, der sich nicht ändert) in der separaten Klasse `Tester`.[10] Der Typparameter `C` repräsentiert den Typ des getesteten Containers:

```
//: containers/Test.java
// Framework for performing timed tests of containers.

public abstract class Test<C> {
    String name;
    public Test(String name) { this.name = name; }
    // Override this method for different tests.
    // Returns actual number of repetitions of test.
    abstract int test(C container, TestParam tp);
} ///:~
```

Jedes `Test`-Objekt speichert den Namen dieses Tests. Die `test()`-Methode erwartet den zu testenden Container sowie ein „Transferobjekt“ mit den verschiedenen Parametern des spezifischen Tests, darunter die Parameter `size` und `loops`, welche die Anzahl der Elemente im Container beziehungsweise die Anzahl der Wiederholungen des Tests angeben. Diese Parameter werden nicht in jedem Test benötigt.

[121] Jeder Container wird einer Reihe von Aufrufen der `test()`-Methode unterzogen, jeweils mit einem anderen `TestParam`-„Transferobjekt“. Die Klasse `TestParam` enthält daher zwei statische `array()`-Methoden, mit denen Sie einfach Arrays von `TestParam`-Objekten erzeugen können. Die erste Version der `array()`-Methode erwartet eine Argumentliste variabler Länge aus einander abwechselnden Element- und Wiederholungsanzahlen. Die zweite Version erwartet dieselbe Art von Liste, wobei die Werte diesesmal in `String`-Objekten verpackt sind, um Kommandozeilenargumente parsen zu können:

```
//: containers/TestParam.java
// A ''data transfer object.''

public class TestParam {
    public final int size;
    public final int loops;
    public TestParam(int size, int loops) {
        this.size = size;
        this.loops = loops;
    }
    // Create an array of TestParam from a varargs sequence:
    public static TestParam[] array(int... values) {
        int size = values.length/2;
        TestParam[] result = new TestParam[size];
        int n = 0;
        for(int i = 0; i < size; i++)
            result[i] = new TestParam(values[n++], values[n++]);
        return result;
```

[10] Krzysztof Sobolewski hat mir geholfen, die generischen Typen für dieses Beispiel auszuknobeln.

```
    }
    // Convert a String array to a TestParam array:
    public static TestParam[] array(String[] values) {
        int[] vals = new int[values.length];
        for(int i = 0; i < vals.length; i++)
            vals[i] = Integer.decode(values[i]);
        return array(vals);
    }
} ///:~
```

[122] Zur Anwendung dieser kleinen Bibliothek übergeben Sie den zu testenden Container zusammen mit einem *List*-Container von `Test`-Objekten der `Tester`-Methode `run()`. (Die `run()`-Methode ist eine überladene generische Methode, die den zum Aufruf erforderlichen Tippaufwand verringert.) Die `run()`-Methode ruft den passenden überladenen Konstruktor und anschließend die `timedTest()`-Methode auf, die jeden Test in der Liste zu diesem Container ausführt. Die `timedTest()`-Methode wiederholt jeden Test für jedes `TestParam`-Objekt in dem von `paramList` referenzierten `TestParam`-Array. Da `paramList` über das von `defaultParam` referenzierte statische Array initialisiert wird, können Sie `paramList` entweder für alle Tests global ändern, indem Sie der Referenzvariablen ein anderes Array zuweisen, oder für jeden einzelnen Test eine individuelle Parameterliste übergeben:

```
//: containers/Tester.java
// Applies Test objects to lists of different containers.
import java.util.*;

public class Tester<C> {
    public static int fieldWidth = 8;
    public static TestParam[] defaultParams =
        TestParam.array(10, 5000, 100, 5000, 1000, 5000, 10000, 500);
    // Override this to modify pre-test initialization:
    protected C initialize(int size) { return container; }
    protected C container;
    private String headline = '''';
    private List<Test<C>> tests;
    private static String stringField() {
        return ''%'' + fieldWidth + ''s'';
    }
    private static String numberField() {
        return ''%'' + fieldWidth + ''d'';
    }
    private static int sizeWidth = 5;
    private static String sizeField = ''%'' + sizeWidth + ''s'';
    private TestParam[] paramList = defaultParams;
    public Tester(C container, List<Test<C>> tests) {
        this.container = container;
        this.tests = tests;
        if(container != null)
            headline = container.getClass().getSimpleName();
    }
    public Tester(C container, List<Test<C>> tests, TestParam[] paramList) {
        this(container, tests);
        this.paramList = paramList;
    }
    public void setHeadline(String newHeadline) {
        headline = newHeadline;
    }
    // Generic methods for convenience :
```

```
    public static <C> void run(C cntnr, List<Test<C>> tests){
        new Tester<C>(cntnr, tests).timedTest();
    }
    public static <C> void
        run(C cntnr, List<Test<C>> tests, TestParam[] paramList) {
        new Tester<C>(cntnr, tests, paramList).timedTest();
    }
    private void displayHeader() {
        // Calculate width and pad with '-':
        int width = fieldWidth * tests.size() + sizeWidth;
        int dashLength = width - headline.length() - 1;
        StringBuilder head = new StringBuilder(width);
        for(int i = 0; i < dashLength/2; i++)
            head.append('-');
        head.append(' ');
        head.append(headline);
        head.append(' ');
        for(int i = 0; i < dashLength/2; i++)
            head.append('-');
        System.out.println(head);
        // Print column headers:
        System.out.format(sizeField, "size");
        for(Test test : tests)
            System.out.format(stringField(), test.name);
        System.out.println();
    }
    // Run the tests for this container:
    public void timedTest() {
        displayHeader();
        for(TestParam param : paramList) {
            System.out.format(sizeField, param.size);
            for(Test<C> test : tests) {
                C kontainer = initialize(param.size);
                long start = System.nanoTime();
                // Call the overriden method:
                int reps = test.test(kontainer, param);
                long duration = System.nanoTime() - start;
                long timePerRep = duration / reps; // Nanoseconds
                System.out.format(numberField(), timePerRep);
            }
            System.out.println();
        }
    }
} ///:~
```

Die Methoden `stringField()` und `numberField()` liefern Formatierungsausdrücke zur formatierten Ausgabe der Ergebnisse. Die voreingestelle Formatierungsbreite kann über das statische `fieldWidth`-Feld geändert werden. Die Methode `displayHeader()` gibt zu jedem Test einen formatierten Kopfbereich aus.

[123] Wenn Sie spezielle Anforderung an die Initialisierung haben, können Sie die Methode `initialize()` überschreiben. Diese Methode liefert einen initialisierten Container von appropriate Kapazität. Sie können entweder den vorhandenen Container modifizieren oder einen neuen erzeugen. In der `timedTest()`-Methode sehen Sie, daß der Rückgabewert von `initialize()` in einer lokalen Variablen namens `kontainer` deponiert wird, which allows you to replace the stored member container with a completely different initialized container.

[124] Der Rückgabewert der `Test`-Methode `test()` ist die Anzahl der in diesem Test durchgeführten Methodenaufrufe und wird verwendet, um die je Methodenaufruf verbrauchte Zeit in Nanosekunden zu messen. Beachten Sie, daß die statische `System`-Methode `nanoTime()` typischerweise Werte mit einer Granularität größer Eins liefert (diese Granularität hängt von Rechnerarchitektur und Betriebssystem ab) und somit einen gewissen Anteil an rattle im Ergebnis bewirkt.

[125] Die Ergebnisse können von einem Rechner zum anderen abweichen. Die Tests sind nur dazu da, um die Performanz der verschiedenen Container vergleichen zu können.

18.10.2 Implementierungen des Interfaces List

[126] Das folgende Programm ist ein Performanztest für die wichtigsten im Interface *`List`* deklarierten Methoden. Zum Vergleich werden auch die wichtigsten Methoden des Interfaces *`Queue`* getestet. Jede Containerklasse wird zwei separaten Testreihen unterzogen. Die *`Queue`*-Methoden werden anhand eines Containers vom Typ `LinkedList` getestet:

```
//: containers/ListPerformance.java
// Demonstrates performance differences in Lists.
// {Args: 100 500} Small to keep build testing short
import java.util.*;
import net.mindview.util.*;

public class ListPerformance {
    static Random rand = new Random();
    static int reps = 1000;
    static List<Test<List<Integer>>> tests =
        new ArrayList<Test<List<Integer>>>();
    static List<Test<LinkedList<Integer>>> qTests =
        new ArrayList<Test<LinkedList<Integer>>>();
    static {
        tests.add(new Test<List<Integer>>("add") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops;
                int listSize = tp.size;
                for(int i = 0; i < loops; i++) {
                    list.clear();
                    for(int j = 0; j < listSize; j++)
                        list.add(j);
                }
                return loops * listSize;
            }
        });
        tests.add(new Test<List<Integer>>("get") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops * reps;
                int listSize = list.size();
                for(int i = 0; i < loops; i++)
                    list.get(rand.nextInt(listSize));
                return loops;
            }
        });
        tests.add(new Test<List<Integer>>("set") {
            int test(List<Integer> list, TestParam tp) {
                int loops = tp.loops * reps;
                int listSize = list.size();
                for(int i = 0; i < loops; i++)
```

```
            list.set(rand.nextInt(listSize), 47);
        return loops;
    }
});
tests.add(new Test<List<Integer>>("iteradd") {
    int test(List<Integer> list, TestParam tp) {
        final int LOOPS = 1000000;
        int half = list.size() / 2;
        ListIterator<Integer> it = list.listIterator(half);
        for(int i = 0; i < LOOPS; i++)
            it.add(47);
        return LOOPS;
    }
});
tests.add(new Test<List<Integer>>("insert") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        for(int i = 0; i < loops; i++)
            list.add(5, 47); // Minimize random-access cost
        return loops;
    }
});
tests.add(new Test<List<Integer>>("remove") {
    int test(List<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            list.addAll(new CountingIntegerList(size));
            while(list.size() > 5)
                list.remove(5); // Minimize random-access cost
        }
        return loops * size;
    }
});
// Tests for queue behavior:
qTests.add(new Test<LinkedList<Integer>>("addFirst") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            for(int j = 0; j < size; j++)
                list.addFirst(47);
        }
        return loops * size;
    }
});
qTests.add(new Test<LinkedList<Integer>>("addLast") {
    int test(LinkedList<Integer> list, TestParam tp) {
        int loops = tp.loops;
        int size = tp.size;
        for(int i = 0; i < loops; i++) {
            list.clear();
            for(int j = 0; j < size; j++)
                list.addLast(47);
        }
```

```
                return loops * size;
            }
        });
        qTests.add(new Test<LinkedList<Integer>>("rmFirst") {
            int test(LinkedList<Integer> list, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    list.clear();
                    list.addAll(new CountingIntegerList(size));
                    while(list.size() > 0)
                        list.removeFirst();
                }
                return loops * size;
            }
        });
        qTests.add(new Test<LinkedList<Integer>>("rmLast") {
            int test(LinkedList<Integer> list, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    list.clear();
                    list.addAll(new CountingIntegerList(size));
                    while(list.size() > 0)
                        list.removeLast();
                }
                return loops * size;
            }
        });
    }
    static class ListTester extends Tester<List<Integer>> {
        public ListTester(List<Integer> container,
                          List<Test<List<Integer>>> tests) {
            super(container, tests);
        }
        // Fill to the appropriate size before each test:
        @Override
        protected List<Integer> initialize(int size) {
            container.clear();
            container.addAll(new CountingIntegerList(size));
            return container;
        }
        // Convenience method:
        public static void run(List<Integer> list,
                               List<Test<List<Integer>>> tests) {
            new ListTester(list, tests).timedTest();
        }
    }
    public static void main(String[] args) {
        if(args.length > 0)
            Tester.defaultParams = TestParam.array(args);
        // Can only do these two tests on an array:
        Tester<List<Integer>> arrayTest =
            new Tester<List<Integer>>(null, tests.subList(1, 3)) {
                // This will be called before each test. It
                // produces a non-resizeable array-backed list:
                @Override
```

```
                protected List<Integer> initialize(int size) {
                    Integer[] ia = Generated.array(
                        Integer.class, new CountingGenerator.Integer(), size);
                    return Arrays.asList(ia);
                }
            };
        arrayTest.setHeadline("Array as List");
        arrayTest.timedTest();
        Tester.defaultParams =
            TestParam.array(10, 5000, 100, 5000, 1000, 1000, 10000, 200);
        if(args.length > 0)
            Tester.defaultParams = TestParam.array(args);
        ListTester.run(new ArrayList<Integer>(), tests);
        ListTester.run(new LinkedList<Integer>(), tests);
        ListTester.run(new Vector<Integer>(), tests);
        Tester.fieldWidth = 12;
        Tester<LinkedList<Integer>> qTest =
            new Tester<LinkedList<Integer>>(new LinkedList<Integer>(), qTests);
        qTest.setHeadline("Queue tests");
        qTest.timedTest();
    }
} /* Output: (Sample)
          --- Array as List ---
           size     get     set
             10     130     183
            100     130     164
           1000     129     165
          10000     129     165
          --------------------- ArrayList ---------------------
           size     add     get     set iteradd  insert  remove
             10     121     139     191     435    3952     446
            100      72     141     191     247    3934     296
           1000      98     141     194     839    2202     923
          10000     122     144     190    6880   14042    7333
          --------------------- LinkedList ---------------------
           size     add     get     set iteradd  insert  remove
             10     182     164     198     658     366     262
            100     106     202     230     457     108     201
           1000     133    1289    1353     430     136     239
          10000     172   13648   13187     435     255     239
          ----------------------- Vector -----------------------
           size     add     get     set iteradd  insert  remove
             10     129     145     187     290    3635     253
            100      72     144     190     263    3691     292
           1000      99     145     193     846    2162     927
          10000     108     145     186    6871   14730    7135
          -------------------- Queue tests --------------------
           size    addFirst     addLast     rmFirst      rmLast
             10         199         163         251         253
            100          98          92         180         179
           1000          99          93         216         212
          10000         111         109         262         384
  */// :~
```

[127] Jeder Test setzt sorgfältige Überlegung voraus, um sicherzustellen, daß Sie sinnvolle Ergebnisse bekommen. Beispielsweise leert der „add"-Test den *List*-Container, bevor er ihn mit der festgelegten Anzahl von Elementen füllt. Das Aufrufen der `clear()`-Methode ist somit ein Teil des Tests

und wirkt sich, inbesondere bei kleinen Tests, auf die benötigte Zeit aus. Obwohl die Testergebnisse hier sinnvoll wirken, könnten Sie in Betracht ziehen, die Testbibliothek so umzuschreiben, daß *außerhalb* der Zeitmessung eine Vorbereitungsmethode aufgerufen wird (die in diesem Fall die `clear()`-Methode aufruft).

[128] Beachten Sie bei jedem Test, daß Sie die Anzahl der Methodenaufrufe exakt berechnen und aus der `test()`-Methode zurückgeben, damit die Zeitmessung korrekt ist.

[129] Die „get"- und „set"-Tests verwenden einen Zufallszahlengenerator, um die Zugriffspositionen im *`List`*-Container zu wählen. Die Ausgabe zeigt, daß ein arraygestützter *`List`*-Container oder ein `ArrayList`-Container diese Zugriffe unabhängig von der Kapazität des Containers schnell und sehr konsistent ausführt, während die Zugriffszeiten beim Containertyp `LinkedList` bei zunehmender Kapazität deutlich wachsen. Offensichtlich ist die verkette Liste keine gute Wahl, wenn Sie viele Zugriffe auf beliebige Positionen benötigen.

[130] Der „iteradd"-Test verwendet einen in der Mitte des *`List`*-Containers positionierten Iterator, um neue Elemente einzusetzen. Diese Operation wird beim Containertyp `ArrayList` mit wachsender Kapazität teurer, bleibt dagegen aber beim Typ `LinkedList` unabhängig von der Kapazität konstant und vergleichsweise günstig. Dieses Verhalten ist dadurch zu erklären, daß der Containertyp `ArrayList` während des Einsetzens Speicherplatz anfordern und alle bereits gespeicherten Referenzen kopieren muß. Je mehr Elemente ein solcher Container enthält, umso aufwändiger wird dieser Vorgang. Ein Container vom Typ `LinkedList` muß dagegen lediglich einen Verweis auf ein neues Element anlegen, wobei der restliche Container nicht verändert zu werden braucht. Daher ist der zu erwartende Aufwand ungeachtet der Kapazität stets ungefähr gleich.

[131] Die „insert"- und „remove"-Tests beziehen sich nicht auf Anfang oder Ende des *`List`*-Containers, sondern auf die Speicherposition 5, um Elemente einzusetzen beziehungsweise zu entfernen. Ein Container vom Typ `LinkedList` implementiert ein spezielles Verhalten für die Endpunkte der Liste, um die Zugriffsgeschwindigkeit bei Verwendung als Warteschlange zu erhöhen. Beim Einsetzen oder Entfernen von Elementen in der Mitte der Liste, nehmen Sie den Aufwand durch den Zugriff auf eine beliebige Position in Kauf, der von der spezifischen Implementierung des Interfaces *`List`* abhängt, wie wir bereits gesehen haben. Durch das Einsetzen an beziehungsweise Entfernen von Elementen von Position 5, sollte der Aufwand für den Zugriff auf eine beliebige Position vernachlässigbar sein, so daß wir nur den Aufwand der Einsetzungs- beziehungsweise Löschoperation sehen und das optimierte Verhalten am Anfang und Ende der Liste umgehen. Die Ausgabe dokumentiert, daß der Aufwand beim Einsetzen beziehungsweise Entfernen von Elementen bei einem `LinkedList`-Container gering ist und nicht von der Kapazität der Liste abhängt. Beim Containertyp `ArrayList` ist das Einsetzen und Löschen dagegen *sehr teuer* und die Unkosten wachsen mit der Kapazität der Liste.

[132] Die Warteschlangentests zeigen, wie schnell Sie Elemente an den Enden eines Containers vom Typ `LinkedList` einsetzen oder von dort entfernen können. Dieses Verhalten wurde im Hinblick auf die Implementierung des Interfaces *`Queue`* optimiert.

[133–134] In der Regel rufen Sie nur die `Tester`-Methode `run()` auf und übergeben den Container und die Liste der durchzuführenden Tests. In diesem Fall muß die `initialize()`-Methode überschrieben werden, so daß der *`List`*-Container vor jedem Test geleert und wieder gefüllt wird, da andernfalls die Kontrolle über die Kapazität des *`List`*-Containers während der verschiedenen Tests verloren gehen würde. Die Klasse `ListTester` ist von `Tester` abgeleitet, bewerkstelligt ihre Initialisierung mit Hilfe der Klasse `CountingIntegerList` (Seite 627) und überschreibt auch die Methode `run()`. Wir wollen auch den Arrayzugriff mit dem Containerzugriff vergleichen (vor allem beim Containertyp `ArrayList`). Für den ersten Test wird in der `main()`-Methode mit Hilfe einer anonymen inneren Klasse ein spezielles `Test`-Objekt erzeugt. Die `initialize()`-Methode wird überschrieben und erzeugt

bei jedem Aufruf ein neues Objekt (das `container`-Argument des `Test`-Konstruktors wird mit `null` bewertet, das gespeicherte Containerobjekt also nicht beachtet). Das neue Objekt wird mit Hilfe der `Generated`-Methode `array()` (definiert in Unterbschnitt 17.6.3) und der statischen Methoden `Arrays.asList()` erzeugt. Da Sie einen arraygestützten *`List`*-Container nicht um Elemente erweitern oder kürzen können, werden für diesen Fall nur zwei Tests ausgeführt. Die *`List`*-Methode `subList()` wird aufgerufen, um die gewünschten Tests aus der Testliste zu wählen.

[135] Beim Zugriff auf eine beliebige Position sind die Methoden `get()` und `set()` bei einem arraygestützten *`List`*-Container geringfügig schneller, als bei einem `ArrayList`-Container. Dieselben Operationen sind bei einem `LinkedList`-Container dramatisch teurer, da dieser Containertyp nicht für den Zugriff auf beliebige Positionen optimiert ist.

[136] Die Klasse `Vector` sollte vermieden werden, da sie nur noch in der Bibliothek vorhanden ist, um ältere Programme zu unterstützen. (`Vector` funktioniert in diesem Programm nur, weil die Klasse aus Gründen der Vorwärtskompatibilität das Interface *`List`* implementiert.)

[137] Es ist wahrscheinlich am besten, wenn Sie stets den Containertyp `ArrayList` wählen und auf `LinkedList` umsteigen, wenn Sie die zusätzliche Funktionalität brauchen oder feststellen, daß sich durch viele Einsetzungs- und Entfernungsoperationen in der Mitte des Containers Leistungseinbußen ergeben. Wenn Sie eine feste Anzahl von Elementen verarbeiten, wählen Sie entweder einen arraygestützten *`List`*-Container (wie etwa den Rückgabewert der statischen `Arrays`-Methode `asList()`) oder falls erforderlich ein „richtiges“ Array.

[138] Der Containertyp `CopyOnWriteArrayList` ist eine spezielle Implementierung des Interfaces *`List`* für die Threadprogrammierung und wird in den Unterabschnitten 22.7.7 und 22.9.2 sowie in Abschnitt 22.10 (Seite 1001) diskutiert.

Übungsaufgabe 29: (2) Ändern Sie das Beispiel *ListPerformance.java* (Seiten 670–673), so daß die *`List`*-Container `String`-Objekte anstelle von `Integer`-Objekten enthalten. Verwenden Sie eine der Generatorklassen aus Kapitel 17, um Testdaten zu erzeugen. ■

Übungsaufgabe 30: (3) Vergleichen Sie die Performanz der statischen `Collections`-Methode `sort()` zwischen den Containertypen `ArrayList` und `LinkedList`. ■

Übungsaufgabe 31: (5) Schreiben Sie eine Containerklasse, die ein `String`-Array kapselt und nur das Hinzufügen und Abfragen von `String`-Objekten erlaubt, so daß während der Anwendung keine Typumwandlungen stattfinden. Wenn das interne Array nicht groß genug ist, um das nächste hinzuzufügende Element aufzunehmen, soll der Container das alte Array durch ein neues Array passender Länge ersetzen. Vergleichen Sie in der `main()`-Methode die Performanz Ihres Containers mit einem Container vom Typ `ArrayList<String>`. ■

Übungsaufgabe 32: (2) Wiederholen Sie Übungsaufgabe 31 mit einem `int`-Array und vergleichen Sie die Performanz mit einem Container vom Typ `ArrayList<Integer>`. Nehmen Sie das Inkrementieren jedes Elementes in den beiden Containern in Ihren Performanzvergleich auf. ■

Übungsaufgabe 33: (5) Schreiben Sie eine Klasse `FastTraversalLinkedList`, die intern einen `LinkedList`-Container für schnelles Einsetzen und Entfernen und einen `ArrayList`-Container für schnelles Traversieren und `get()`-Operationen verwendet. Testen Sie Ihre Klasse mit dem Programm *ListPerformance.java* (Seiten 670–673). ■

18.10.3 Gefahren bei „Mikrobenchmarks“

[139] Beim Entwickeln sogenannter *Mikrobenchmarks* müssen Sie sorgfältig darauf achten, nicht zuviel vorauszusetzen und Ihren Test soweit als möglich einzuengen, um den Zeitverbrauch nur der

Dinge zu messen, die interessant sind. Sie müssen außerdem gewährleisten, daß Ihre Tests lange genug laufen, um interessante Daten erzeugen zu können und dabei berücksichtigen, daß Teile der HotSpot-Technologie von Java erst aktiv werden, wenn das Programm eine gewisse Zeit lang läuft (dieser Aspekt ist auch bei Programmen mit kurzer Laufzeit wichtig).

[140] Die Ergebnisse hängen von Ihrem Rechner und Ihrer Laufzeitumgebung ab. Sie sollten die Tests daher selbst laufen lassen und die Ergebnisse im Buch verifizieren. Achten Sie dabei weniger auf die absoluten Zahlen, sondern auf die Performanzunterschiede zwischen den verschiedenen Containertypen.

[141] Darüber hinaus ist ein Profiler eher als Sie im Stande, eine Performanzanalyse durchzuführen. Java wird mit einem Profiler ausgeliefert (siehe Anhang in *http://www.mindview.net/Books/BetterJava*) und es gibt sowohl freie als auch kommerzielle Profiler von Drittanbietern.

[142] Die statische `Math`-Methode `random()` liefert ein Beispiel [wofür?]. Schließt der Wertebereich zwischen 0 und 1 die Randpunkte ein? In mathematischer Notation: Ist der Wertebereich $(0, 1)$, $[0, 1)$, $(0, 1]$ oder $[0, 1]$? (Die eckige Klammer bedeutet „inklusive“, die runde Klammer dagegen „nicht inklusive“). Ein Testprogramm *kann* die Antwort liefern:

```
//: containers/RandomBounds.java
// Does Math.random() produce 0.0 and 1.0?
// {RunByHand}
import static net.mindview.util.Print.*;

public class RandomBounds {
    static void usage() {
        print("Usage:");
        print("\tRandomBounds lower");
        print("\tRandomBounds upper");
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length != 1) usage();
        if(args[0].equals("lower")) {
            while(Math.random() != 0.0)
                ; // Keep trying
            print("Produced 0.0!");
        }
        else if(args[0].equals("upper")) {
            while(Math.random() != 1.0)
                ; // Keep trying
            print("Produced 1.0!");
        }
        else
            usage();
    }
} ///:~
```

Das Programm wird entweder so aufgerufen:

```
java RandomBounds lower
```

oder so:

```
java RandomBounds upper
```

Sie müssen das Programm in beiden Fällen von Hand abbrechen. Die `random()`-Methode der Klasse `Math` scheint niemals 0.0 oder 1.0 zu liefern. Angesichts von etwa 2^{62} `double`-Werten zwischen 0 und 1 ist die Wahrscheinlichkeit, einen bestimmten Wert experimentell zu erhalten, so gering, daß sie die

Lebensdauer des Rechners oder gar des Programmierers überschreiten kann. Das Ergebnis lautet, daß der Wertebereich der `random()`-Methode 0.0 enthält, in mathematischer Schreibweise lautet der Wertebereich $[0, 1)$. Sie müssen Ihre Experimente also sorgfältig untersuchen und die jeweiligen Einschränkungen verstehen.

18.10.4 Implementierungen des Interfaces Set

[143] Je nachdem welches Verhalten Sie brauchen, haben Sie die Wahl zwischen drei Implementierungen des Interfaces *`Set`*: `TreeSet`, `HashSet` und `LinkedHashSet`. Das folgende Testprogramm liefert einen Anhaltspunkt für den Performanzunterschied zwischen diesen drei Klassen:

```
//: containers/SetPerformance.java
// Demonstrates performance differences in Sets.
// {Args: 100 5000} Small to keep build testing short
import java.util.*;

public class SetPerformance {
    static List<Test<Set<Integer>>> tests =
        new ArrayList<Test<Set<Integer>>>();
    static {
        tests.add(new Test<Set<Integer>>("add") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    set.clear();
                    for(int j = 0; j < size; j++)
                        set.add(j);
                }
                return loops * size;
            }
        });
        tests.add(new Test<Set<Integer>>("contains") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops;
                int span = tp.size * 2;
                for(int i = 0; i < loops; i++)
                    for(int j = 0; j < span; j++)
                        set.contains(j);
                return loops * span;
            }
        });
        tests.add(new Test<Set<Integer>>("iterate") {
            int test(Set<Integer> set, TestParam tp) {
                int loops = tp.loops * 10;
                for(int i = 0; i < loops; i++) {
                    Iterator<Integer> it = set.iterator();
                    while(it.hasNext())
                        it.next();
                }
                return loops * set.size();
            }
        });
    }
    public static void main(String[] args) {
        if(args.length > 0)
```

```
            Tester.defaultParams = TestParam.array(args);
        Tester.fieldWidth = 10;
        Tester.run(new TreeSet<Integer>(), tests);
        Tester.run(new HashSet<Integer>(), tests);
        Tester.run(new LinkedHashSet<Integer>(), tests);
    }
} /* Output: (Sample)
          ------------- TreeSet -------------
           size       add  contains   iterate
             10       746       173        89
            100       501       264        68
           1000       714       410        69
          10000      1975       552        69
          ------------- HashSet -------------
           size       add  contains   iterate
             10       308        91        94
            100       178        75        73
           1000       216       110        72
          10000       711       215       100
          ---------- LinkedHashSet ----------
           size       add  contains   iterate
             10       350        65        83
            100       270        74        55
           1000       303       111        54
          10000      1615       256        58
    *///:~
```

[144] Der Containertyp `HashSet` ist `TreeSet` in der Regel überlegen, insbesondere aber beim Einsetzen und Abfragen von Elementen, also den beiden wichtigsten Operationen. Die Klasse `TreeSet` existiert nur, weil sie ihre Elemente in sortierter Reihenfolge ablegt und wird nur verwendet, wenn eine sortierte Menge erforderlich ist. Das Iterieren über die Elemente des Containers ist bei `TreeSet` gewöhnlich schneller als bei `HashSet`, sowohl aufgrund der zur sortierten Ablage erforderlichen inneren Struktur als auch da das Iterieren über die Elemente eine eher wahrscheinlichere Operation ist.

[145] Beachten Sie, daß das Einsetzen von Elementen beim Containertyp `LinkedHashSet` teurer ist als bei `HashSet`. Der Grund liegt beim zusätzlichen Aufwand, eine verkettete Liste und einen hashbasierten Container zu pflegen.

Übungsaufgabe 34: (1) Ändern Sie das Beispiel *SetPerformance.java*, so daß die *`Set`*-Container `String`-Objekte statt `Integer`-Objekten enthalten. Verwenden Sie eine der Generatorklassen aus Kapitel 17, um Testdaten zu erzeugen. ■

18.10.5 Implementierungen des Interfaces Map

[146] Das folgende Testprogramm liefert einen Anhaltspunkt für den Performanzunterschied zwischen den Implementierungen des Interfaces *`Map`*:

```
//: containers/MapPerformance.java
// Demonstrates performance differences in Maps.
// {Args: 100 5000} Small to keep build testing short
import java.util.*;

public class MapPerformance {
    static List<Test<Map<Integer,Integer>>> tests =
        new ArrayList<Test<Map<Integer,Integer>>>();
```

```
    static {
        tests.add(new Test<Map<Integer,Integer>>(''put'') {
            int test(Map<Integer,Integer> map, TestParam tp) {
                int loops = tp.loops;
                int size = tp.size;
                for(int i = 0; i < loops; i++) {
                    map.clear();
                    for(int j = 0; j < size; j++)
                        map.put(j, j);
                }
                return loops * size;
            }
        });
        tests.add(new Test<Map<Integer,Integer>>(''get'') {
            int test(Map<Integer,Integer> map, TestParam tp) {
                int loops = tp.loops;
                int span = tp.size * 2;
                for(int i = 0; i < loops; i++)
                    for(int j = 0; j < span; j++)
                        map.get(j);
                return loops * span;
            }
        });
        tests.add(new Test<Map<Integer,Integer>>(''iterate'') {
            int test(Map<Integer,Integer> map, TestParam tp) {
                int loops = tp.loops * 10;
                for(int i = 0; i < loops; i ++) {
                    Iterator it = map.entrySet().iterator();
                    while(it.hasNext())
                        it.next();
                }
                return loops * map.size();
            }
        });
    }
    public static void main(String[] args) {
        if(args.length > 0)
            Tester.defaultParams = TestParam.array(args);
        Tester.run(new TreeMap<Integer,Integer>(), tests);
        Tester.run(new HashMap<Integer,Integer>(), tests);
        Tester.run(new LinkedHashMap<Integer,Integer>(),tests);
        Tester.run(new IdentityHashMap<Integer,Integer>(), tests);
        Tester.run(new WeakHashMap<Integer,Integer>(), tests);
        Tester.run(new Hashtable<Integer,Integer>(), tests);
    }
} /* Output: (Sample)
          ---------- TreeMap ----------
         size     put     get iterate
           10     748     168     100
          100     506     264      76
         1000     771     450      78
        10000    2962     561      83
          ---------- HashMap ----------
         size     put     get iterate
           10     281      76      93
          100     179      70      73
         1000     267     102      72
```

```
        10000    1305     265      97
        ------- LinkedHashMap -------
         size     put     get iterate
           10     354     100      72
          100     273      89      50
         1000     385     222      56
        10000    2787     341      56
        ------ IdentityHashMap ------
         size     put     get iterate
           10     290     144     101
          100     204     287     132
         1000     508     336      77
        10000     767     266      56
        -------- WeakHashMap --------
         size     put     get iterate
           10     484     146     151
          100     292     126     117
         1000     411     136     152
        10000    2165     138     555
        --------- Hashtable ---------
         size     put     get iterate
           10     264     113     113
          100     181     105      76
         1000     260     201      80
        10000    1245     134      77
*///:~
```

Das Einsetzen neuer Schlüssel/Wert-Paare wird, mit Ausnahme von `IdentityHashMap`, bei allen Implementierungen des Interfaces *`Map`* mit zunehmender Kapazität deutlich langsamer. Im allgemeinen ist das Nachschlagen allerdings erheblich günstiger als das Einsetzen. Das ist vorteilhaft, da Sie typischerweise Einträge viel häufiger nachschlagen als einsetzen.

[147] Die Containertypen `Hashtable` und `HashMap` sind ungefähr gleich performant. Das ist nicht überraschend, da `HashMap` die veraltete Klasse `Hashtable` ersetzen soll und beide Klassen auf demselben Speicherungs- und Nachschlagemechanismus (siehe später in diesem Kapitel) aufbauen.

[148] Der Containertyp `TreeMap` ist generall langsamer als `HashMap` und dient, wie `TreeSet` dazu, die gespeicherten Elemente zu sortieren. Ein Baum ist stets geordnet und bedarf keiner speziellen Sortierung. Nachdem Sie einen `TreeMap`-Container gefüllt haben, liefert die Methode `keySet()` eine Sicht auf die Schlüssel in Form eines *`Set`*-Containers, dessen `toArray()`-Methode diese Schlüssel wiederum als Array zurückgibt. Mit Hilfe der statischen `Arrays`-Methode `binarySearch()` sind Elemente eines sortierten Arrays schnell auffindbar. Nachdem die Klasse `HashMap` auf schnelles Finden von Schlüsseln ausgerichtet ist, kommt diese Vorgehensweise selbstverständlich nur dann in Frage, wenn die Klasse `HashMap` aufgrund ihres Verhaltens ausscheidet. Sie können außerdem über die `TreeMap`-Methode `putAll()` in einem Schritt einen `HashMap`-Container erzeugen. Letztendlich sollten Sie einen `HashMap`-Container wählen und sich nur dann für einen `TreeMap`-Container entscheiden, wenn Sie einen stets sortierten *`Map`*-Container brauchen.

[149] Der Containertyp `LinkedHashMap` ist beim Einsetzen tendentiell langsamer als `HashMap`, da er sowohl eine verkettete Liste (zum Erhalten der Einsetzungsreihenfolge) als auch eine hashbasierte Datenstruktur pflegt. Aufgrund der verketteten Liste ist `LinkedHashMap` beim Iterieren schneller als `HashMap`.

[150] Die Performanz des Containertyps `IdentityHashMap` fällt aus dem Rahmen, da Vergleiche mit Hilfe des `==`-Operators anstelle der `equals()`-Methode durchgeführt werden. Die Klasse `WeakHashMap` wird in Unterabschnitt 18.12.1 beschrieben.

Übungsaufgabe 35: (1) Ändern Sie das Beispiel *MapPerformance.java*, so daß Sie die Klasse `SlowMap` (Seite 655) testen können. ■

Übungsaufgabe 36: (5) Ändern Sie die Klasse `SlowMap` (Seite 655), so daß sie keine zwei, sondern nur noch einen `ArrayList`-Container von `MapEntry`-Objekten enthält. Verifizieren Sie, daß die überarbeitete Version korrekt funktioniert. Verwenden Sie das Beispiel *MapPerformance.java*, um die Geschwindigkeit Ihrer geänderten `SlowMap`-Klasse zu testen. Ändern Sie nun die `put()`-Methode, so daß jedesmal nach dem Einsetzen eines Schlüssel/Wert-Paares die `sort()`-Methode sowie die `get()`-Methode, so daß der angeforderte Schlüssel mit Hilfe der statischen `Collections`-Methode `binarySearch()` gesucht wird. Vergleichen Sie die Performanz zwischen der alten und der neuen Version. ■

Übungsaufgabe 37: (2) Ändern Sie die Klasse `SimpleHashMap` (Seite 658), so daß sie Container und Referenzvariablen vom Typ `ArrayList` anstelle von `LinkedList` verwendet. ■

18.10.5.1 Performanzfaktoren beim Containertyp HashMap

[151] Sie können die Performanz eines Containers vom Typ `HashMap` durch manuelles Feinabstimmen verbessern. Die folgenden vier Begriffe sind notwendig, damit Sie die Faktoren verstehen, die sich auf die Performanz eines `HashMap`-Containers auswirken:

- Die *Kapazität* (*capacity*) gibt die Anzahl der Buckets in der Hashtabelle an.
- Die *Anfangskapazität* (*initial capacity*) gibt die Anzahl der Buckets beim Erzeugen der Hashtabelle an. Die Containerklassen `HashMap` und `HashSet` verfügen über Konstruktoren, mit denen Sie die Anfangskapazität einstellen können.
- Die *Größe* (*size*) gibt an, wieviele Einträge sich im Augenblick in der Hashtabelle befinden.
- Ein *Ladefaktor* (*load factor*) von 0 bedeutet eine leere Tabelle, ein Ladefaktor von 0.5 eine halbvolle Tabelle und so weiter. Eine wenig geladene Tabelle hat nur wenige Kollisionen und eignet sich optimal für Einsetzungs- oder Nachschlageoperationen (verlangsamt aber das Traversieren per Iterator). Die Containerklassen `HashMap` und `HashSet` verfügen über Konstruktoren, mit denen Sie den Ladefaktor einstellen können. Wird dieser Ladefaktor erreicht, so erhöht der Container automatisch seine Kapazität (Anzahl der Buckets), indem er sie ungefähr verdoppelt und die bereits gespeicherten Elemente neu auf die Buckets verteilt (dieser Vorgang wird als **Rehashing** bezeichnet).

[152] Für die Klasse `HashMap` ist ein Ladefaktor von 0.75 voreingestellt, das heißt es findet kein Rehashing statt, bis die Hashtabelle zu drei Vierteln gefüllt ist. Dies ist ein guter Kompromiß zwischen Zeitaufwand und Speicherbedarf. Ein höherer Ladefaktor verringert zwar den von Hashtabelle beanspruchten Speicherplatz, erhöht aber den Aufwand beim Nachschlagen. Dieser Aspekt ist wesentlich, da Sie erheblich mehr Nachschlage- als Einsetzungsoperationen ausführen (betrifft sowohl `get()` als auch `put()`).

[153] Wenn Sie bereits wissen, daß Sie viele Schlüssel/Wert-Paare in einem `HashMap`-Container speichern werden, verhindern Sie den Aufwand durch automatisches Rehashing, indem Sie den Container mit einer angemessen großen Anfangskapazität erzeugen.[11]

[11] Joshua Bloch schreibt in einer privaten Mitteilung: „... I believe that we erred by allowing implementation details (such as hash table size and load factor) into our APIs. The client should perhaps tell us the maximum expected size of a collection, and we should take it from there. Clients can easily do more harm than good by choosing values for these parameters. As an extreme example, consider `Vectors` `capacityIncrement`. No one should ever set this, and we shouldn't have provided it. If you set it to any nonzero value, the asymptotic cost of a sequence of appends goes from linear to quadratic. In other words, it destroys your performance. Over time, we're beginning to wise up about

Übungsaufgabe 38: (3) Lesen Sie die API-Dokumentation der Containerklasse `HashMap` nach. Erzeugen Sie einen `HashMap`-Container, füllen Sie ihn mit Einträgen und bestimmen Sie seinen Ladefaktor. Testen Sie die Zugriffsgeschwindigkeit beim Nachschlagen für diesen Container und versuchen Sie anschließend, die Geschwindigkeit zu erhöhen, indem Sie einen neuen `HashMap`-Container mit größerer Anfangskapazität erzeugen und den Inhalt des alten Containers in den neuen kopieren. Testen Sie die Zugriffsgeschwindigkeit des neuen Containers. ■

Übungsaufgabe 39: (6) Legen Sie in der Klasse `SimpleHashMap` (Seite 658) eine private `rehash()`-Methode an, die aufgerufen wird, wenn der Ladefaktor über 0.75 hinaus wächst. Beim Rehashing verdoppeln Sie zunächst die Anzahl der Buckets (b) und suchen anschließend nach der ersten Primzahl, die größer als die doppelte Anzahl der Buckets ist ($p > 2b$) und wählen diese Primzahl als neue Bucketanzahl ($b' := p$). ■

18.11 Die statischen Methoden der Hilfsklasse Collections

[154] Die Klasse `java.util.Collections` bietet eine Reihe von statischen Hilfsmethoden für Container. Einige davon haben Sie bereits kennengelernt, darunter `addAll()`, `reverseOrder()` und `binarySearch()`. Die folgende Liste nennt und beschreibt die übrigen Hilfsmethoden mit Ausnahme der Methoden, die synchronisierte Container (Unterabschnitt 18.11.3) oder unveränderliche Container (Unterabschnitt 18.11.2) zurückgeben. Die Liste verwendet in relevanten Fällen generische Typen:

- `checkedCollection(Collection<T>, Class<T> type)`,
  `checkedList(List<T>, Class<T> type)`,
  `checkedMap(Map<K,V>, Class<K> keyType, Class<V> valueType)`,
  `checkedSet(Set<T>, Class<T> type)`,
  `checkedSortedMap(SortedMap<K,V>, Class<K> keyType, Class<V> valueType)` und
  `checkedSortedSet(SortedSet<T>, Class<T> type)`:

  Diese Methoden liefern eine *zur Laufzeit* typsichere Sicht auf einen Container vom Typ *`Collection`* beziehungsweise einen Untertyp davon. Sie verwenden diese Methode, wenn es nicht möglich ist, den entsprechenden statisch typgeprüften Container zu wählen. (Siehe auch die Besprechung dieser Methoden in Abschnitt 16.13.)

- `max(Collection)` und `min(Collection)`: Liefern das größte beziehungsweise kleinste Element des als Argument übergebenen *`Collection`*-Containers bezüglich der natürlichen Ordnung der Elemente.

- `max(Collection, Comparator)` und `min(Collection, Comparator)`: Liefern das größte beziehungsweise kleinste Element des als Argument übergebenen *`Collection`*-Containers bezüglich des Komparators.

this sort of thing. If you look at `IdentityHashMap`, you'll see that it has no low-level tuning parameters."

Übersetzt etwa: „... ich glaube, es war ein Fehler, daß wir erlaubt haben, Implementierungsdetails wie die Größe der Hashtabelle oder den Ladefaktor über die Schnittstelle der Containerobjekte einzustellen. Vielleicht sollten die Programmierer uns die maximal zu erwartende Größe eines Containers mitteilen und wir würden diese Angabe übernehmen. Programmierer können sich leicht mehr schaden als nützen, wenn sie diese Eigenschaften selbst bewerten. Die Eigenschaft `capacityIncrement` der Klasse `Vector` ist ein extremes Beispiel. Niemand sollte jemals diesen Parameter selbst bewerten und wir hätten ihn nicht entblößen dürfen. Wenn Sie die Eigenschaft auf einen von Null verschiedenen Wert setzen, schlägt das Wachstumsverhalten des asymptotischen Aufwandes bei einer Reihe von Einsetzungsoperationen am Ende des Vektors von linear nach quadratisch um, vernichtet also die Performanz. Wir sind im Laufe der Zeit zu dieser Einsicht gekommen. Die Klasse `IdentityHashMap` hat beispielsweise keine systemnahen Feinabstimmungsparameter mehr."

- `indexOfSubList(List source, List target)`: Gibt den ersten Index des ersten Vorkommens von `target` in dem von `source` referenzierten *`List`*-Container oder -1, falls kein Vorkommen existiert.
- `lastIndexOfSubList(List source, List target)`: Gibt den ersten Index des letzten Vorkommens von `target` in dem von `source` referenzierten *`List`*-Container oder -1, falls kein Vorkommen existiert.
- `replace(List<T>, T oldVal, T newVal)`: Ersetzt alle Vorkommen von `oldVal` im *`List`*-Container durch `newVal`.
- `reverse(List)`: Kehrt die Reihenfolge der Elemente im übergebenen *`List`*-Container um.
- `reverseOrder()` und `reverseOrder(Comparator<T>)`: Gibt einen Komparator zurück, der die natürliche Ordnung der Elemente eines Containers vom Typ *`Collection`* umgekehrt, die das Interface *`Comparable<T>`* implementieren. Die zweite Version kehrt die durch den Komparator definierte Reihenfolge um.
- `rotate(List, int distance)`: Bewegt alle Elemente des *`List`*-Containers um `distance` Positionen vorwärts. Die am Ende „herausgefallenen“ Elemente, werden am Anfang wieder eingesetzt.
- `shuffle(List)` und `shuffle(List, Random)`: Liefert eine zufällige Permutation der Elemente des *`List`*-Containers. Die erste Version verwendet ihre eigene Quelle von Zufallszahlen, die zweite Version erwartet einen Zufallsgenerator.
- `sort(List<T>)` und `sort(List<T>, Comparator<? super T> c)`: Sortiert die Elemente des übergebenen *`List`*-Containers bezüglich der natürlichen Ordnung. Die zweite Version erwartet einen Komparator.
- `copy(List<? super T> dest, List<? extends T> src)`: Kopiert Elemente aus dem *`List`*-Container `src` in den *`List`*-Container `dest`.
- `swap(List, int i, int j)`: Vertauscht die Elemente an den Positionen `i` und `j` des *`List`*-Containers. Die `swap()`-Methode ist wahrscheinlich schneller als eine von Hand geschriebene Methode.
- `fill(List<? super T>, T x)`: Ersetzt alle Elemente des Containers durch `x`.
- `nCopies(int n, T x)`: Gibt einen unveränderlichen *`List<T>`*-Container zurück. Der Container hat die Größe `n` und alle Elemente verweisen auf `x`.
- `disjoint(Collection, Collection)`: Gibt `true` zurück, wenn beide *`Collection`*-Container elementfremd sind.
- `frequency(Collection, Object x)`: Gibt die Anzahl der Element im Container zurück, die mit `x` übereinstimmen.
- `emptyList()`, `emptyMap()` und `emptySet()`: Gibt einen unveränderlichen leeren Container des Typs *`List`*, *`Map`* oder *`Set`* zurück. Die Container sind generisch, erhalten also den gewünschten Typparameter.
- `singleton(T x)`, `singletonList(T x)` und `singletonMap(K key, V value)`: Gibt einen unveränderlichen Container des Typs *`List<T>`*, *`Map<K,V>`* oder *`Set<T>`* zurück, der ein einzelnes Element enthält, nämlich das Argument der Methode.
- `list(Enumeration<T> e)`: Gibt einen `ArrayList<T>`-Container zurück, der die Element in der Reihenfolge herausgibt, in der sie von der veralteten Klasse *`Enumeration`* (Vorgänger der

Klasse *Iterator*) zurückgegeben werden. Zur Umwandlung älterer Programme.

- `enumeration(Collection<T>)`: Gibt ein veraltetes *Enumeration<T>*-Objekt für das Argument zurück.

[155] Beachten Sie, daß die Methoden `min()` und `max()` mit dem Containertyp *Collection* arbeiten, nicht mit *List*. Sie brauchen sich also nicht darum zu kümmern, ob ein *Collection*-Container sortiert ist, oder nicht. (Der Inhalt eines *List*-Containers muß erst sortiert werden, bevor Sie die binäre Suche starten können.)

[156] Das folgende Beispiel zeigt die Verwendung der meisten Hilfsmethoden in der obigen Liste:

```
//: containers/Utilities.java
// Simple demonstrations of the Collections utilities.
import java.util.*;
import static net.mindview.util.Print.*;

public class Utilities {
    static List<String> list =
        Arrays.asList("one Two three Four five six one".split(" "));
    public static void main(String[] args) {
        print(list);
        print("'list' disjoint (Four)?: " +
              Collections.disjoint(list, Collections.singletonList("Four")));
        print("max: " + Collections.max(list));
        print("min: " + Collections.min(list));
        print("max w/ comparator: "
            + Collections.max(list, String.CASE_INSENSITIVE_ORDER));
        print("min w/ comparator: "
            + Collections.min(list, String.CASE_INSENSITIVE_ORDER));
        List<String> sublist = Arrays.asList("Four five six".split(" "));
        print("indexOfSubList: " + Collections.indexOfSubList(list, sublist));
        print("lastIndexOfSubList: " + Collections.lastIndexOfSubList(list, sublist));
        Collections.replaceAll(list, "one", "Yo");
        print("replaceAll: " + list);
        Collections.reverse(list);
        print("reverse: " + list);
        Collections.rotate(list, 3);
        print("rotate: " + list);
        List<String> source = Arrays.asList("in the matrix".split(" "));
        Collections.copy(list, source);
        print("copy: " + list);
        Collections.swap(list, 0, list.size() - 1);
        print("swap: " + list);
        Collections.shuffle(list, new Random(47));
        print("shuffled: " + list);
        Collections.fill(list, "pop");
        print("fill: " + list);
        print("frequency of 'pop': " + Collections.frequency(list, "pop"));
        List<String> dups = Collections.nCopies(3, "snap");
        print("dups: " + dups);
        print("'list' disjoint 'dups'?: " +
              Collections.disjoint(list, dups));
        // Getting an old-style Enumeration:
        Enumeration<String> e = Collections.enumeration(dups);
        Vector<String> v = new Vector<String>();
        while(e.hasMoreElements())
            v.addElement(e.nextElement());
```

```
        // Converting an old-style Vector
        // to a List via an Enumeration:
        ArrayList<String> arrayList =
            Collections.list(v.elements());
        print("arrayList: " + arrayList);
    }
} /* Output:
      [one, Two, three, Four, five, six, one]
      'list' disjoint (Four)?: false
      max: three
      min: Four
      max w/ comparator: Two
      min w/ comparator: five
      indexOfSubList: 3
      lastIndexOfSubList: 3
      replaceAll: [Yo, Two, three, Four, five, six, Yo]
      reverse: [Yo, six, five, Four, three, Two, Yo]
      rotate: [three, Two, Yo, Yo, six, five, Four]
      copy: [in, the, matrix, Yo, six, five, Four]
      swap: [Four, the, matrix, Yo, six, five, in]
      shuffled: [six, matrix, the, Four, Yo, five, in]
      fill: [pop, pop, pop, pop, pop, pop, pop]
      frequency of 'pop': 7
      dups: [snap, snap, snap]
      'list' disjoint 'dups'?: true
      arrayList: [snap, snap, snap]
   *///:~
```

Die Ausgabe erklärt die Wirkung der einzelnen Hilfsmethoden. Beachten Sie das unterschiedliche Verhalten der Methoden `min()` und `max()` mit beziehungsweise ohne den Komparator `CASE_INSENSITIVE_ORDER`, bedingt durch die Groß-/Kleinschreibung.

18.11.1 Sortieren von und Suchen in List-Containern

[157] Die Hilfsmethoden zum Sortieren von und Suchen in Listen haben dieselben Namen und Signaturen wie die entsprechenden Methoden für Arrayobjekte, sind ebenfalls statisch und liegen in der Hilfsklasse `Collections`, im Gegensatz zu `Arrays`. Das folgende Beispiel verwendet die Testdaten im *`List`*-Container `list` aus dem Beispiel *Utilities.java* von Seite 684:

```
//: containers/ListSortSearch.java
// Sorting and searching Lists with Collections utilities.
import java.util.*;
import static net.mindview.util.Print.*;

public class ListSortSearch {
    public static void main(String[] args) {
        List<String> list = new ArrayList<String>(Utilities.list);
        list.addAll(Utilities.list);
        print(list);
        Collections.shuffle(list, new Random(47));
        print("Shuffled: " + list);
        // Use a ListIterator to trim off the last elements:
        ListIterator<String> it = list.listIterator(10);
        while(it.hasNext()) {
            it.next();
            it.remove();
```

```
        }
        print("Trimmed: " + list);
        Collections.sort(list);
        print("Sorted: " + list);
        String key = list.get(7);
        int index = Collections.binarySearch(list, key);
        print("Location of " + key + " is " + index +
              ", list.get(" + index + ") = " + list.get(index));
        Collections.sort(list, String.CASE_INSENSITIVE_ORDER);
        print("Case-insensitive sorted: " + list);
        key = list.get(7);
        index = Collections.binarySearch(list, key, String.CASE_INSENSITIVE_ORDER);
        print("Location of " + key + " is " + index +
              ", list.get(" + index + ") = " + list.get(index));
    }
} /* Output:
      [one, Two, three, Four, five, six, one, one, Two, three, Four,
        five, six, one]
      Shuffled: [Four, five, one, one, Two, six, six, three, three,
        five, Four, Two, one, one]
      Trimmed: [Four, five, one, one, Two, six, six, three, three, five]
      Sorted: [Four, Two, five, five, one, one, six, six, three, three]
      Location of six is 7, list.get(7) = six
      Case-insensitive sorted: [five, five, Four, one, one, six, six,
        three, three, Two]
      Location of three is 7, list.get(7) = three
   *///:~
```

Analog zum Suchen und Sortieren bei Arrays müssen Sie, wenn Sie einen *List*-Container per Komparator sortiert haben, der `binarySearch()`-Methode denselben Komparator übergeben.

[158] Das Programm führt außerdem die statische `Collections`-Methode `shuffle()` vor, welche die Reihenfolge der Elemente eines *List*-Containers permutiert. Ein Listeniterator wird erzeugt und auf eine Startposition im permutieren Container gesetzt, um alle Elemente bis zum Ende der Liste zu entfernen.

Übungsaufgabe 40: (5) Schreiben Sie eine Klasse, die zwei `String`-Objekte enthält und das Interface *Comparable* derart implementiert, daß sich der Vergleich nur auf das erste `String`-Objekt bezieht. Füllen Sie ein Array und einen `ArrayList`-Container mit Hilfe der Generatorklasse `RandomGenerator` mit Objekten Ihrer Klasse. Zeigen Sie, daß das Sortieren korrekt funktioniert. Schreiben Sie nun einen zweiten Komparator, der sich nur auf das zweite `String`-Objekt bezieht und zeigen Sie, daß das Sortieren korrekt funktioniert. Führen Sie auch eine binäre Suche bezüglich ihrer Komparatoren durch. ■

Übungsaufgabe 41: (3) Ändern Sie die Klasse aus Übungsaufgabe 40, so daß sich ihre Objekte als Elemente in einen `HashSet`- und als Schlüssel in einen `HashMap`-Container einsetzen lassen. ■

Übungsaufgabe 42: (2) Ändern Sie Übungsaufgabe 40, so daß die Objekte nach alphabetischer Sortierung angeordnet werden. ■

18.11.2 Die unmodifiable-Methoden

[159] Es ist häufig von Nutzen, eine nur-lesbare Version eines Containers vom Typ *Collection* oder *Map* zu verwenden. Die Hilfsklasse `Collections` stellt Methoden zur Verfügung, die einen gewöhnlichen Container erwarten und eine Referenz auf eine nur-lesbare Version zurückgeben. Es gibt

mehrere Varianten dieser Methoden für die Containertypen *Collection* (falls Sie einen Container keinem spezielleren Typ zuordnen können), *List*, *Set* und *Map*. Das folgende Beispiel zeigt, wie Sie zu jedem Containertyp eine nur-lesbare Version bekommen:

```
//: containers/ReadOnly.java
// Using the Collections.unmodifiable methods.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ReadOnly {
    static Collection<String> data = new ArrayList<String>(Countries.names(6));
    public static void main(String[] args) {
        Collection<String> c =
            Collections.unmodifiableCollection(new ArrayList<String>(data));
        print(c); // Reading is OK
        //! c.add("one"); // Can't change it

        List<String> a =
            Collections.unmodifiableList(new ArrayList<String>(data));
        ListIterator<String> lit = a.listIterator();
        print(lit.next()); // Reading is OK
        //! lit.add("one"); // Can't change it

        Set<String> s =
            Collections.unmodifiableSet(new HashSet<String>(data));
        print(s); // Reading is OK
        //! s.add("one"); // Can't change it

        // For a SortedSet:
        Set<String> ss =
            Collections.unmodifiableSortedSet(new TreeSet<String>(data));

        Map<String,String> m =
            Collections.unmodifiableMap(
                new HashMap<String,String>(Countries.capitals(6)));
        print(m); // Reading is OK
        //! m.put("Ralph", "Howdy!");

        // For a SortedMap:
        Map<String,String> sm =
            Collections.unmodifiableSortedMap(
                new TreeMap<String,String>(Countries.capitals(6)));
    }
} /* Output:
      [ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO]
      ALGERIA
      [BULGARIA, BURKINA FASO, BOTSWANA, BENIN, ANGOLA, ALGERIA]
      {BULGARIA=Sofia, BURKINA FASO=Ouagadougou, BOTSWANA=Gaberone,
        BENIN=Porto-Novo, ANGOLA=Luanda, ALGERIA=Algiers}
   *///:~
```

[160] Das Aufrufen einer `unmodifiableXXX()`-Methode für einen bestimmten Containertyp bewirkt keine Typprüfung zur Übersetzungszeit. Nachdem die Transformation vollzogen ist, ruft aber jeder Methodenaufruf, der eine Änderungen des Containerinhaltes bewirken würde, eine Ausnahme vom Typ `UnsupportedOperationException` hervor.

[161] In jedem Fall müssen Sie den Container mit Daten initialisieren, *bevor* Sie ihn nur-lesbar machen. Nach der Initialisierung ist es am besten, wenn Sie statt der ursprünglichen Referenz auf den Container nur noch die von der `unmodifiableXXX()`-Methode gelieferte Referenz verwenden.

Sie umgehen dadurch das Risiko, den Inhalt des Containers versehentlich zu ändern, nach dem er in den nur-lesbaren Status „umgeschaltet“ wurde. Anderseits können Sie die Referenz auf den modifizierbaren Container als privates Feld in Ihrer Klasse speichern und die nur-lesbare Referenz auf diesen Container über eine Methode zurückgeben. Auf diese Weise können Sie den Container innerhalb der Klasse verändern, während der Inhalt außerhalb der Klasse nur gelesen werden kann.

18.11.3 Die synchronized-Methoden

[162] Das Schlüsselwort `synchronized` ist ein wichtiger Bestandteil der Threadprogrammierung, einem komplizierten Gebiet, mit dem wir uns erst in Kapitel 22 auseinandersetzen werden. An dieser Stelle soll nur vermerkt werden, daß die Hilfsklasse `Collections` über Methoden verfügt, mit denen der Zugriff auf einen ganzen Container automatisch synchronisiert werden kann. Die Syntax ähnelt der Notation der `unmodifiableXXX()`-Methoden:

```
//: containers/Synchronization.java
// Using the Collections.synchronized methods.
import java.util.*;

public class Synchronization {
    public static void main(String[] args) {
        Collection<String> c =
            Collections.synchronizedCollection(new ArrayList<String>());
        List<String> list =
            Collections.synchronizedList(new ArrayList<String>());
        Set<String> s =
            Collections.synchronizedSet(new HashSet<String>());
        Set<String> ss =
            Collections.synchronizedSortedSet(new TreeSet<String>());
        Map<String,String> m =
            Collections.synchronizedMap(new HashMap<String,String>());
        Map<String,String> sm =
            Collections.synchronizedSortedMap(new TreeMap<String,String>());
    }
} ///:~
```

Am besten wird der neu erzeugte Container unmittelbar der entsprechenden `synchronizedXXX()`-Methode übergeben, wie in diesem Beispiel. Dann besteht keine Gefahr, versehentlich die unsynchronisierte Version zu exponieren.

18.11.3.1 Schneller Abbruch

[163] Die Containerklassen von Java verfügen über einen Mechanismus, der verhindert, daß mehr als ein ~~Prozeß~~ den Inhalt eines Containers verändert. Das Problem tritt auf, wenn sich das Programm mitten in einer Iteration über die Elemente eines Containers befindet und ein anderer ~~Prozeß~~ ein Element in diesen Container einsetzt, ändert oder aus ihm löscht. Vielleicht hatten Sie dieses Element bereits eingesetzt, steht das Einsetzen noch bevor oder die Größe des Container schrumpft, nach dem Sie die `size()`-Methode aufgerufen haben. Es gibt viele Möglichkeiten für ein Unglück. Die Containerbibliothek verwendet einen Mechanismus namens *schneller Abbruch* (*fast fail*), der auf Änderungen im Container achtet, für die Ihre ~~Prozeß~~ nicht verantwortlich ist. Stellt der Mechanismus fest, daß „jemand anderes“ den Container modifiziert, so wirft er unmittelbar eine Ausnahme vom Typ `ConcurrentModificationException` aus. Dies ist der „schnelle Abbruch“, das heißt der Container versucht nicht, daß Problem später mit Hilfe eines komplizierten Algorithmus zu untersuchen.

[164] Es ist nicht schwer, den schnellen Abbruch zu demonstrieren. Es reicht aus, einen Iterator anzufordern und anschließend ein weiteres Element in den Container einzusetzen, auf den der Iterator verweist:

```
//: containers/FailFast.java
// Demonstrates the ''fail-fast'' behavior.
import java.util.*;

public class FailFast {
    public static void main(String[] args) {
        Collection<String> c = new ArrayList<String>();
        Iterator<String> it = c.iterator();
        c.add(''An object'');
        try {
            String s = it.next();
        } catch(ConcurrentModificationException e) {
            System.out.println(e);
        }
    }
} /* Output:
      java.util.ConcurrentModificationException
   *////:~
```

Die Ausnahme wird durch das Einsetzen eines Elementes in den Container hervorgerufen, *nachdem* der Iterator beim Container angefordert wurde. Die Möglichkeit, daß zwei Teile des Programms denselben Container modifizieren können, bewirkt einen unsicheren Zustand dieses Programms. Die Ausnahme weist Sie darauf hin, daß Sie Ihren Quelltext ändern sollten. In diesem Fall genügt es, den Iterator anzufordern, nachdem alle Elemente in den Container eingesetzt worden sind.

[165] Die Klassen `ConcurrentHashMap`, `CopyOnWriteArrayList` und `CopyOnWriteArraySet` basieren auf Ansätzen, die Ausnahmen vom Typ `UnsupportedOperationException` vermeiden.

18.12 Weiche, schwache und Phantomreferenzen

[166] Das Package `java.lang.ref` enthält Klassen, die flexiblere Handhabung von Objekten hinsichtlich der automatischen Speicherbereinigung gestatten. Diese Klassen sind besonders dann nützlich, wenn Sie mit großen Objekten hantieren und der vorhandene Arbeitsspeicher unter Umständen knapp wird. Drei Klassen sind von der abstrakten Klasse `Reference` abgeleitet: `SoftReference`, `WeakReference` und `PhantomReference`. Jede dieser drei Klassen stellt für die automatische Speicherbereinigung eine eigene Indirektionsebene dar, wenn ein Objekt nur über eines dieser `Reference`-Objekte erreichbar ist.

[167] Ein Objekt ist *erreichbar*, wenn es irgendwo in Ihrem Programm gefunden werden kann. Beispielsweise kann eine gewöhnliche Objektreferenz auf dem Stack liegen, die unmittelbar auf das Objekt verweist. Das Objekt kann aber auch von einem intermediären Objekt oder über eine Kette von intermediären Objekten referenziert werden. Die automatische Speicherbereinigung kann ein erreichbares Objekt nicht aufräumen, da es im Programm noch gebraucht wird. Ist ein Objekt dagegen nicht mehr erreichbar, so hat das Programm keine Möglichkeit, dieses Objekt zu verwenden und es kann sicher der automatischen Speicherbereinigung übergeben werden.

[168] `Reference`-Objekte werden verwendet, wenn Sie zwar eine Referenz auf ein Objekt behalten, es aber grundsätzlich zur automatischen Speicherbereinigung freigeben möchten. Sie können mit einem solchen Objekt weiterarbeiten, gestatten aber, daß es aufgeräumt wird, wenn die Erschöpfung des Arbeitsspeichers unmittelbar bevorsteht.

[169] Ein `Reference`-Objekt ist ein Stellvertreter zwischen Ihnen und der gewöhnlichen Objektreferenz. Außerdem darf es keine gewöhnlichen Referenzen (die nicht in einem `Reference`-Objekt verpackt sind) auf das Objekt mehr geben. Die automatische Speicherbereinigung kann ein Objekt nicht aufräumen, wenn es noch eine gewöhnliche Referenzen darauf gibt.

[170] Die „Bindungsstärke“ nimmt in der Reihenfolge `SoftReference` („weiche Referenz“), `WeakReference` („schwache Referenz“) und `PhantomReference` („Phantomreferenz“) ab und jede Klasse stellt einen anderen Grad von Erreichbarkeit dar. Weiche Referenzen werden verwendet, um arbeitsspeicherempfindliche Zwischenspeicher zu implementieren. Schwache Referenzen dienen der Implementierung „kanonisierter *`Map`*-Container“. Ein solcher Container kann an mehreren Stellen im Programm zugleich verwendet werden, um Speicherplatz zu sparen, ohne dabei den mehrfachen Zugriff auf seine Schlüssel und Werte zu verweigern. Phantomreferenzen werden verwendet, um Aufräumarbeiten vor der Zerstörung eines Objektes flexibler einplanen zu können, als über den Finalisierungsmechanismus von Java.

[171] Bei Objekten der Klassen `SoftReference` und `WeakReference` haben Sie die Wahl, ob Sie sie in einem Container vom Typ `ReferenceQueue` (dem für die Aufräumarbeiten vor der Zerstörung des Objektes verwendeten Instrument) deponieren oder nicht. Objekte der Klasse `PhantomReference` müssen dagegen in einem Container dieses Typs registriert werden. Das folgende Beispiel führt die drei Referenztypen vor:

```
//: containers/References.java
// Demonstrates Reference objects
import java.lang.ref.*;
import java.util.*;

class VeryBig {
    private static final int SIZE = 10000;
    private long[] la = new long[SIZE];
    private String ident;
    public VeryBig(String id) { ident = id; }
    public String toString() { return ident; }
    protected void finalize() {
        System.out.println("Finalizing " + ident);
    }
}

public class References {
    private static ReferenceQueue<VeryBig> rq = new ReferenceQueue<VeryBig>();
    public static void checkQueue() {
        Reference<? extends VeryBig> inq = rq.poll();
        if(inq != null)
            System.out.println("In queue: " + inq.get());
    }
    public static void main(String[] args) {
        int size = 10;
        // Or, choose size via the command line:
        if(args.length > 0)
            size = new Integer(args[0]);
        LinkedList<SoftReference<VeryBig>> sa =
            new LinkedList<SoftReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            sa.add(new SoftReference<VeryBig>(new VeryBig("Soft " + i), rq));
            System.out.println("Just created: " + sa.getLast());
            checkQueue();
        }
        LinkedList<WeakReference<VeryBig>> wa =
```

```
            new LinkedList<WeakReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            wa.add(new WeakReference<VeryBig>(new VeryBig("Weak " + i), rq));
            System.out.println("Just created: " + wa.getLast());
            checkQueue();
        }
        SoftReference<VeryBig> s =
            new SoftReference<VeryBig>(new VeryBig("Soft"));
        WeakReference<VeryBig> w =
            new WeakReference<VeryBig>(new VeryBig("Weak"));
        System.gc();
        LinkedList<PhantomReference<VeryBig>> pa =
            new LinkedList<PhantomReference<VeryBig>>();
        for(int i = 0; i < size; i++) {
            pa.add(new PhantomReference<VeryBig>(new VeryBig("Phantom " + i), rq));
            System.out.println("Just created: " + pa.getLast());
            checkQueue();
        }
    }
} /* (Execute to see output) *////:~
```

Beim Ausführen dieses Programms (Sie können die Ausgabe in eine Textdatei umleiten, um sie seitenweise zu lesen) sehen Sie, daß die Objekte von der automatischen Speicherbereinigung aufgeräumt werden, obwohl Sie über das `Reference`-Objekt noch immer Zugriff haben (die `get()`-Methode liefert die eigentliche Objektreferenz). Der `ReferenceQueue`-Container erzeugt stets ein `Reference`-Objekt mit dem Inhalt `null`. Um diesen Mechanismus zu nutzen, leiten Sie eine eigene Klasse von einer der drei Unterklassen von `Reference` ab und statten sie mit gewünschten Funktionalität aus.

18.12.1 Die Klasse WeakHashMap

[172] Die Containerbibliothek enthält einen speziellen Typ für schwache Referenzen unter dem Interface *`Map`*: `java.util.WeakHashMap`. Diese Klasse erleichtert das Erzeugen eines kanonisierten *`Map`*-Containers. Ein solcher Container spart Arbeitsspeicher, indem nur ein Objekt eines bestimmten Wertes erzeugt wird. Wenn das Programm den Wert braucht, schlägt es das vorhandene Objekt im Container nach und verwendet es (statt ein neues Objekt zu erzeugen). Der Container kann die Werte während seiner Initialisierung erzeugen, generiert sie aber wahrscheinlicher erst bei Bedarf.

[173] Bei diesem arbeitsspeichersparenden Verfahren gestattet die Klasse `WeakHashMap` der automatischen Speicherbereinigung, ihre Schlüssel und Werte automatisch aufzuräumen. Bei den Schlüsseln und Werten, die in einem `WeakHashMap`-Container gespeichert werden, ist keine zusätzliche Behandlung erforderlich; der Container verpackt sie automatisch in `WeakReference`-Objekten. Ein Schlüssel/Wert-Paar wird aufgeräumt, wenn sein Schlüssel nicht mehr gebraucht wird, wie das folgende Beispiel zeigt:

```
//: containers/CanonicalMapping.java
// Demonstrates WeakHashMap.
import java.util.*;

class Element {
    private String ident;
    public Element(String id) { ident = id; }
    public String toString() { return ident; }
    public int hashCode() { return ident.hashCode(); }
    public boolean equals(Object r) {
```

```
        return r instanceof Element &&
            ident.equals(((Element) r).ident);
    }
    protected void finalize() {
        System.out.println("Finalizing " +
                           getClass().getSimpleName() + " " + ident);
    }
}

class Key extends Element {
    public Key(String id) { super(id); }
}

class Value extends Element {
    public Value(String id) { super(id); }
}

public class CanonicalMapping {
    public static void main(String[] args) {
        int size = 1000;
        // Or, choose size via the command line:
        if(args.length > 0)
            size = new Integer(args[0]);
        Key[] keys = new Key[size];
        WeakHashMap<Key,Value> map = new WeakHashMap<Key,Value>();
        for(int i = 0; i < size; i++) {
            Key k = new Key(Integer.toString(i));
            Value v = new Value(Integer.toString(i));
            if(i % 3 == 0)
                keys[i] = k; // Save as "real" references
            map.put(k, v);
        }
        System.gc();
    }
} /* (Execute to see output) *////:~
```

[174] Die Klasse `Key` muß eine `hashCode()`- und eine `equals()`-Methode besitzen, um als Schlüssel einer hashbasierten Datenstruktur verwendet werden zu können. Die `hashCode()`-Methode wurde in Abschnitt 18.9 beschrieben.

[175] Beim Aufrufen des Programms können Sie beobachten, daß die automatische Speicherbereinigung jeden dritten Schlüssel ausläßt, weil das von `keys` referenzierte Array eine gewöhnliche Referenz auf diesen Schlüssel speichert, das zugehörige Objekt also noch nicht aufgeräumt werden kann.

18.13 Die veralteten Containerklassen aus Java 1.0/1.1

[176] Bedauerlicherweise wurden viele Programme und Anwendungen entwickelt, die Containerklassen aus den Java-Versionen 1.0 und 1.1 verwenden. Selbst in neu geschriebenen Quelltexten finden sich hin und wieder Vorkommen dieser Klassen. Obwohl Sie diese alten Containerklassen beim Entwickeln neuer Programme nicht mehr verwenden sollten, müssen Sie sich also ihrer Eigenschaften und Fähigkeiten bewußt sein. Der Funktionsumfang der alten Containerklassen ist andererseits so begrenzt, daß es nicht viel darüber zu sagen gibt. Da die alten Containerklassen nicht mehr zeitgemäß sind, will ich zu unterlassen versuchen, einige ihrer unschönen Design-Entscheidungen überzubetonen.

18.13.1 Die Klasse Vector und das Interface Enumeration

[177] Die Klasse `Vector` war der einzige Containertyp bei Java 1.0/1.0, der seine Größe selbständig erweitern konnte und wurde dementsprechend häufig verwendet. Die Schwachstellen dieser Klasse sind zu zahlreich, um an dieser Stelle beschrieben werden zu können (siehe hierzu die erste Auflage der englischen Ausgabe, die Sie unter der Webadresse *http://www.mindview.net* kostenlos herunterladen können). Sie können sich die Klasse `Vector` wie eine `ArrayList`-Klasse mit ungeschickt gewählten Methodennamen vorstellen. In der überarbeiteten Containerbibliothek wurde `Vector` dahingehend verbessert, daß die Klasse die Interfaces *`Collection`* und *`List`* implementiert. Diese Änderung ist widernatürlich, da sie einen Programmierer zu der Fehlannahme verleiten kann, die Klasse `Vector` sei im Hinblick auf ihre Funktionalität verbessert worden, wobei die Klasse eigentlich nur vorhanden ist, um die Funktionstüchtigkeit älterer Programme und Anwendungen zu gewährleisten.

[178] Bei Java 1.0/1.0 wurde für den Begriff „Iterator“ die neue Bezeichnung „Enumeration“ erfunden, statt eine Benennung zu wählen, mit der jeder bereits vertraut ist, nämlich „Iterator“. Das Interface *`Enumeration`* ist kleiner als *`Iterator`* und deklariert nur zwei Methoden mit längeren Namen: Die Methode `hasMoreElements()` gibt `true` zurück, wenn der *`Enumeration`*-Container weitere Elemente enthält. Die Methode `nextElement()` gibt das nächste Element des *`Enumeration`*-Containers zurück, falls eines vorhanden ist, andernfalls ruft die Methode eine Ausnahme hervor.

[179] *`Enumeration`* ist nur ein Interface, keine Implementierung und wird sogar bei neuen Bibliotheken noch verwendet. Das ist zwar bedauerlich, in der Regel aber unbedenklich. Auch wenn Sie in Ihren eigenen Programmen stets das Interface *`Iterator`* wählen sollten, müssen Sie damit rechnen, daß eine Bibliothek die Übergabe von Objekten des Typs *`Enumeration`* erwartet.

[180] Das folgende Beispiel zeigt, wie Sie mit Hilfe der statischen `Collections`-Methode `enumeration()` aus einem *`Collection`*-Container einen *`Enumeration`*-Container erzeugen können:

```
//: containers/Enumerations.java
// Java 1.0/1.1 Vector and Enumeration.
import java.util.*;
import net.mindview.util.*;

public class Enumerations {
    public static void main(String[] args) {
        Vector<String> v = new Vector<String>(Countries.names(10));
        Enumeration<String> e = v.elements();
        while(e.hasMoreElements())
            System.out.print(e.nextElement() + ", ");
        // Produce an Enumeration from a Collection:
        e = Collections.enumeration(new ArrayList<String>());
    }
} /* Output:
      ALGERIA, ANGOLA, BENIN, BOTSWANA, BULGARIA, BURKINA FASO, BURUNDI,
        CAMEROON, CAPE VERDE, CENTRAL AFRICAN REPUBLIC,
   *///:~
```

Die `elements()`-Methode gibt eine Referenz auf einen Container vom Typ *`Enumeration`* zurück, über dessen Elemente Sie vorwärts iterieren können.

[181] Die letzte Zeile erzeugt einen `ArrayList`-Container und ruft die statische `enumeration()`-Methode der Hilfsklasse `Collections` auf, um den `ArrayList`-Container mit Hilfe seines Iterators an das Interface *`Enumeration`* anzupassen. Sie können einen älteren Quelltext, der den Typ *`Enumeration`* verlangt, auch mit den neuen Containerklassen kombinieren.

18.13.2 Die Klasse Hashtable

[182] Wie Sie bei den Performanzvergleichen in Abschnitt 18.10 gesehen haben, ist die Ähnlichkeit der Klassen `Hashtable` und `HashMap` sehr ausgeprägt, bis hin zu den Methodennamen. Es gibt keinen Grund mehr, in einem neuen Quelltext `Hashtable` anstelle von `HashMap` zu wählen.

18.13.3 Die Klasse Stack

[183] Das Konzept des Stapelspeichers (*stack*) wurde in Abschnitt 12.8 vorgestellt, zusammen mit der Containerklasse `LinkedList`. An der Klasse `Stack` bei Java 1.0/1.0 ist sonderbar, daß sie die Klasse `Vector` nicht etwa durch Komposition einbindet, sondern von `Vector` abgeleitet ist. Die Klasse `Stack` hat somit die Eigenschaften und das Verhalten von `Vector`, zusammen mit einigen zusätzlichen Eigenschaften und Fähigkeiten durch das Verhalten eines Stapelspeichers. Es ist schwierig auszumachen, ob die Designer diesen Ansatz für eine besonders nützliche Lösung gehalten und daher bewußt gewählt, oder sich naiv für diesen Entwurf entschieden haben. Der Ansatz wurde jedenfalls nicht durchgesehen, bevor er überstürzt in die Distribution aufgenommen wurde, so daß dieses schlechte Design noch immer vorhanden ist (Sie sollten keinen Gebrauch davon machen).

[184] Das folgende einfache Beispiel deponiert die `String`-Darstellung der Elemente eines Aufzählungstyps in einem `Stack`-Container. Das Beispiel zeigt, daß Sie auch einen `LinkedList`-Container als Stapelspeicher bedienen oder die in Abschnitt 12.8 definierte Klasse `Stack` wählen können:

```
//: containers/Stacks.java
// Demonstration of Stack Class.
import java.util.*;
import static net.mindview.util.Print.*;

enum Month { JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE,
             JULY, AUGUST, SEPTEMBER, OCTOBER, NOVEMBER }

public class Stacks {
    public static void main(String[] args) {
        Stack<String> stack = new Stack<String>();
        for(Month m : Month.values())
            stack.push(m.toString());
        print("stack = " + stack);
        // Treating a stack as a Vector:
        stack.addElement("The last line");
        print("element 5 = " + stack.elementAt(5));
        print("popping elements:");
        while(!stack.empty())
            printnb(stack.pop() + " ");

        // Using a LinkedList as a Stack:
        LinkedList<String> lstack = new LinkedList<String>();
        for(Month m : Month.values())
            lstack.addFirst(m.toString());
        print("lstack = " + lstack);
        while(!lstack.isEmpty())
            printnb(lstack.removeFirst() + " ");

        // Using the Stack class from
        // the Holding Your Objects Chapter:
        net.mindview.util.Stack<String> stack2 =
            new net.mindview.util.Stack<String>();
        for(Month m : Month.values())
            stack2.push(m.toString());
```

```
        print("stack2 = " + stack2);
        while(!stack2.empty())
            printnb(stack2.pop() + " ");
    }
} /* Output:
      stack = [JANUARY, FEBRUARY, MARCH, APRIL, MAY, JUNE, JULY, AUGUST,
        SEPTEMBER, OCTOBER, NOVEMBER]
      element 5 = JUNE
      popping elements:
      The last line NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL
        MARCH FEBRUARY JANUARY lstack = [NOVEMBER, OCTOBER, SEPTEMBER, AUGUST,
        JULY, JUNE, MAY, APRIL, MARCH, FEBRUARY, JANUARY]
      NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH FEBRUARY
        JANUARY stack2 = [NOVEMBER, OCTOBER, SEPTEMBER, AUGUST, JULY, JUNE,
        MAY, APRIL, MARCH, FEBRUARY, JANUARY]
        NOVEMBER OCTOBER SEPTEMBER AUGUST JULY JUNE MAY APRIL MARCH FEBRUARY
      JANUARY
   *///:~
```

Die Konstanten des Aufzählungstyps `Month` werden per `toString()` in ihre `String`-Darstellung umgewandelt, mit Hilfe der `push()`-Methode in den `Stack`-Container eingesetzt und später per `pop()` am oberen Ende des Stapelspeichers wieder entnommen. Das Beispiel betont, daß ein `Stack`-Container auch `Vector`-Methoden unterstützt. Das ist aufgrund der Verbungsbeziehung zwischen den Klassen `Stack` und `Vector` möglich (`Stack` ist von `Vector` abgeleitet). Alle Operationen eines `Vector`-Containers sind somit auch bei einem `Stack`-Container verfügbar, beispielsweise die Methode `elementAt()`.

[185] Wie bereits mehrmals erwähnt, sollten Sie einen `LinkedList`-Container oder die auf einem `LinkedList`-Container aufbauende Klasse `net.mindview.util.Stack` wählen, wenn Sie Stapelspeicherverhalten brauchen.

18.13.4 Die Klasse BitSet

[186] Ein Container vom Typ `BitSet` wird verwendet, um viele binäre Informationen effizient zu speichern. Die Klasse `BitSet` ist nur hinsichtlich ihrer Größe effizient. Wenn Sie effizienten Zugriff brauchen, ist `BitSet` geringfügig langsamer als ein natives Array.

[187] Die minimale Größe eines `BitSet`-Containers entspricht der Länge des Datentyps `long`, also 64 Bit. Daraus folgt, daß Sie Platz verschwenden, wenn Sie einen `BitSet`-Container mit weniger Informationen füllen, etwa nur 8 Bit. Wenn es auf Speicherplatz ankommt, schreiben Sie besser eine eigene Klasse oder verwenden ein Array, um Ihre Schalter zu speichern. (Dieser Fall tritt nur ein, wenn Sie viele Objekte erzeugen, die Listen mit binären Informationen enthalten und sollte ausschließlich aufgrund der Untersuchung eines Profilers oder einer anderen Metrik entschieden werden. Wenn Sie eine solche Entscheidung fällen, weil Sie den Eindruck haben, daß etwas zuviel Speicherplatz verbraucht, so entwerfen Sie letztlich nutzlose Komplexität und verschwenden eine Menge Zeit.)

[188] Eine normaler Container dehnt sich aus, wenn Sie weitere Elemente einsetzen. Auch der Containertyp `BitSet` hat diese Fähigkeit. Das folgende Beispiel zeigt, wie ein solcher Container funktioniert:

```
//: containers/Bits.java
// Demonstration of BitSet.
import java.util.*;
```

```
import static net.mindview.util.Print.*;

public class Bits {
    public static void printBitSet(BitSet b) {
        print("bits: " + b);
        StringBuilder bbits = new StringBuilder();
        for(int j = 0; j < b.size() ; j++)
            bbits.append(b.get(j) ? "1" : "0");
        print("bit pattern: " + bbits);
    }
    public static void main(String[] args) {
        Random rand = new Random(47);
        // Take the LSB of nextInt():
        byte bt = (byte) rand.nextInt();
        BitSet bb = new BitSet();
        for(int i = 7; i >= 0; i--)
            if(((1 << i) & bt) != 0)
                bb.set(i);
            else
                bb.clear(i);
        print("byte value: " + bt);
        printBitSet(bb);

        short st = (short) rand.nextInt();
        BitSet bs = new BitSet();
        for(int i = 15; i >= 0; i--)
            if(((1 << i) & st) != 0)
                bs.set(i);
            else
                bs.clear(i);
        print("short value: " + st);
        printBitSet(bs);

        int it = rand.nextInt();
        BitSet bi = new BitSet();
        for(int i = 31; i >= 0; i--)
            if(((1 << i) & it) != 0)
                bi.set(i);
            else
                bi.clear(i);
        print("int value: " + it);
        printBitSet(bi);

        // Test bitsets >= 64 bits:
        BitSet b127 = new BitSet();
        b127.set(127);
        print("set bit 127: " + b127);
        BitSet b255 = new BitSet(65);
        b255.set(255);
        print("set bit 255: " + b255);
        BitSet b1023 = new BitSet(512);
        b1023.set(1023);
        b1023.set(1024);
        print("set bit 1023: " + b1023);
    }
} /* Output:
      byte value: -107
      bits: {0, 2, 4, 7}
      bit pattern: 1010100100000000000000000000000000000000000000000000000000000000
```

```
    short value: 1302
    bits: {1, 2, 4, 8, 10}
    bit pattern: 01101000101000000000000000000000000000000000000000000000000000000
    int value: -2014573909
    bits: {0, 1, 3, 5, 7, 9, 11, 18, 19, 21, 22, 23, 24, 25, 26, 31}
    bit pattern: 11010101010100000011011111100001000000000000000000000000000000000
    set bit 127: {127}
    set bit 255: {255}
    set bit 1023: {1023, 1024}
*///:~
```

Ein Zufallsgenerator erzeugt `byte`-, `short`- und `int`-Werte, die in den `BitSet`-Containern in die entsprechenden Bitmuster umgewandelt werden. Da ein `BitSet`-Container mindestens 64 Bit lang ist, bewirkt keiner dieser Werte eine Erweiterung des Speichervermögens. Anschließend werden einige größere `BitSet`-Container erzeugt. Sie sehen, daß ein Container vom Typ `BitSet` bei Bedarf verlängert wird.

[189] Bei einer festen Anzahl von Schaltern, die Sie jeweils mit einem Namen versehen können, ist der Containertyp `EnumSet` (siehe Kapitel 20) in der Regel eine bessere Wahl als `BitSet`, da Sie sich bei einem `EnumSet`-Container auf die Namen anstelle der Bitpositionen beziehen können und somit Fehler vermeiden. `EnumSet` schützt Sie auch davor, versehentlich neue Schalterpositionen anzulegen, wodurch einige ernsthafte und schwierig zu findende Fehler verursacht werden können. Die einzigen Situationen, in denen Sie einen `BitSet`- anstelle eines `EnumSet`-Containers wählen sollten, treten auf, wenn Sie nicht wissen, wieviele Schalter zur Laufzeit benötigt werden, wenn Sie nicht jedem Schalter einen Namen geben können oder wenn Sie eine der speziellen Operationen des Containertyps `BitSet` brauchen (siehe API-Dokumentation der Klassen `BitSet` und `EnumSet`).

18.14 Zusammenfassung

[190] Die Containerbibliothek ist wohl die wichtigste Bibliothek einer objektorientierten Programmiersprache. Die meisten Programme und Anwendungen nutzen mehr Container als andere Bibliothekskomponenten. Bei einigen Sprachen, zum Beispiel bei Python, sind die fundamentalen Containertypen (Listen, Schlüssel/Wert-Abbildungen und Mengen) sogar bereits eingebaut.

[191] Sie haben in Kapitel 12 gesehen, daß sich viele sehr interessante Aufgaben mit Containern mühelos lösen lassen. Es gibt allerdings einen Punkt, an dem Sie gezwungermaßen mehr über Container wissen müssen, um sie zweckmäßig verwenden zu können. Insbesondere müssen Sie genug über Hashverfahren wissen, um die `hashCode()`-Methode selbst implementieren zu können (und zu verstehen, wann dieser Schritt notwendig wird). Darüber hinaus müssen Sie mit den Eigenschaften und Fähigkeiten der verschiedenen Implementierungen der Containerinterfaces vertraut sein, damit Sie eine zu Ihren Anforderungen passende Wahl treffen können. In diesem Kapitel wurden diese Konzepte abgedeckt und einige zusätzliche nützliche Einzelheiten der Containerbibliothek diskutiert. Sie sollten nun einigermaßen gut vorbereitet sein, um die Containerklassen von Java in Ihrer alltäglichen Programmierarbeit einsetzen zu können.

[192] Das Design einer Containerbibliothek ist eine schwierige Aufgabe (dies gilt für die meisten Design-Probleme in Bibliotheken). In C++ deckt die Containerbibliothek die Basis mit einer Vielzahl unterschiedlicher Klassen ab. Dieser Ansatz ist zwar besser, als das was vor den Containerklassen in C++ vorhanden war (nämlich nichts), läßt sich aber nicht gut auf Java übertragen. Am anderen extremen Ende gibt es eine Containerbibliothek, die nur aus einer einzigen Klasse namens `container` besteht, welche sich zugleich sowohl wie eine Aneinandereihung von Elementen als auch wie ein assoziatives Array verhält. Die Containerbibliothek von Java schlägt einen Mittelweg ein:

Die volle Funktionalität einer ausgereiften Containerbibliothek, aber leichter zu erlernen, als die Containerklassen von C++ und ähnliche Containerbibliotheken. Das Ergebnis kann stellenweise ein wenig sonderbar wirken. Im Gegensatz zu den Entscheidungen in den frühen Java-Bibliotheken, sind diese Auffälligkeiten aber keine Mißgeschicke, sondern sorgfältig abgewogene Entscheidungen unter Berücksichtigung der daraus erwachsenden Komplexität.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 19

Ein-/Ausgabe

Inhaltsübersicht

[0] Der Entwurf eines guten Ein-/Ausgabesystems gehört zu den anspruchsvolleren Aufgaben beim Entwickeln einer Programmiersprache. Das zeigt sich in der Vielzahl unterschiedlicher Ansätze.

[1] Die Herausforderung scheint darin zu bestehen, alle Möglichkeiten abzudecken. Es gibt nicht nur verschiedene Datenquellen und -ziele, mit denen Sie kommunizieren müssen (Dateien, die Konsole, Netzwerkverbindungen, usw.), sondern auch viele verschiedene Zugriffsmöglichkeiten (sequentiell, per Dateizeiger, gepuffert, byteorientiert, zeichenorientiert, zeilenweise, wortweise, usw.)

[2] Die Entwickler der Ein-/Ausgabebibliothek von Java haben das Problem durch eine Vielzahl von Klassen gelöst. Die Ein-/Ausgabebibliothek umfaßt derart viele Klassen, daß sie auf den ersten Blick durchaus furchterregend sein kann (ironischerweise *verhindert* das Design der Ein-/Ausgabebibliothek von Java sogar, daß die Anzahl der Klassen explodiert). Nach Java 1.0 hat die Ein-/Ausgabebibliothek außerdem eine wesentliche Änderung erfahren, als der ursprünglich byteorientierten Bibliothek die zeichenorientierten, unicodebasierten Ein-/Ausgabeklassen hinzugefügt wurden. Das `java.nio`-Package und seine Klassen wurden aufgenommen, um die Performanz zu verbessern und die Funktionalität zu erweitern. (Das „n“ steht für „neu“, also „neue Ein-/Ausgabebibliothek“. Wir werden diese Bezeichnung wohl auch in Zukunft noch jahrelang benutzen, obwohl das `java.nio`-Package bereits in Version 1.4 des Java Development Kits eingeführt wurde, mittlerweile also „alt“ ist.) Sie müssen sich mit einer gewissen Auswahl von Klassen vertraut machen, um den Entwurf der Ein-/Ausgabebibliothek von Java zu verstehen, bevor Sie sie zweckmäßig anwenden können. Es ist außerdem wichtig, die Entwicklungsgeschichte der Ein-/Ausgabebibliothek von Java zu verstehen, auch wenn Sie auf dem Standpunkt stehen „Langweilen Sie mich nicht mit Geschichte. Zeigen Sie mir lieber, wie ich mit der Bibliothek umgehen muß“. Ohne den geschichtlichen Hintergrund, werden Sie bei einigen der Klassen durcheinanderbringen, unter welchen Voraussetzungen Sie sie verwenden sollen und wann nicht.

[3] Dieses Kapitel ist eine Einführung in die Vielfalt und Anwendung der Ein-/Ausgabeklassen der Standardbibliothek von Java.

19.1 Die Klasse File

[4] Bevor wir uns den Klassen zuwenden, die tatsächlich Daten aus Strömen lesen oder an diese übergeben, betrachten wir eine Hilfsklasse für Datei- und Verzeichnisangelegenheiten.

[5] Der Klassename `java.io.File` ist trügerisch, da sich die Klasse nicht auf Dateien bezieht. Eigentlich wäre „FilePath“ ein besserer Name für diese Klasse gewesen. Ein `File`-Objekt repräsentiert entweder den Namen einer bestimmten Datei oder die Namen einer Menge von Dateien in einem Verzeichnis. Im letzteren Fall können Sie die Menge von Dateien mit Hilfe der `list()`-Methode abfragen, die ein `String`-Array zurückgibt. Es ist sinnvoll, statt eines Objektes einer der flexibleren Containerklassen ein Array zurückzugeben, da die Anzahl der Elemente fixiert ist und Sie ein weiteres `File`-Objekt erzeugen müssen, wenn Sie den Inhalt eines anderen Verzeichnisses darstellen möchten. Dieser Abschnitt zeigt ein Anwendungwendungsbeispiel für die Klasse `File`, zusammen mit dem Interface *`java.io.FilenameFilter`*.

19.1.1 Inhalt eines Verzeichnisses

[6] Angenommen, Sie möchten den Inhalt eines Verzeichnisses auflisten. Ein `File`-Objekt kann auf zweierlei Weise verwendet werden. Die Methode `list()` liefert, ohne Argumente aufgerufen, den kompletten Inhalt ihres `File`-Objektes. Wenn Sie dagegen eine eingeschränkte Auswahl brauchen, zum Beispiel alle Dateien mit der Endung *.java*, so müssen Sie einen „Verzeichnisfilter“ (*directory filter*) verwenden, das heißt eine Klasse, die sich um die Auswahl der anzuzeigenden Einträge des `File`-Objektes kümmert.

[7] Das Ergebnis wurde beim folgenden Beispiel mit Hilfe der `Arrays`-Methode `sort()` sowie des Komparators `String.CASE_INSENSITIVE_ORDER` mühelos alphabetisch sortiert:

```
//: io/DirList.java
// Display a directory listing using regular expressions.
// {Args: ''D.*java''}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList {
    public static void main(String[] args) {
        File path = new File('.');
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new DirFilter(args[0]));
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for(String dirItem : list)
            System.out.println(dirItem);
    }
}

class DirFilter implements FilenameFilter {
    private Pattern pattern;
    public DirFilter(String regex) {
        pattern = Pattern.compile(regex);
    }
    public boolean accept(File dir, String name) {
        return pattern.matcher(name).matches();
```

```
    }
} /* Output:
      DirectoryDemo.java
      DirList.java
      DirList2.java
      DirList3.java
   *///:~
```

Die Klasse `io.DirFilter` implementiert das Interface *`java.io.FilenameFilter`*. Beachten Sie, wie einfach *`FilenameFilter`* aufgebaut ist:

```
public interface FilenameFilter {
    boolean accept(File dir, String name);
}
```

[8] Die einzige Existenzberechtigung der Klasse `DirFilter` besteht darin, die `accept()`-Methode zu liefern, die von `list()` aufgerufen wird, um auszuwerten, welche Dateinamen aufgelistet werden sollen. Dieser Ansatz wird häufig als Rückruffunktion (*call back*) bezeichnet. Es handelt sich genau genommen um ein Beispiel für das Entwurfsmuster *Strategy*: Die `list()`-Methode implementiert die Grundfunktionalität und die `accept()`-Methode „die Strategie“, hier vom Typ *`FilenameFilter`*, um den Algorithmus zu vervollständigen, so daß die `list()`-Methode ihre Aufgabe erfüllen kann. Da `list()` ein Argument vom Typ *`FilenameFilter`* erwartet, können Sie (selbst zur Laufzeit) ein Objekt einer beliebigen Klasse übergeben, die das Interface *`FilenameFilter`* implementiert, um festzulegen, wie die `list()`-Methode arbeitet. Eine „Strategie“ macht das Verhalten eines Quelltextes flexibel.

[9] Die *`FilenameFilter`*-Methode `accept()` erwartet zwei Argumente, nämlich ein `File`-Objekt, welches das Verzeichnis repräsentiert, das eine bestimmte Datei enthält und ein `String`-Object, welches den Namen dieser Datei angibt. Die `list()`-Methode ruft pro Datei im Verzeichnis einmal die `accept()`-Methode auf, um festzustellen, ob die Datei aufgelistet werden soll. Die Entscheidung richtet sich nach dem `boolean`-Wert, den `accept()` zurückgibt.

[10] Die `accept()`-Methode der Klasse `DirFilter` verwendet ein `Matcher`-Objekt, um der Dateiname zum regulären Ausdruck `regex` paßt. Die `list()`-Methode ruft `accept()` auf und gibt ein `String`-Array zurück.

19.1.1.1 Anonyme innere Klassen

[11] Das obige Beispiel (`DirList`) eignet sich ideal, um das Programm so umzuschreiben, daß es eine anonyme innere Klasse verwendet (siehe Kapitel 11). Wir legen zunächst eine Methode namens `filter()` an, die eine Referenz vom Typ *`FilenameFilter`* zurückgibt:

```
//: io/DirList2.java
// Uses anonymous inner classes.
// {Args: ''D.*java''}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList2 {
    public static FilenameFilter filter(final String regex) {
        // Creation of anonymous inner class:
        return new FilenameFilter() {
                private Pattern pattern = Pattern.compile(regex);
                public boolean accept(File dir, String name) {
                    return pattern.matcher(name).matches();
```

```
                }
            }; // End of anonymous inner class
        }
        public static void main(String[] args) {
            File path = new File(".");
            String[] list;
            if(args.length == 0)
                list = path.list();
            else
                list = path.list(filter(args[0]));
            Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
            for(String dirItem : list)
                System.out.println(dirItem);
        }
    } /* Output:
          DirectoryDemo.java
          DirList.java
          DirList2.java
          DirList3.java
       *///:~
```

Beachten Sie, daß das Argument der `filter()`-Methode als `final` deklariert sein muß. Diese Forderung ist durch die anonyme innere Klasse gegeben, damit sie ein Objekt außerhalb ihres Geltungsbereiches verwenden kann.

[12] Dieses Design ist eine Verbesserung gegenüber `DirList`, da die Implementierung des *`FilenameFilter`*-Interfaces nun stärker an die Klasse `DirList2` gebunden ist. Der Ansatz läßt sich noch um einen weiteren Schritt fortführen, wobei die anonyme innere Klasse als Argument im Aufruf der `list()`-Methode definiert wird. Die Klasse `DirList3` wird dadurch etwas kürzer als `DirList2`:

```
//: io/DirList3.java
// Building the anonymous inner class "in-place."
// {Args: "D.*java"}
import java.util.regex.*;
import java.io.*;
import java.util.*;

public class DirList3 {
    public static void main(final String[] args) {
        File path = new File(".");
        String[] list;
        if(args.length == 0)
            list = path.list();
        else
            list = path.list(new FilenameFilter() {
                    private Pattern pattern = Pattern.compile(args[0]);
                    public boolean accept(File dir, String name) {
                        return pattern.matcher(name).matches();

                }});
        Arrays.sort(list, String.CASE_INSENSITIVE_ORDER);
        for(String dirItem : list)
            System.out.println(dirItem);
    }
} /* Output:
      DirectoryDemo.java
      DirList.java
      DirList2.java
```

```
    DirList3.java
*///:~
```

Das Argument der `main()`-Methode ist nun als `final` deklariert, da die anonyme innere Klasse `args[0]` direkt verwendet.

[13] Die Beispiele `DirList2` und `DirList3` zeigen das Definieren einmalig gebrauchter Klassen in Gestalt anonymer innerer Klassen. Dieser Ansatz hat unter anderem den Vorteil, daß die Anweisungen zur Lösung einer bestimmten Aufgabe an einer Stelle isoliert sind. Auf der anderen Seite ist ein Quelltext mit anonymen inneren Klassen nicht immer einfach zu lesen. Gebrauchen Sie diesen Ansatz also mit Vernunft.

Übungsaufgabe 1: (3) Ändern Sie *DirList.java* (oder eine seiner beiden Varianten) so, daß der *`FilenameFilter`* beliebige Dateien öffnet, liest (verwenden Sie hierfür die Hilfsklasse `TextFile`) und akzeptiert, wenn sie eine der auf der Kommandozeile übergebenen Zeichenketten enthalten. ■

Übungsaufgabe 2: (2) Legen Sie eine Klasse namens `SortedDirList` an, deren Konstruktor ein `File`-Objekt erwartet. Die Klasse listet die in diesem Verzeichnis enthaltenen Dateien sortiert auf. Legen Sie zwei überladene `list()`-Methoden an. Die erste Version listet den gesamten Verzeichnisinhalt auf, die zweite Version dagegen nur die Teilmenge, die zu ihrem Argument paßt (regulärer Ausdruck). ■

Übungsaufgabe 3: (3) Ändern Sie *DirList.java* (oder eine seiner beiden Varianten) so, daß das Programm die Größen aller ausgewählten Dateien addiert. ■

19.1.2 Directory: Eine Hilfsklasse für Operationen auf Verzeichnissen

[14] Eine häufige Programmieraufgabe besteht darin, Operationen auf ausgewählten Dateien in einem bestimmten Verzeichnis oder einem ganzen Verzeichnisbaum durchzuführen. Es wäre nützlich, eine Klasse oder Methode zur Verfügung zu haben, die diese Mengen von Dateien zusammenstellt. Die folgende Hilfsklasse `net.mindview.util.Directory` erzeugt entweder mittels ihrer `local()`-Methode ein Array von `File`-Objekten, welche die Dateien eines bestimmten Verzeichnisses repräsentieren oder mittels ihrer `walk()`-Methode ein *`List<File>`*-Objekt, welches den Verzeichnisbaum mit einem bestimmten Verzeichnis als Wurzel repräsentiert. (`File`-Objekte sind besser als Dateinamen geeignet, da sie mehr Informationen enthalten.) Die Auswahl der Dateien ist durch einen regulären Ausdruck definiert, den Sie angeben müssen:

```
//: net/mindview/util/Directory.java
// Produce a sequence of File objects that match a
// regular expression in either a local directory,
// or by walking a directory tree.
package net.mindview.util;
import java.util.regex.*;
import java.io.*;
import java.util.*;

public final class Directory {
    public static File[]
        local(File dir, final String regex) {
        return dir.listFiles(new FilenameFilter() {
                private Pattern pattern = Pattern.compile(regex);
                public boolean accept(File dir, String name) {
                    return pattern.matcher(new File(name).getName()).matches();
                }
            });
```

```
    }
    public static File[]
        local(String path, final String regex) { // Overloaded
        return local(new File(path), regex);
    }
    // A two-tuple for returning a pair of objects:
    public static class TreeInfo implements Iterable<File> {
        public List<File> files = new ArrayList<File>();
        public List<File> dirs = new ArrayList<File>();
        // The default iterable element is the file list:
        public Iterator<File> iterator() {
            return files.iterator();
        }
        void addAll(TreeInfo other) {
            files.addAll(other.files);
            dirs.addAll(other.dirs);
        }
        public String toString() {
            return "dirs: " + PPrint.pformat(dirs) +
                "\n\nfiles: " + PPrint.pformat(files);
        }
    }
    public static TreeInfo walk(String start, String regex) { // Begin recursion
        return recurseDirs(new File(start), regex);
    }
    public static TreeInfo walk(File start, String regex) { // Overloaded
        return recurseDirs(start, regex);
    }
    public static TreeInfo walk(File start) { // Everything
        return recurseDirs(start, ".*");
    }
    public static TreeInfo walk(String start) {
        return recurseDirs(new File(start), ".*");
    }
    static TreeInfo recurseDirs(File startDir, String regex){
        TreeInfo result = new TreeInfo();
        for(File item : startDir.listFiles()) {
            if(item.isDirectory()) {
                result.dirs.add(item);
                result.addAll(recurseDirs(item, regex));
            } else // Regular file
                if(item.getName().matches(regex))
                    result.files.add(item);
        }
        return result;
    }
    // Simple validation test:
    public static void main(String[] args) {
        if(args.length == 0)
            System.out.println(walk("."));
        else
            for(String arg : args)
                System.out.println(walk(arg));
    }
} ///:~
```

Die Methode `local()` ruft `listFiles()` auf, eine Variante der `File`-Methode `list()`, um eine `File`-Array zu erzeugen und implementiert mittels einer lokalen anonymen inneren Klasse das Interface *`FilenameFilter`*. Wenn Sie anstelle eines Arrays ein *`List`*-Objekt brauchen, können Sie das von `local()` zurückgegebene Array mit der `Arrays`-Methode `asList()` umwandeln.

[15] Die `walk()`-Methode wandelt den Namen des Startverzeichnisses in ein `File`-Objekt um und startet die rekursive Verarbeitung des Verzeichnisbaums durch Aufrufen der Methode `recurseDirs()`, die bei jedem Rekursionsschritt mehr Informationen erfaßt. Die `walk()`-Methode gibt ein „Tupel“ von Objekten zurück, um gewöhnliche Dateien von Verzeichnissen unterscheiden zu können, genauer ein *`List<File>`*-Objekt mit Dateien und weiteres *`List<File>`*-Objekt mit Verzeichnissen. Die Felder `files` und `dirs` sind absichtlich als `public` deklariert, da die Aufgabe der Hilfsklasse `TreeInfo` lediglich darin besteht, die `File`-Objekte der Dateien und Verzeichnisse zusammenzusuchen, *if you were just returning a List, you wouldn't make it private, so just because you are returning a pair of objects, it doesn't mean you need to make them private*. Beachten Sie, daß die Klasse `TreeInfo` das Interface *`Iterable`* implementiert und einen „Standarditerator“ über die ausgewählten Dateien (das Feld `files`) erzeugt. Die Verzeichnisse fragen Sie ab, in dem Sie nach `.dirs` dereferenzieren, siehe *DirectoryDemo.java* auf Seite 707.

[16] Die `TreeInfo`-Methode `toString()` stützt sich auf die Hilfsklasse `net.mindview.util.PPrint`, einen „Pretty Printer“, um die Lesbarkeit der Ausgabe zu verbessern. Die (von `Object`) ererbte Standardmethode `toString()` gibt alle Elemente eines Containers in einer einzigen Zeile aus. Dieser Ausgabemodus ist für Kollektionen mit vielen Elementen ungeeignet und eine alternative Formatierung wünschenswert. Die Hilfsklasse `PPrint` gibt jedes Element eingerückt in einer eigenen Zeile aus:

```
//: net/mindview/util/PPrint.java
// Pretty-printer for collections
package net.mindview.util;
import java.util.*;

public class PPrint {
    public static String pformat(Collection<?> c) {
        if(c.size() == 0) return ''[]'';
        StringBuilder result = new StringBuilder(''[''); 
        for(Object elem : c) {
            if(c.size() != 1)
                result.append(''\n '');
            result.append(elem);
        }
        if(c.size() != 1)
            result.append(''\n'');
        result.append('']'');
        return result.toString();
    }
    public static void pprint(Collection<?> c) {
        System.out.println(pformat(c));
    }
    public static void pprint(Object[] c) {
        System.out.println(pformat(Arrays.asList(c)));
    }
} ///:~
```

Die `pformat()`-Methode erzeugt aus einer Kollektion ein `String`-Objekt mit formatiertem Inhalt und wird von einer der beiden `pprint()`-Methoden aufgerufen. Beachten Sie, daß die Spezialfälle „leere Kollektion“ und „Kollektion enthält genau ein Element“ separat behandelt werden.

[17] Die Hilfsklasse `Directory` gehört zum Package `net.mindview.util`. Ein Anwendungsbeispiel für `Directory`:

```
//: io/DirectoryDemo.java
// Sample use of Directory utilities.
import java.io.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DirectoryDemo {
    public static void main(String[] args) {
        // All directories:
        PPrint.pprint(Directory.walk(".").dirs);
        // All files beginning with 'T'
        for(File file : Directory.local(".", "T.*"))
            print(file);
        print("----------------------");
        // All Java files beginning with 'T':
        for(File file : Directory.walk(".", "T.*\\.java"))
            print(file);
        print("=======================");
        // Class files containing "Z" or "z":
        for(File file : Directory.walk(".", ".*[Zz].*\\.class"))
            print(file);
    }
} /* Output: (Sample)
      [.\xfiles]
      .\TestEOF.class
      .\TestEOF.java
      .\TransferTo.class
      .\TransferTo.java
      ----------------------
      .\TestEOF.java
      .\TransferTo.java
      .\xfiles\ThawAlien.java
      =======================
      .\FreezeAlien.class
      .\GZIPcompress.class
      .\ZipCompress.class
   */// :~
```

Eventuell müssen Sie Ihre Kenntnisse über reguläre Ausdrücke auffrischen, um das zweite Argument der Methoden `local()` und `walk()` zu verstehen (siehe Abschnitt 14.6).

[18] Wir können noch einen Schritt weitergehen und eine Hilfsklasse entwickeln, die einen Verzeichnisbaum durchläuft *und* die ausgewählten Dateien mit Hilfe eines *`Strategy`*-Objektes verarbeitet (ein weiteres Beispiel für das *Strategy*-Entwurfsmuster):

```
//: net/mindview/util/ProcessFiles.java
package net.mindview.util;
import java.io.*;

public class ProcessFiles {
    public interface Strategy {
        void process(File file);
    }
    private Strategy strategy;
    private String ext;
    public ProcessFiles(Strategy strategy, String ext) {
```

```
            this.strategy = strategy;
            this.ext = ext;
        }
        public void start(String[] args) {
            try {
                if(args.length == 0)
                    processDirectoryTree(new File(''.''));
                else
                    for(String arg : args) {
                        File fileArg = new File(arg);
                        if(fileArg.isDirectory())
                            processDirectoryTree(fileArg);
                        else {
                            // Allow user to leave off extension:
                            if(!arg.endsWith(''.'' + ext))
                                arg += ''.'' + ext;
                            strategy.process(new File(arg).getCanonicalFile());
                        }
                    }
            } catch(IOException e) {
                throw new RuntimeException(e);
            }
        }
        public void processDirectoryTree(File root) throws IOException {
            for(File file : Directory.walk(
                root.getAbsolutePath(), ''.*\\.'' + ext))
                strategy.process(file.getCanonicalFile());
        }
        // Demonstration of how to use it:
        public static void main(String[] args) {
            new ProcessFiles(new ProcessFiles.Strategy() {
                    public void process(File file) {
                        System.out.println(file);
                    }
                }, ''java'').start(args);
        }
    } /* (Execute to see output) *////:~
```

Das Interface *Strategy* ist in der Klasse `ProcessFiles` verschachtelt, das heißt Sie müssen die Syntax `implements ProcessFiles.`*Strategy* verwenden, um das Interface zu implementieren, ~~which provides more context for the reader~~. Die Klasse `ProcessFiles` erledigt die gesamte Arbeit: Sie findet die Dateien mit einer bestimmten Endung (`ext`-Argument des Konstruktors) und übergibt jede passende Datei dem *Strategy*-Objekt (erstes Argument des Konstruktors).

[19] Wenn Sie das Programm ohne Argument aufrufen, nimmt `ProcessFiles` an, daß Sie alle Unterverzeichnisse des aktuellen Arbeitsverzeichnisses durchlaufen wollen. Sie können eine oder mehrere Dateien mit oder ohne Endung (das Programm fügt die Endung an, falls erforderlich) sowie eines oder mehrere Verzeichnisse übergeben.

[20] Die `main()`-Methode zeigt ein einfaches Beispiel für die Anwendung der Hilfsklasse `ProcessFiles`. Das Programm gibt die Namen von *.java* Dateien an, je nachdem ob und welche Kommandozeilenargumente Sie übergeben.

Übungsaufgabe 4: (2) Verwenden Sie die `Directory`-Methode `walk()`, um die Größen aller Dateien eines Verzeichnisbaumes zu addieren, deren Namen zu einem regulären Ausdruck passen. ■

Übungsaufgabe 5: (1) Ändern Sie *ProcessFiles.java* so, daß statt einer hartkodierten Dateiendung per Kommandozeile ein regulärer Ausdruck übergeben werden kann. ■

19.1.3 Abfragen und Ändern von Datei- und Verzeichniseigenschaften

[21] Die Klasse `File` ist mehr als nur Repräsentant einer existierenden Datei oder eines vorhandenen Verzeichnisses. Sie können `File`-Objekte verwenden, um ein neues Verzeichnis und sogar einen ganzen Verzeichnispfad anzulegen. Sie können Dateieigenschaften abfragen (zum Beispiel Größe, Datum der letzten Änderung oder Lese- und Schreibberechtigung), feststellen, ob das `File`-Objekt eine Datei oder ein Verzeichnis darstellt und Dateien löschen. Das folgende Beispiel führt einige dieser Methoden vor (Sie finden eine vollständige Beschreibung aller Methoden in der API-Dokumentation der Klasse `File`):

```
//: io/MakeDirectories.java
// Demonstrates the use of the File class to
// create directories and manipulate files.
// {Args: MakeDirectoriesTest}
import java.io.*;

public class MakeDirectories {
    private static void usage() {
        System.err.println(
            ''Usage:MakeDirectories path1 ...\n'' +
            ''Creates each path\n'' +
            ''Usage:MakeDirectories -d path1 ...\n'' +
            ''Deletes each path\n'' +
            ''Usage:MakeDirectories -r path1 path2\n'' +
            ''Renames from path1 to path2'');
        System.exit(1);
    }
    private static void fileData(File f) {
        System.out.println(
            ''Absolute path: '' + f.getAbsolutePath() +
            ''\n Can read: '' + f.canRead() +
            ''\n Can write: '' + f.canWrite() +
            ''\n getName: '' + f.getName() +
            ''\n getParent: '' + f.getParent() +
            ''\n getPath: '' + f.getPath() +
            ''\n length: '' + f.length() +
            ''\n lastModified: '' + f.lastModified());
        if(f.isFile())
            System.out.println(''It's a file'');
        else if(f.isDirectory())
            System.out.println(''It's a directory'');
    }
    public static void main(String[] args) {
        if(args.length < 1) usage();
        if(args[0].equals(''-r'')) {
            if(args.length != 3) usage();
            File
                old = new File(args[1]),
                rname = new File(args[2]);
            old.renameTo(rname);
            fileData(old);
            fileData(rname);
            return; // Exit main
```

```
            }
            int count = 0;
            boolean del = false;
            if(args[0].equals("-d")) {
                count++;
                del = true;
            }
            count--;
            while(++count < args.length) {
                File f = new File(args[count]);
                if(f.exists()) {
                    System.out.println(f + " exists");
                    if(del) {
                        System.out.println("deleting..." + f);
                        f.delete();
                    }
                }
                else { // Doesn't exist
                    if(!del) {
                        f.mkdirs();
                        System.out.println("created " + f);
                    }
                }
                fileData(f);
            }
        }
    } /* Output: (80% match)
          created MakeDirectoriesTest
          Absolute path: d:\aaa-TIJ4\code\io\MakeDirectoriesTest
          Can read: true
          Can write: true
          getName: MakeDirectoriesTest
          getParent: null
          getPath: MakeDirectoriesTest
          length: 0
          lastModified: 1101690308831
          It's a directory
       *///:~
```

Die `fileData()`-Methode ruft verschiedene `File`-Methoden auf, um Eigenschaften von Dateien und Verzeichnissen abzufragen.

[22] Die erste im Körper der `main()`-Methode aufgerufene `File`-Methode ist `renameTo()`. Die `renameTo()`-Methode gestattet eine Datei umzubenennen oder ans Ende eines anderen Pfades zu verschieben, der wiederum durch ein `File`-Objekt gegeben ist. Die `renameTo()`-Methode gestattet auch das Umbenennen von beliebigen Verzeichnissen.

[23] Beim Ausprobieren des obigen Programms werden Sie feststellen, daß der Verzeichnispfad beliebig komplex sein kann und sich die `mkdirs()`-Methode um alles kümmert.

Übungsaufgabe 6: (5) Finden Sie mit Hilfe der Klasse `ProcessFiles` alle *.java* Dateien in einem Verzeichnisbaum, die nach einem bestimmten Datum geändert wurden. ■

19.2 Ein- und Ausgabe

[24] Die Ein-/Ausgabebibliotheken von Programmiersprachen bauen häufig auf dem Konzept des abstrakten Datenstroms auf, der eine „Datenquelle“ beziehungsweise ein „Datenziel“ in Gestalt eines Objektes repräsentiert, welches Daten senden oder empfangen kann. Der Datenstrom verbirgt die Einzelheiten im Inneren des eigentlichen Ein-/Ausgabegerätes.

[25] Die Ein-/Ausgabeklassen der Java-Bibliothek sind zunächst nach Ein- beziehungsweise Ausgabe unterteilt (siehe Klassenhierarchie in der API-Dokumentation). Infolge der Ableitung besitzt jede von `java.io.InputStream` oder `java.io.Reader` abgeleitete Klasse einige primitive `read()`-Methoden, um einzelne `byte`-Werte beziehungsweise `byte`-Arrays lesen zu können. Analog besitzt jede von `java.io.OutputStream` oder `java.io.Writer` abgeleitete Klasse einige primitive `write()`-Methoden, um einzelne `byte`-Werte beziehungsweise `byte`-Arrays schreiben zu können. In der Regel rufen Sie diese Methoden allerdings nicht selbst auf, sondern verwenden hierfür Klassen mit komfortableren Schnittstellen. Sie erzeugen Ihre Datenstromobjekte daher nur sehr selten aus einer einzelnen Klasse, sondern verwenden mehrere ineinander verschachtelte Objekte, um die benötigte Funktionalität zu erhalten (das *Decorator*-Entwurfsmuster beschreibt diesen Ansatz, wie Sie in diesem Abschnitt lernen werden). Die Tatsache, daß Sie mehr als ein Objekt brauchen, um einen einzelnen Datenstrom zu erhalten, ist der primäre Grund für die Unübersichtlichkeit der Ein-/Ausgabebibliothek von Java.

[26] Es hilft, die Klassen hinsichtlich ihrer Funktionalität in Kategorien einzuteilen. Für Java 1.0 legten die Designer der Bibliothek fest, daß alle Klassen, die etwas mit Eingabe zu tun haben, von `InputStream` und alle Klassen, die etwas mit Ausgabe zu tun haben, von `OutputStream` abgeleitet werden.

[27] Ich versuche im folgenden, wie auch andernorts in diesem Buch, einen Überblick über die Klassen zu liefern und setze voraus, daß Sie die Einzelheiten, etwa die Methoden einer Klasse, in der API-Dokumentation nachlesen.

19.2.1 Von InputStream abgeleitete Klassen

[28] Die Hierarchie unterhalb von `InputStream` repräsentiert Klassen, die Eingaben von verschiedenen Quellen erwarten. Diese Datenquellen sind:

- Ein Array von Bytes.
- Ein `String`-Objekt.
- Eine Datei.
- Eine „Pipe“. (Verhält sich wie ein Rohr: Sie stecken am einen Ende etwas hinein und es kommt an anderen Ende wieder heraus.)
- Eine Abfolge von Datenströmen. Sie können mehrere Datenströme zu einem einzigen zusammenfassen.
- Andere Quellen, etwa Internetverbindungen. (Siehe „Thinking in Enterprise Java“, unter *www.mindview.net*).

[29] Zu jeder dieser Datenquellen gehört eine Unterklasse von `InputStream`. Darüber hinaus ist auch `FilterInputStream` von `InputStream` abgeleitet. `FilterInputStream` stellt den „Sockel“ für eine Implementierung des *Decorator*-Entwurfsmusters dar, ermöglicht also to attach attributes oder nützliche Methoden zum Eingabedatenstrom.

Klasse und Funktion	Konstruktorargumente	Verwendungszweck
`ByteArrayInputStream` verwendet einen Puffer im Arbeitsspeicher als Eingabedatenstrom (`InputStream`).	Der Puffer aus dem die Bytes gelesen werden sollen.	Als Datenquelle. In dem Sie es mit einem `FilterInputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`StringBufferInputStream` (*deprecated*) wandelt ein `String`- in ein `InputStream`-Objekt um.	Ein `String`-Objekt. Die unterliegende Implementierung verwendet aber ein `StringBuffer`-Objekt.	Als Datenquelle. In dem Sie es mit einem `FilterInputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`FileInputStream` liest Informationen aus einer Datei ein.	Ein `String`-Objekt, welches den Dateinamen enthält, beziehungsweise ein `File`- oder `FileDescriptor`-Objekt, welches die Datei repräsentiert.	Als Datenquelle. In dem Sie es mit einem `FilterInputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`PipedInputStream` liefert die Daten, die an das komplementäre `PipedOutputStream`-Objekt übergeben werden. Implementiert eine „Pipe“.	Ein `PipedOutputStream`-Objekt.	Als Datenquelle bei Threads. In dem Sie es mit einem `FilterInputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`SequenceInputStream` wandelt zwei oder mehr `InputStream`-Objekte in ein einziges `InputStream` um.	Zwei `InputStream`-Objekt oder ein *`Enumeration`*-Objekt als Container für mehrere `InputStream`-Objekte.	Als Datenquelle. In dem Sie es mit einem `FilterInputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`FilterInputStream` ist eine abstrakte Klasse. ~~An interface~~ für Dekorationen, die nützliche Funktionalität für die übrigen von `InputStream` abgeleiteten Klassen implementieren.	Siehe Tab. 19.3.	Siehe Tab. 19.3.

Tabelle 19.1: Von InputStream abgeleitete Klassen.

19.2.2 Von OutputStream abgeleitete Klassen

[30] Die Klassen in dieser Kategorie bestimmen das Ziel Ihrer Ausgabe: ein Array von Bytes (kein `String`-Objekt; Sie können das Array aber umwandeln), eine Datei oder eine „Pipe“.

[31] `FilterOutputStream` stellt den „Sockel“ für eine Implementierung des *Decorator*-Entwurfsmusters dar, ermöglicht also ~~to attach attributes~~ oder nützliche Methoden zum Ausgabedatenstrom.

19.3 ~~Adding attributes and useful interfaces~~

[32] Das Entwurfsmuster *Decorator* wurde in Kapitel 16 eingeführt (siehe Seite 559). Die Ein-/Ausgabebibliothek von Java viele verschiedene Kombinationen von Eigenschaften und Fähigkeiten und dies ist die Begründung für die Verwendung des *Decorator*-Entwurfsmusters. (Es ist nicht klar, ob diese Design-Entscheidung tatsächlich gut war, insbesondere im Hinblick auf die Ein-/Ausgabebibliotheken anderer Programmiersprachen. Aber dies ist der Grund für diese Entscheidung.) Der Grund für die Existenz der Filterklassen in der Ein-/Ausgabebibliothek von Java besteht darin, daß sie die „Sockelklassen“ für Dekoratoren sind. Dekorator und dekoriertes Objekt müssen dieselbe Schnittstelle besitzen, wobei die Schnittstelle des Dekorators größer sein darf (trifft auf einige Filterklassen zu).

[33] Das *Decorator*-Entwurfsmuster hat allerdings auch einen Nachteil. Dekoratoren bewirken zwar eine beachtliche Flexibilität, da Sie mühelos mix and match attributes können, verkomplizieren allerdings auch den Quelltext. Die Ein-/Ausgabebibliothek von Java ist schwierig zu handhaben, da in der Regel Objekte mehrerer Klassen geschachtelt werden müssen (der „Basistyp“ und seine Dekoratoren), um das erforderliche Ein-/Ausgabeobjekt zu bekommen.

[34] Die Klassen mit Dekoratorschnittstelle, um ein bestimmtes `InputStream`- oder `OutputStream`-Objekt steuern zu können, heißen `FilterInputStream` und `FilterOutputStream` (keine sehr intuitiven Namen). `FilterInputStream` und `FilterOutputStream` sind von den Basisklassen `InputStream` beziehungsweise `OutputStream` der Ein-/Ausgabebibliothek abgeleitet (eine wesentliche Voraussetzung für Dekoratoren lautet, daß sie und die dekorierten Objekte gemeinesame Schnittstellen haben müssen).

19.3.1 Lesen aus einem InputStream per FilterInputStream

[35] Die Unterklassen von `FilterInputStream` verrichten zwei verschiedene Aufgaben. Die Klasse `DataInputStream` gestattet, Werte primitiven Typs und Unicodezeichenketten einzulesen. (Die Bezeichner der entsprechenden Methoden beginnen mit „read“ und enthalten den Datentyp, zum Beispiel `readByte()` oder `readFloat()`.) Zusammen mit `DataOutputStream` ermöglicht `DataInputStream` Werte primitiven Typs über einen Datenstrom von einem Ort an einen anderen zu verschieben. Tabelle 19.3 gibt diese „Orte“ an.

[36] Die übrigen Unterklassen von `FilterInputStream` betreffen das Verhalten eines `InputStream`-Objektes, das heißt ob es einen Puffer verwendet oder nicht, ob es die eingelesenen Zeilen speichert (so daß Sie Zeilennummern abfragen oder vergeben können) und ob Sie ein einzelnes Zeichen [!zurückdrängen (to push back)] können. Die beiden letzten Klassen (`PushbackInputStream` und

Klasse und Funktion	Konstruktorargumente	Verwendungszweck
`ByteArrayOutputStream` legt im Arbeitsspeicher einen Puffer an. Alle dem Datenstrom übergebenen Daten werden in diesem Puffer gespeichert.	Anfangsgröße des Puffers (optional).	Angabe des Datenziels: In dem Sie es mit einem `FilterOutputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`FileOutputStream` schreibt Informationen in eine Datei.	Ein `String`-Objekt. Die unterliegende Implementierung verwendet aber ein `StringBuffer`-Objekt.	Angabe des Datenziels: In dem Sie es mit einem `FilterOutputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`PipedOutputStream`. Jede diesem Objekt übergebene Information wird automatisch dem zugehörigen `PipedInputStream`-Objekt übergeben. Implementiert eine „Pipe“.	Ein `PipedInputStream`-Objekt.	Als Datenziel bei Threads. In dem Sie es mit einem `FilterOutputStream`-Objekt verknüpfen, erhalten Sie eine nützliche Schnittstelle.
`FilterOutputStream` ist eine abstrakte Klasse. An interface für Dekorationen, die nützliche Funktionalität für die übrigen von `OutputStream` abgeleiteten Klassen implementieren.	Siehe Tab. 19.4.	Siehe Tab. 19.4.

Tabelle 19.2: Von `OutputStream` abgeleitete Klassen.

[//.//]) sind vermutlich dazu gedacht, einen Compiler zu schreiben (und wurden möglicherweise hinzugefügt, um einen Java-Compiler in Java entwickeln zu können), das heißt Sie werden sie im Rahmen Ihrer Programmierarbeit wahrscheinlich verwenden.

[37] Da Sie, unabhängig vom Eingabegerät, fast immer einen Eingabepuffer verwenden, wäre es sinnvoller gewesen, wenn die Ein-/Ausgabebibliothek die ungepufferte Eingabe als Sonderfall behandeln würde, nicht aber die gepufferte.

Klasse und Funktion	Konstruktorargumente	Verwendungszweck
`DataInputStream` wird zusammen mit `DataOutputStream` verwendet, um Werte primitiven Typs aus einem Datenstrom zu lesen.	Ein `InputStream`-Objekt.	Stellt eine komplette Schnittstelle zum Lesen von Werten beliebigen primitiven Typs zur Verfügung.
`BufferedInputStream` wird verwendet, um bei Anfrage nach mehr Daten die physikalischen Leseoperationen zu vermeiden. Sie „verwenden einen Puffer“.	Ein `InputStream`-Objekt mit optionaler Puffergröße.	Liefert noch keine Schnittstelle, sondern fügt lediglich die Pufferung hinzu. Verknüpfen Sie die Kombination zusätzlich mit einem Objekt, welches eine Schnittstelle besitzt.
`LineNumberInputStream` pflegt die Zeilennummern im Eingabedatenstrom. Sie haben die Methoden `getLineNumber()` und `setLineNumber()` zur Verfügung.	Ein `InputStream`-Objekt.	Fügt lediglich die Zeilennummierung hinzu. Verknüpfen Sie die Kombination zusätzlich mit einem Objekt, welches eine Schnittstelle besitzt.
`PushbackInputStream` verfügt über einen ~~push/back~~ Puffer für ein Byte, damit Sie das zuletzt gelesene Byte ~~push/back~~ können.	Ein `InputStream`-Objekt.	Tritt in der Regel im Scanner eines Compilers auf. Sie werden diese Klasse wahrscheinlich nicht brauchen.

Tabelle 19.3: Von FilterInputStream abgeleitete Klassen.

Klasse und Funktion	Konstruktorargumente	Verwendungszweck
`DataOutputStream` wird zusammen mit `DataInputStream` verwendet, um Werte primitiven Typs in einen Datenstrom zu schreiben.	Ein `OutputStream`-Objekt.	Stellt eine komplette Schnittstelle zum Schreiben von Werten beliebigen primitiven Typs zur Verfügung.
`PrintStream` wird verwendet, um formatierte Ausgabe zu erzeugen. Während sich `DataOutputStream` um die Speicherung der Daten kümmert, dient `PrintStream` der Visualisierung.	Ein `OutputStream`-Objekt mit optionalem booleschen Parameter, der anzeigt, ob der Puffer bei Zeilenumbruch geleert werden soll.	Verwenden Sie diese Klasse als äußerste Schachtelungsebene bei einem `OutputStream`-Objekt. Wahrscheinlich werden Sie diese Kombination häufig anwenden.
`BufferedOutputStream` wird verwendet, um bei mehreren Schreiboperationen die physikalischen Schreiboperationen zu vermeiden. Sie „verwenden einen Puffer“. Sie können die `flush()`-Methode aufrufen, um den Puffer zu leeren.	Ein `OutputStream`-Objekt mit optionaler Puffergröße.	Liefert noch keine Schnittstelle, sondern fügt lediglich die Pufferung hinzu. Verknüpfen Sie die Kombination zusätzlich mit einem Objekt, welches eine Schnittstelle besitzt.

Tabelle 19.4: Von FilterOutputStream abgeleitete Klassen.

19.3.2 Schreiben an einen OutputStream per FilterOutputStream

[38] `DataOutputStream` ist das Gegenstück zu `DataInputStream` und formatiert jeden Wert primitiven Typs sowie `String`-Objekte, so daß sie von einem `DataInputStream`-Objekt unter einer beliebigen Plattform wieder gelesen werden können. Die Bezeichner der entsprechenden Methoden beginnen mit „write“ und enthalten den Datentyp, zum Beispiel `writeByte()` oder `writeFloat()`.

[39] Die Klasse `PrintStream` war ursprünglich dazu gedacht, Werte primitiven Typs und `String`-Objekte in sichtbarer Formatierung ausgeben zu können. Die Funktionalität von `PrintStream` unterscheidet sich von `DataOutputStream`, deren Aufgabe darin besteht, Datenelemte so an den Datenstrom zu übergeben, daß sie von einem `DataInputStream`-Objekt portabel wieder eingelesen werden können.

[40] Die beiden wichtigsten Methoden der Klasse `PrintStream` sind `print()` und `println()`, welche überladen sind, um die verschiedenen Typen drucken zu können. Der Unterschied zwischen `print()` und `println()` besteht darin, daß die letztere Methode nach ihrer Ausgabe einen Zeilenumbruch hinzufügt.

[41] Die Klasse `PrintStream` ist problematisch, weil sie alle Ausnahmen vom Typ `IOException` abfängt. (Sie müssen den Fehlerstatus per `checkError()` explizit testen. Die Methode gibt `true` zurück, wenn ein Fehler aufgetreten ist.) Außerdem funktioniert die Internationalisierung bei `PrintStream` nicht richtig und Zeilenumbrüche werden nicht plattformübergreifend behandelt. Diese Probleme sind bei der Klasse `PrintWriter` gelöst (siehe **??**).

[42] Die Klasse `BufferedOutputStream` modifiziert das Verhalten des Datenstroms durch Zwischenschalten eines Puffers, so daß Sie nicht mit jeder Schreiboperation in den Datenstrom einen physikalischen Schreibvorgang auslösen. Ausgabepufferung ist generell sinnvoll.

19.4 Die abstrakten Klassen Reader und Writer

[43] Java 1.1 hat einige wesentliche Veränderungen an der Grundstruktur der Ein-/Ausgabebibliothek vollzogen. Auf den ersten Blick liegt der Gedanke nahe, daß die abstrakten Klassen `Reader` und `Writer` die ebenfalls abstrakten Klassen `InputStream` und `OutputStream` ersetzen sollen, aber dies trifft nicht zu. Obwohl Teile der ursprünglichen byteorientierten Bibliothek stellenweise als *deprecated* deklariert ist (die Verwendung der entsprechenden Klassen beziehungsweise Methoden ruft eine Warnmeldung des Compilers hervor), leisten die Hierarchien unter den Basisklassen `InputStream` und `OutputStream` noch immer gute Dienste bei byteorientierten Ein-/Ausgabeoperationen. Die abstrakten Klassen `Reader` und `Writer` gestatten dagegen unicodebasierte, zeichenorientierte Ein-/Ausgabeoperationen. Außerdem

- hat Java 1.1 die Hierarchien unter den Basisklassen `InputStream` und `OutputStream` um einige neue Klassen erweitert. Daher werden beide Hierarchien offensichtlich nicht ersetzt.
- gibt es Situationen, in denen Sie Klassen aus einer der byteorientierten Hierarchien mit Klassen aus einer der zeichenorientierten Hierarchien kombinieren müssen. Zu diesem Zweck existieren die folgenden beiden „Adapterklassen“:
  - `InputStreamReader` wandelt ein `InputStream`-Objekt in ein `Reader`-Objekt um.
  - `OutputStreamWriter` wandelt ein `OutputStream`-Objekt in ein `Writer`-Objekt um.

[44] Der wichtigste Grund für die Existenz der Klassen `Reader` und `Writer` ist aber die Internationalisierung. Die alten Hierarchien unter den Basisklassen `InputStream` und `OutputStream` unterstützen nur 8-Bit-Byteströme, nicht aber 16-Bit Unicodezeichen. Da die Internationalisierung auf Unicode

Byteorientierte Klasse (Java 1.0)	Entspr. zeichenorientierte Klasse (Java 1.1)
`InputStream`	`Reader`, Adapter: `InputStreamReader`
`OutputStream`	`Writer`, Adapter: `OutputStreamWriter`
`FileInputStream`	`FileReader`
`FileOutputStream`	`FileWriter`
`StringBufferInputStream` (*deprecated*)	`StringReader`
Unter Java 1.0 ist keine Klasse vorhanden.	`StringWriter`
`ByteArrayInputStream`	`CharArrayReader`
`ByteArrayOutputStream`	`CharArrayWriter`
`PipedInputStream`	`PipedReader`
`PipedOutputStream`	`PipedWriter`

Tabelle 19.5: ...

beruht (und der native Java-Typ `char` für 16-Bit Unicodezeichen konzipiert ist), wurden die neuen Klassenhierarchien unter den Basisklassen `Reader` und `Writer` in die Bibliothek aufgenommen, um auch bei allen Ein-/Ausgabeoperationen Unicode zu unterstützen. Außerdem gestatten die neuen Bibliotheksteile schnellere Operationen als die alten.

19.4.1 Datenquellen und Datenziele

[45] Zu fast jeder der ursprünglichen byteorientierten Datenstromklassen gehört eine entsprechende Klasse aus den Hierarchien unter den Basisklassen `Reader` beziehungsweise `Writer`, um den nativen Umgang mit Unicode zu ermöglichen. Dennoch gibt es Situationen, in denen Sie eine byteorientierte Klasse verwenden müssen, insbesondere sind zum Beispiel die Klassen im Package `java.util.zip` byte- und nicht zeichenorientiert. Die vernünftigste Vorgehensweise besteht darin, stets eine zeichenorientierte Klasse zu *versuchen*. Sie erkennen die Situationen in denen Sie byteorientierte Klassen verwenden müssen daran, daß sich Ihr Programm nicht übersetzen läßt.

[46] Tabelle 19.5 stellt die alten byteorientierten und die neuen zeichenorientierten Datenquellen beziehungsweise -ziele (das heißt woher die Daten physikalisch kommen beziehungsweise wohin sie gesendet werden) einander gegenüber. In der Regel sind die Schnittstellen der gepaarten Klassen in den beiden Hierarchien ähnlich oder sogar identisch.

19.4.2 Anpassen des Verhaltens eines Datenstroms

[47] Die Klassen aus den Hierarchien unter den byteorientierten Basisklassen `InputStream` beziehungsweise `OutputStream` können durch Dekoratoren vom Typ `FilterInputStream` beziehungsweise `FilterOutputStream` an den Typ des jeweiligen Datenstroms angepaßt werden. Die Hierarchien unter den zeichenorientierten Basisklassen `Reader` und `Writer` folgen diesem Ansatz, allerdings nicht exakt.

[48] Die Zusammenhänge in Tabelle 19.6 sind weniger stark ausgeprägt als in Tabelle 19.5. Die Abweichungen ergeben sich durch die Anordnung der Klassen in ihren Hierarchien. `BufferedOutputStream` ist zwar von `FilterOutputStream` abgeleitet, aber `BufferedWriter` nicht von `FilterWriter`. (Nebenbei bemerkt ist `FilterWriter` zwar eine abstrakte Klasse, aber es gibt keine abgeleitete Klasse. Die Klasse scheint ein Platzhalter zu sein, damit Sie sich nicht fragen, warum die entsprechende Position in der Hierarchie nicht besetzt ist.) Die Schnittstellen der Klassen are quite a close match dennoch.

[49] Eine Richtung ist klar: Wenn Sie eine `readLine()`-Methode brauchen, sollen Sie nicht `DataInputStream` (*deprecated*-Warnung zur Übersetzungszeit) verwenden, sondern `BufferedReader`. Other than this, DataInputStream is still a "preferred" member of the I/O library.

[50] Der Übergang zu einem `PrintWriter` wird dadurch erleichtert, daß die Klasse `PrintWriter` sowohl Konstruktoren für `OutputStream`- als auch für `Writer`-Objekt hat. Die Formatierungsmethoden der Schnittstelle von `PrintWriter` sind nahezu dieselben, wie bei `PrintStream`.

[51] In der Version 5 der Java Standard Edition (SE 5) wurden neue `PrintWriter`-Konstruktoren hinzugefügt, um das Anlegen von Ausgabedateien zu erleichtern, siehe Unterunterabschnitt 19.6.4.1.

[52] Einer der `PrintWriter`-Konstruktoren gestattet sogar die Leerung des Puffers automatisch durchzuführen. Die Leerung geschieht nach jedem `println()`, wenn das Objekt mit dem entsprechenden Konstruktoraufruf erzeugt wurde.

19.4.3 Unveränderte Klassen

[53] Einige Klassen wurden zwischen Java 1.0 und Java 1.1 nicht geändert:

- `DataOutputStream`
- `File`
- `RandomAccessFile`
- `SequenceInputStream`

`DataOutputStream` wird unverändert verwendet, so for storing and retrieving data in a transportable format, you use the InputStream and OutputStream hierarchies.

19.5 Eine Klasse für sich: RandomAccessFile

[54] Die Klasse `java.io.RandomAccessFile` ist für Dateien gedacht, die Datensätze bekannter Größe enthalten. Die `seek()`-Methode dient der Positionierung des Dateizeigers, bevor ein Datensatz

Byteorientierte Klasse (Java 1.0)	Entspr. zeichenorientierte Klasse (Java 1.1)
`FilterInputStream`	`FilterReader`
`FilterOutputStream`	`FilterWriter` (abstrakte Klasse ohne abgeleitete Klassen)
`BufferedInputStream`	`BufferedReader` (hat ebenfalls eine `readLine()`-Methode)
`BufferedOutputStream`	`BufferedWriter`
`DataInputStream`	Verwenden Sie `DataInputStream`, except when you need to use readLine(), when you should use a BufferedReader.
`PrintStream`	`PrintWriter`
`LineNumberInputStream` (*deprecated*)	`LineNumberReader`
`StreamTokenizer`	`StreamTokenizer` (Verwenden Sie den Konstruktor mit `Reader`-Argument.)
`StreamTokenizer`	`PushbackReader`

Tabelle 19.6: ...

gelesen oder geändert wird. Die Datensätze müssen keine einheitliche Größe haben. Es genügt, die Größe der einzelnen Datensätze bestimmen zu können, um ihre Position in der Datei zu ermitteln.

[55] Es fällt auf den ersten Blick hin schwer, sich vorzustellen, daß die Klasse `RandomAccessFile` *nicht* zur Hierarchie unter den Basisklassen `InputStream` beziehungsweise `OutputStream` gehört. Aber die einzige Gemeinsamkeit zwischen `RandomAccessFile` und `InputStream`/`OutputStream` besteht darin, daß auch `RandomAccessFile` die Interfaces *`java.io.DataInput`* und *`java.io.DataOutput`* implementiert (auch die Klassen `java.io.DataInputStream` und `java.io.DataOutputStream` implementieren *`DataInput`* beziehungsweise *`DataOutput`*). `RandomAccessFile` nutzt nicht einmal die Funktionalität der bereits vorhandenen Klassen `InputStream` und `OutputStream`, sondern ist eine völlig separate, von Grund auf neu entwickelte Klasse mit eigenen (größtenteils nativen) Methoden. Das ist so, weil sich die Klasse `RandomAccessFile` durch ihr Verhalten wesentlich von den übrigen Klassen der Ein-/Ausgabebibliothek unterscheidet: der Dateizeiger kann vor- und rückwärts bewegt werden. Auf jeden Fall steht `RandomAccessFile` alleine da, nur von `Object` abgeleitet.

[56] `RandomAccessFile` funktioniert im wesentlichen wie eine Kombination der Klassen `DataInputStream` und `DataOutputStream`, zusammen mit den Methoden `getFilePointer()` (gibt die Position des Dateizeigers zurück), `seek()` (setzt den Dateizeiger auf eine neue Position in der Datei) und `length()` (gibt die Größe der Datei zurück). Die Konstruktoren der Klasse `RandomAccessFile` erwarten außerdem ein zweites Argument, nämlich die Angabe, ob die Datei nur zum Lesen („r“) oder zum Lesen und Schreiben („rw“) geöffnet werden soll (wie `fopen()` bei C). Das Öffnen von Dateien zum Schreiben alleine wird nicht unterstützt, so daß `RandomAccessFile` als von `DataInputStream` abgeleitete Klasse durchgehen könnte.

[57] Die drei Suchmethoden `getFilePointer()`, `seek()` und `length()` existieren im `java.io`-Package nur in der Klasse `RandomAccessFile`. Die Klasse `BufferedInputStream` gestattet lediglich, per `mark()` eine Position zu kennzeichnen (die Position wird in einem Feld gespeichert) und mittels `reset()` später wieder dorthin zurückzukehren. Diese Positionierungsfunktionalität ist eingeschränkt und daher nur selten nützlich.

[58] Die Funktionalität der Klasse `RandomAccessFile` wird größtenteils, wenn nicht gar vollständig von den [!!!!(memory-mapped files)!] aus dem `java.nio`-Package verdrängt, die seit Version 1.4 des Java Development Kits vorhanden sind (siehe Unterabschnitt 19.10.6).

19.6 Typische Anwendungsbeispiele für die Ein-/Ausgabeströme

[59] Obwohl Sie die Klassen der Ein-/Ausgabebibliothek von Java auf vielfältige Weise miteinander kombinieren können, treten in der Praxis nur wenige Kombinationen auf. Sie können die folgenden Beispiele als Referenz typischer Anwendungsfälle verwenden.

[60] In den folgenden Beispielen werden Ausnahmen der Einfachheit halber nicht behandelt, sondern über die Konsole ausgegeben. Diese Vorgehensweise ist nur bei kleinen Beispielen, Hilfsklassen und Hilfsmethoden angebracht. Ziehen Sie bei Ihren eigenen Anwendungen und Programmen auch anspruchsvollere Möglichkeiten zur Fehlerbehandlung in Betracht.

19.6.1 Gepuffertes Einlesen einer Datei

[61] Wenn Sie eine Datei zum zeichenorientierten Einlesen öffnen möchten, verwenden Sie ein `java.io.FileReader`-Objekt und rufen den `FileReader`-Konstruktor mit einem `String`- oder `File`-Argument auf, welches den Dateinamen beziehungsweise die Datei repräsentiert. Aus Geschwindigkeitsgründen ist es sinnvoll, die Eingabe zu puffern. Sie übergeben daher die `FileReader`-Referenz

einem Konstruktor der Klasse `java.io.BufferedReader`. Die Schnittstelle des `BufferedReader`-Objektes enthält eine `readLine()`-Methode. Daher ist das `BufferedReader`-Objekt das äußerste Objekt dieser Kombination. Gibt die `readLine()`-Methode `null` zurück, so ist das Dateiende erreicht:

```
//: io/BufferedInputFile.java
import java.io.*;

public class BufferedInputFile {
    // Throw exceptions to console:
    public static String read(String filename) throws IOException {
        // Reading input by lines:
        BufferedReader in = new BufferedReader(new FileReader(filename));
        String s;
        StringBuilder sb = new StringBuilder();
        while((s = in.readLine())!= null)
            sb.append(s + ''\n'');
        in.close();
        return sb.toString();
    }
    public static void main(String[] args) throws IOException {
        System.out.print(read(''BufferedInputFile.java''));
    }
} /* (Execute to see output) *////:~
```

Das von `sb` referenzierte `StringBuilder`-Objekt erfaßt den gesamten Dateiinhalt (einschließlich der Zeilenumbrüche, die erneut angefügt werden müssen, da sie von `readLine()` entfernt werden). Zuletzt schließt `close()` die Datei[1].

Übungsaufgabe 7: (2) Öffnen Sie eine Textdatei und lesen Sie den Inhalt Zeile für Zeile ein. Speichern Sie jede Zeile als `String`-Objekt und übergeben Sie dieses anschließend einem `LinkedList`-Objekt. Geben Sie den Inhalt des `LinkedList`-Objektes in umgekehrter Reihenfolge wieder aus. ■

Übungsaufgabe 8: (1) Ändern Sie Übungsaufgabe 7 so, daß Sie den Namen der eingelesenen Datei als Kommandozeilenargument übergeben können. ■

Übungsaufgabe 9: (1) Ändern Sie Übungsaufgabe 8 so, daß alle eingelesenen Zeilen vor der Übergabe an das `LinkedList`-Objekt in Großbuchstaben umgewandelt werden und geben Sie das Ergebnis über `System.out` aus. ■

Übungsaufgabe 10: (2) Ändern Sie Übungsaufgabe 8 so, daß Sie einige Suchwörter als Kommandozeilenargumente übergeben können. Geben Sie alle Zeilen aus, die eines der Suchwörter enthalten. ■

Übungsaufgabe 11: (2) Die Klasse `GreenhouseController` aus Beispiel *innerclasses/GreenhouseController.java* enthält eine Reihe hartkodierter Ereignisse. Ändern Sie das Programm, so daß es die Ereignisse und relativen Zeiten aus einer Textdatei einliest. Schwierigkeitsgrad 8: Verwenden Sie eine Fabrikmethode, um die Ereignisse zu erzeugen, siehe „Thinking in Patterns“ auf *www.mindview.net*. ■

[1] Das ursprüngliche Design der Ein-/Ausgabebibliothek von Java sah vor, daß die `close()`-Methode während der Verarbeitung der `finalize()`-Methode aufgerufen wird (siehe *FileInputStream.java* und *FileOutputStream.java*). Da allerdings, wie andernorts in diesem Buch diskutiert, die Finalisierung nicht auf die von den Java-Designern vorgesehene Weise funktionierte (das heißt, unregelmäßig gebrochen war), bleibt nur das explizite Aufrufen der `close()`-Methode als sichere Vorgehensweise, um eine Datei zu schließen.

19.6.2 Einlesen einer Datei aus dem Arbeitsspeicher

[62] Im folgenden Beispiel wird das von der `BufferedInputFile`-Methode `read()` (siehe voriges Beispiel) zurückgegebene `String`-Objekt verwendet, um ein `StringReader`-Objekt zu erzeugen. Anschließend liest die `StringReader`-Methode `read()` die Eingabe zeichenweise und gibt Zeichen um Zeichen über die Konsole aus:

```
//: io/MemoryInput.java
import java.io.*;

public class MemoryInput {
    public static void main(String[] args) throws IOException {
        StringReader in = new StringReader(
            BufferedInputFile.read("MemoryInput.java"));
        int c;
        while((c = in.read()) != -1)
            System.out.print((char)c);
    }
} /* (Execute to see output) *////:~
```

Beachten Sie, daß die `read()`-Methode der Klasse `StringReader` das nächste Zeichen stets als `int`-Wert zurückgibt, der in `char` umgewandelt werden muß, um korrekt angezeigt werden zu können.

19.6.3 Einlesen formatierter Daten aus dem Arbeitsspeicher

[63] Wenn Sie „formatierte“ Daten einlesen möchten, verwenden Sie ein `java.io.DataInputStream`-Objekt. Die Klasse `DataInputStream` ist byteorientiert. Damit sind Sie an die Klassen aus der Hierarchie unter der Basisklasse `InputStream` gebunden (im Gegensatz zur Hierarchie unter `Reader`). `InputStream`-Klassen können selbstverständlich jede beliebige Eingabe byteorientiert einlesen (zum Beispiel Dateien), im folgenden Beispiel verwenden Sie ein per `getBytes()` in ein `byte`-Array umgewandeltes `String`-Objekt:

```
//: io/FormattedMemoryInput.java
import java.io.*;

public class FormattedMemoryInput {
    public static void main(String[] args) throws IOException {
        try {
            DataInputStream in = new DataInputStream(
                new ByteArrayInputStream(
                    BufferedInputFile.read(
                        "FormattedMemoryInput.java").getBytes()));
            while(true)
                System.out.print((char)in.readByte());
        } catch(EOFException e) {
            System.err.println("End of stream");
        }
    }
} /* (Execute to see output) *////:~
```

Die Konstruktoren der Klasse `java.io.ByteArrayInputStream` erwarten ein `byte`-Array. Wir verwenden die `String`-Methode `getBytes()`, um das von der `BufferedInputFile`-Methode `read()` zurückgegebene `String`-Objekt entsprechend umzuwandeln. Das resultierende `ByteArrayInputStream`-Objekt ist ein passendes `InputStream`-Argument für den Konstruktor der Klasse `DataInputStream`.

[64] Wenn Sie die Zeichen aus einem `DataInputStream`-Objekt Byte für Byte auslesen, ist jedes Byte ein gültiger (*legitimate*) Rückgabewert, das heißt der Rückgabewert kann *nicht* verwendet werden, um das Dateiende festzustellen. Stattdessen, gibt die `available()`-Methode an, wieviele Zeichen noch vorhanden sind. Das folgende Beispiel zeigt, wie sich eine Datei Byte für Byte einlesen läßt:

```
//: io/TestEOF.java
// Testing for end of file while reading a byte at a time.
import java.io.*;

public class TestEOF {
    public static void main(String[] args) throws IOException {
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("TestEOF.java")));
        while(in.available() != 0)
            System.out.print((char)in.readByte());
    }
} /* (Execute to see output) *////:~
```

Beachten Sie, daß die Funktionsweise der `available()`-Methode vom Typ des eingelesenen Mediums abhängt. Die Methode gibt buchstäblich die Anzahl Bytes zurück, die *ohne Blockierung* gelesen werden können. Bei einer Datei bedeutet dies, „die ganze Datei“. Bei einem anderen Medium kann die Interpretation abweichen. Verwenden Sie die `available()`-Methode daher mit Sorgfalt.

[65] Sie können das Dateiende auch mittels Abfangen einer Ausnahme bestimmen (siehe *io/FormattedMemoryInput.java*). Die Verwendung von Ausnahmen zur Steuerung des Programmablaufs gilt aber als Mißbrauch des Mechanismus' zur Ausnahmebehandlung.

19.6.4 Schreiben in eine Ausgabedatei

[66] Wenn Sie Daten in eine Datei schreiben möchten, verwenden Sie ein `java.io.FileWriter`-Objekt. Es ist in der Regel sinnvoll, die Ausgabe durch Verpacken in ein `java.io.BufferedWriter`-Objekt zu puffern. (Entfernen Sie probehalber einmal die Pufferung, um die Auswirkung auf die Performanz zu beobachten. Pufferung bewirkt im allgemeinen eine drastische Geschwindigkeitszunahme bei Ein-/Ausgabeoperationen.) Im folgenden Beispiel wird das `FileWriter`-Objekt mit einem `PrintWriter`-Objekt dekoriert, um die Ausgabe formatieren zu können. Die auf diese Weise erzeugte Datei kann als gewöhnliche Textdatei gelesen werden:

```
//: io/BasicFileOutput.java
import java.io.*;

public class BasicFileOutput {
    static String file = "BasicFileOutput.out";
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new StringReader(
                BufferedInputFile.read("BasicFileOutput.java")));
        PrintWriter out = new PrintWriter(
            new BufferedWriter(new FileWriter(file)));
        int lineCount = 1;
        String s;
        while((s = in.readLine()) != null )
            out.println(lineCount++ + ": " + s);
        out.close();
        // Show the stored file:
        System.out.println(BufferedInputFile.read(file));
```

```
    }
} /* (Execute to see output) *////:~
```

Beim Schreiben der Zeilen in die Ausgabedatei werden Zeilennummern hinzugefügt. Beachten Sie hierbei, daß die Klasse `LineNumberReader` nicht auftritt (es ist eine alberne nutzlose Klasse). Wie Sie am obigen Beispiel sehen, ist es kein Problem, existierende Zeilennummern zu pflegen.

[67] Die `readLine()`-Methode gibt `null` zurück, wenn ein Eingabestrom leer ist. Die `close()`-Methode des von `out` referenzierten `PrintWriter`-Objektes wird explizit aufgerufen, da andernfalls der Ausgabepuffer eventuell nicht geleert wird, die Datei also unvollständig ist.

19.6.4.1 Abkürzung: Schreiben in eine Textdatei

[68] Seit der SE 5 hat die Klasse `PrintWriter` einen zusätzlichen Konstruktor, damit Sie nicht jedesmal, wenn Sie eine Textdatei öffnen und etwas hineinschreiben möchten, die Dekoratoren manuell anlegen müssen. Das folgende Beispiel zeigt *BasicFileOutput.java*, umgeschrieben für diesen Konstruktor:

```
//: io/FileOutputShortcut.java
import java.io.*;

public class FileOutputShortcut {
    static String file = ''FileOutputShortcut.out'';
    public static void main(String[] args) throws IOException {
        BufferedReader in = new BufferedReader(
            new StringReader(
                BufferedInputFile.read(''FileOutputShortcut.java''')));
        // Here's the shortcut:
        PrintWriter out = new PrintWriter(file);
        int lineCount = 1;
        String s;
        while((s = in.readLine()) != null )
            out.println(lineCount++ + '': '' + s);
        out.close();
        // Show the stored file:
        System.out.println(BufferedInputFile.read(file));
    }
} /* (Execute to see output) *////:~
```

Auch hier wird die Ausgabe gepuffert, aber Sie müssen den Puffer nicht mehr selbst anlegen. Leider gibt es für die übrigen der häufigeren Anwendungsfälle keine derartigen Abkürzungen, das heißt die typischen Ein-/Ausgabeoperationen bringen stets vergleichsweise viel überflüssigen Quelltext mit sich. Die in diesem Buch verwendete Hilfsklasse `net.mindview.util.TextFile` (Seiten 726–728) vereinfacht aber die häufigsten Anwendungsfälle.

Übungsaufgabe 12: (3) Ändern Sie Übungsaufgabe 8 so, daß das Programm zusätzlich eine Textdatei zum Schreiben öffnet. Schreiben Sie den Inhalt des `LinkedList`-Objektes mit Zeilennummern in diese Datei (ohne Verwendung der `LineNumber`-Klassen). ■

Übungsaufgabe 13: (3) Ändern Sie *BasicFileOutput.java* so, daß das Programm ein `LineNumberReader`-Objekt verwendet, um die Zeilennummern pflegen zu können. Überzeugen Sie sich davon, daß es einfacher ist, die Zeilennummern programmatisch neu zu schreiben. ■

Übungsaufgabe 14: (2) Schreiben Sie, beginnend bei *BasicFileOutput.java* ein Programm, das die Geschwindigkeit von gepufferten/ungepufferten Schreiboperationen vergleicht. ■

19.6.5 Schreiben und Lesen von binären Daten

[69] Ein `PrintWriter`-Objekt formatiert Daten in menschenlesbarer Weise. Sollen die Daten dagegen über einen anderen Datenstrom eingelesen werden, so verwenden Sie ein `DataOutputStream`-Objekt zum Schreiben und ein `DataInputStream`-Objekt zum Lesen dieser Daten. Der Typ des Datenstroms ist frei wählbar. Im folgenden Beispiel wird ein Datei mit gepuffertem Lese- beziehungsweise Schreibzugriff verwendet. Die Klassen `java.io.DataInputStream` und `java.io.DataOutputStream` sind byteorientiert, das heißt ihre Konstruktoren erwarten ein Argument vom Typ `InputStream` beziehungsweise `OutputStream`:

```
//: io/StoringAndRecoveringData.java
import java.io.*;

public class StoringAndRecoveringData {
    public static void main(String[] args) throws IOException {
        DataOutputStream out = new DataOutputStream(
            new BufferedOutputStream(
                new FileOutputStream("Data.txt")));
        out.writeDouble(3.14159);
        out.writeUTF("That was pi");
        out.writeDouble(1.41413);
        out.writeUTF("Square root of 2");
        out.close();
        DataInputStream in = new DataInputStream(
            new BufferedInputStream(
                new FileInputStream("Data.txt")));
        System.out.println(in.readDouble());
        // Only readUTF() will recover the
        // Java-UTF String properly:
        System.out.println(in.readUTF());
        System.out.println(in.readDouble());
        System.out.println(in.readUTF());
    }
} /* Output:
      3.14159
      That was pi
      1.41413
      Square root of 2
   *///:~
```

Wenn Sie ein `DataOutputStream`-Objekt verwenden, um Daten zu schreiben, garantiert Java, daß Sie diese Daten per `DataInputStream`-Objekt wieder korrekt einlesen können, unabhängig davon, welche Plattform zum Lesen beziehungsweise Schreiben verwendet wird. Diese Eigenschaft ist sehr wertvoll, wie jeder bestätigen wird, der sich mit plattformspezifischem Lesen und Schreiben von Daten auseinandergesetzt hat. Das Problem verschwindet, wenn Sie auf beiden Seiten Java verwenden[2].

[70] Wenn Sie ein `DataOutputStream`-Objekt verwenden, um eine Zeichenkette zu erzeugen, die Sie über ein `DataInputStream`-Objekt wieder einlesen können, so besteht der einzige verläßliche Weg darin, die UTF-8-Kodierung zu gebrauchen. In *StoringAndRecoveringData.java* übernehmen die Methoden `writeUTF()` und `readUTF()` diese Aufgabe. UTF-8 ist ein Mehrbyteformat und die Kodierungslänge eines Zeichens hängt vom verwendeten Zeichensatz ab. Wenn Sie ausschließlich oder größtenteils mit ASCII-Zeichen arbeiten (die nur sieben Bit benötigen), bedeutet die Verwendung von Unicode eine enorme Verschwendung von Speicherplatz auf der Festplatte beziehungsweise

[2] Eine andere Möglichkeit, um Daten zwischen verschiedenen Plattformen auszutauschen, ist XML und setzt nicht voraus, daß auf allen beteiligten Plattformen Java verwendet wird. XML wird in Abschnitt 19.13 behandelt.

Bandbreite, da UTF-8 ASCII-Zeichen in einem einzelnen Byte kodiert, Nicht-ASCII-Zeichen dagegen in zwei bis drei Bytes. Außerdem wird die Kodierungslänge in den beiden ersten Bytes der UTF-8-Zeichenkette gespeichert. Die Methoden `writeUTF()` und `readUTF()` verwenden allerdings eine Java-spezifische Variante von UTF-8 (die in der API-Dokumenation des Interfaces *`java.io.DataInput`* vollständig beschrieben wird). Wenn Sie eine mittels `writeUTF()` geschriebene Zeichenkette mit einem nicht in Java geschriebenen Programm lesen wollen, müssen Sie das Einlesen selbst programmieren.

[71] Die Methoden `writeUTF()` und `readUTF()` gestatten die gemischte Übergabe von `String`-Objekten und Werten primitiven Typs an ein `DataOutputStream`-Objekt mit der Gewißheit, daß die Daten korrekt als Unicodezeichenketten gespeichert und per `DataInputStream`-Objekt mühelos wieder ausgelesen werden können.

[72] Die `writeDouble()`-Methode übergibt einen `double`-Wert an den Ausgabestrom und die komplementäre `readDouble()`-Methode liest ihn aus dem Eingabestrom (es gibt entsprechende Methoden zum Lesen und Schreiben der übrigen Typen). Die korrekte Funktion dieser Methoden setzt aber voraus, daß Sie die genaue Position des Wertes im Datenstrom kennen, da Sie den `double`-Wert beispielsweise genauso gut als Folge von `byte`- oder `char`-Werten interpretieren können. Ziehen Sie in Betracht, daß die Serialisierung von Objekten oder XML (siehe Abschnitt 19.12 beziehungsweise Abschnitt 19.13) eventuell besser geeignet sind, um komplexe Datenstrukturen zu speichern und wieder aufzubauen.

Übungsaufgabe 15: (4) Lesen Sie die API-Dokumentation der Klassen `DataInputStream` und `DataOutputStream`. Schreiben Sie, ausgehend von *StoringAndRecoveringData.java*, ein Programm, das alle von den Klassen `DataInputStream` und `DataOutputStream` unterstützten Typen je einmal schreibt und anschließend wieder einliest. Vergewissern Sie sich, daß die geschriebenen Werte korrekt eingelesen werden. ■

19.6.6 Lesen und Schreiben von RandomAccessFile-Dateien

[73] Die Klasse `java.io.RandomAccessFile` verhält sich wie eine Kombination aus den Klassen `DataInputStream` und `DataOutputStream` (auch `RandomAccessFile` implementiert die Interfaces *`DataInput`* und *`DataOutput`*). Darüber hinaus gestattet die Methode `seek()`, den Dateizeiger zu positionieren und anschließend Werte zu ändern.

[74] Die Entscheidung für die Klasse `RandomAccessFile` setzt voraus, daß Sie die Struktur der Datei kennen, um Sie bestimmungsgemäß ändern zu können. Die Klasse `RandomAccessFile` verfügt über Methoden, um sowohl Werte primitiver Typen, als auch UTF-8-Zeichenketten zu lesen und zu schreiben. Ein Beispiel:

```
//: io/UsingRandomAccessFile.java
import java.io.*;

public class UsingRandomAccessFile {
    static String file = ''rtest.dat'';
    static void display() throws IOException {
        RandomAccessFile rf = new RandomAccessFile(file, ''r'');
        for(int i = 0; i < 7; i++)
            System.out.println(''Value '' + i + '': '' + rf.readDouble());
        System.out.println(rf.readUTF());
        rf.close();
    }
    public static void main(String[] args) throws IOException {
        RandomAccessFile rf = new RandomAccessFile(file, ''rw'');
```

```
        for(int i = 0; i < 7; i++)
            rf.writeDouble(i*1.414);
        rf.writeUTF("The end of the file");
        rf.close();
        display();
        rf = new RandomAccessFile(file, "rw");
        rf.seek(5*8);
        rf.writeDouble(47.0001);
        rf.close();
        display();
    }
} /* Output:
      Value 0: 0.0
      Value 1: 1.414
      Value 2: 2.828
      Value 3: 4.242
      Value 4: 5.656
      Value 5: 7.069999999999999
      Value 6: 8.484
      The end of the file
      Value 0: 0.0
      Value 1: 1.414
      Value 2: 2.828
      Value 3: 4.242
      Value 4: 5.656
      Value 5: 47.0001
      Value 6: 8.484
      The end of the file
   *///:~
```

Die `display()`-Methode öffnet die Datei *rtest.dat* und zeigt die darin enthaltenen sieben Einträge als `double`-Werte an. Die Datei wird in der `main()`-Methode angelegt, geöffnet und mit Werten initialisiert. Da ein `double`-Wert stets acht Byte beansprucht, bewegt `seek(5*8)` den Dateizeiger auf den fünften Wert.

[75] Wie bereits beschrieben, ist die Klasse `RandomAccessFile` effektiv unabhängig von den Hierarchien unter den byteorientierten Basisklassen `InputStream` und `OutputStream` beziehungsweise unter den zeichenorientierten Basisklassen `Reader` und `Writer`. Die Klasse `RandomAccessFile` implementiert zwar die Interfaces *`DataInput`* und *`DataOutput`*, unterstützt aber keine Dekoratoren, läßt sich also mit keiner von `InputStream` oder `OutputStream` abgeleiteten Klasse kombinieren. Sie müssen voraussetzen, daß `RandomAccessFile` gepuffert ist, diese Fähigkeit also nicht ergänzt werden kann.

[76] Sie haben nur eine Möglichkeit, in die Erzeugung eines `RandomAccessFile`-Objektes einzugreifen: Das zweite Argument des Konstruktors gibt an, ob die Datei zum Lesen („r“) oder zum Lesen und Schreiben („rw“) geöffnet werden soll.

[77] Beachten Sie die Abbildung von Dateien in den Arbeitsspeicher (*memory-mapped files*) im `java.nio`-Package als Alternative zu `RandomAccessFile`.

Übungsaufgabe 16: (2) Lesen Sie die API-Dokumentation der Klasse `RandomAccessFile`. Schreiben Sie, ausgehend von *UsingRandomAccess.java*, ein Programm, das alle von der Klasse `RandomAccessFile` unterstützten Typen je einmal schreibt und anschließend wieder einliest. Vergewissern Sie sich, daß die geschriebenen Werte korrekt eingelesen werden. ■

19.6.7 Pipes

[78] Die Klassen `PipedInputStream`, `PipedOutputStream`, `PipedReader` und `PipedWriter` (alle im `java.io`-Package) wurden im vorliegenden Kapitel nur kurz erwähnt. Dieser Umstand soll keineswegs andeuten, daß diese Klassen nicht gebraucht werden. Ihre Bewandnis erschließt sich aber erst im Zusammenhang mit Threads ([!]*(piped streams)*) dienen der Kommunikation zwischen Threads, siehe Unterabschnitt 22.5.5).

19.7 Hilfsklassen zum Lesen und Schreiben von Dateien

[79] Eine sehr häufige Programmieraufgabe besteht darin, eine Datei in den Arbeitsspeicher einzulesen, zu modifizieren und anschließend wieder zurückzuschreiben. Die Ein-/Ausgabebibliothek von Java verlangt, daß Sie relativ viel Quelltext schreiben müssen, um diese Standardoperationen ausführen zu können. Es gibt keine Hilfsklassen oder -methoden, die Ihnen diese Arbeit erleichtern. Die Dekoratoren erschweren zusätzlich, sich zu merken, wie ein bestimmter Dateityp geöffnet werden soll. Es ist daher sinnvoll, eine Bibliothek von Hilfsklassen anzulegen, welche Ihnen die grundlegenden Arbeitsschritte abnehmen. Seit der SE 5 verfügt die Klasse `PrintWriter` über einen Konstruktor, mit dem Sie eine Textdatei mühelos zum Schreiben öffnen können. Es gibt noch eine Reihe weiterer Anwendungsfälle, die immer wieder auftreten und es ist sinnvoll, die redundanten Anweisungen in Hilfsklassen oder -methoden auszulagern.

19.7.1 Lesen und Schreiben von Textdateien

[80] Die Hilfsklasse `net.mindview.util.TextFile` taucht in vielen Beispielen dieses Buches auf, um das Lesen und Schreiben von Textdateien zu erleichtern. Die Klasse verfügt über statische Methoden, um Textdateien als eine einzige Zeichenkette zu Lesen und zu Schreiben. Die beiden Konstruktoren gestatten, ein `TextFile`-Objekt so zu erzeugen, daß die Zeilen der Datei in einer `ArrayList` gespeichert werden. (Sie haben also die gesamte `ArrayList`-Funktionalität zur Verfügung, um den Dateiinhalt zu bearbeiten.):

```
//: net/mindview/util/TextFile.java
// Static functions for reading and writing text files as
// a single string, and treating a file as an ArrayList.
package net.mindview.util;
import java.io.*;
import java.util.*;

public class TextFile extends ArrayList<String> {
    // Read a file as a single string:
    public static String read(String fileName) {
        StringBuilder sb = new StringBuilder();
        try {
            BufferedReader in= new BufferedReader(new FileReader(
                new File(fileName).getAbsoluteFile()));
            try {
                String s;
                while((s = in.readLine()) != null) {
                    sb.append(s);
                    sb.append("\n");
                }
            } finally {
                in.close();
```

```
        }
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
    return sb.toString();
}
// Write a single file in one method call:
public static void write(String fileName, String text) {
    try {
        PrintWriter out = new PrintWriter(
            new File(fileName).getAbsoluteFile());
        try {
            out.print(text);
        } finally {
            out.close();
        }
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
// Read a file, split by any regular expression:
public TextFile(String fileName, String splitter) {
    super(Arrays.asList(read(fileName).split(splitter)));
    // Regular expression split() often leaves an empty
    // String at the first position:
    if(get(0).equals("")) remove(0);
}
// Normally read by lines:
public TextFile(String fileName) {
    this(fileName, "\n");
}
public void write(String fileName) {
    try {
        PrintWriter out = new PrintWriter(
            new File(fileName).getAbsoluteFile());
        try {
            for(String item : this)
                out.println(item);
        } finally {
            out.close();
        }
    } catch(IOException e) {
        throw new RuntimeException(e);
    }
}
// Simple test:
public static void main(String[] args) {
    String file = read("TextFile.java");
    write("test.txt", file);
    TextFile text = new TextFile("test.txt");
    text.write("test2.txt");
    // Break into unique sorted list of words:
    TreeSet<String> words = new TreeSet<String>(
        new TextFile("TextFile.java", "\\W+"));
    // Display the capitalized words:
    System.out.println(words.headSet("a"));
}
```

```
} /* Output:
      [0, ArrayList, Arrays, Break, BufferedReader, BufferedWriter, Clean, Display,
File, FileReader, FileWriter, IOException, Normally, Output, PrintWriter, Read, Regular,
RuntimeException, Simple, Static, String, StringBuilder, System, TextFile, Tools,
TreeSet, W, Write]
   *///:~
```

Die `read()`-Methode hängt jede Zeile zusammen mit einem Zeilenumbruch an den Inhalt eines `StringBuilder`-Objektes (der ursprüngliche Zeilenumbruch wird beim Einlesen gelöscht) und gibt den gesamten Dateiinhalt als `String`-Objekt zurück. Die `write()`-Methode öffnet eine Datei und schreibt den Inhalt des `String`-Objektes hinein.

[81] Beachten Sie, daß der `close()`-Aufruf zu jeder geöffneten Datei in einer `finally`-Klausel steht, um zu gewährleisten, daß die Datei ordentlich geschlossen wird.

[82] Der erste Konstruktor ruft die `read()`-Methode auf, um den Dateiinhalt in ein `String`-Objekt umzuwandeln. Die `String`-Methode `split()` trennt diese Zeichenkette an den Zeilenumbrüchen in einzelne Zeilen auf. (Wenn sie die Hilfsklasse `TextFile` häufig benutzen, lohnt es sich, den Konstruktor effizienter umzuschreiben.) Es gibt keine `join()`-Methode, so daß die nicht-statische `write()`-Methode Zeile um Zeile „von Hand“ schreiben muß.

[83] Da die Hilfsklasse `TextFile` das Lesen und Schreiben von Dateien vereinfachen soll, wurden alle Ausnahmen vom Typ `IOException` in Ausnahmen vom Typ `RuntimeException` verpackt, so daß der Aufrufer keine `try/catch`-Klauseln braucht. Eventuell müssen Sie eine geänderte Version anlegen, welche die Ausnahmen vom Typ `IOException` an den Aufrufer weitergibt.

[84] In der `main()`-Methode weist ein einfacher Test nach, daß die Methoden der Hilfsklasse funktionieren.

[85] Obwohl die Hilfsklasse `TextFile` nur wenige Zeilen lang ist, spart ihre Verwendung viel Zeit und erleichtert das Leben, wie Sie bei einigen der folgenden Beispiele in diesem Kapitel sehen werden.

[86] Eine andere Möglichkeit, Textdateien einzulesen, ist die Klasse `java.util.Scanner` (vorhanden ab der SE 5). Diese Klasse kann Textdateien allerdings nur lesen, nicht schreiben. Die `Scanner`-Klasse gehört nicht zum `java.io`-Package und dient hauptsächlich dazu, Scanner für Programmiersprachen zu entwickeln.

Übungsaufgabe 17: (4) Schreiben Sie ein Programm, das mittels der Klasse `TextFile` und eines *`Map<Character, Integer>`*-Objektes die Vorkommen der verschiedenen Buchstaben in einer Datei zählt. (Enthält die Datei beispielsweise zwölfmal den Buchstaben „a“, so enthält das mit dem `Character`-Objekt verknüpfte `Integer`-Objekt den Wert 12.) ■

Übungsaufgabe 18: (1) Ändern Sie *TextFile.java* so, daß die Methoden Ausnahmen vom Typ `IOException` an den Aufrufer weitergeben. ■

19.7.2 Lesen von Binärdateien

[87] Die Hilfsklasse `io.BinaryFile` ähnelt `TextFile` und vereinfacht das Einlesen von Binärdateien:

```
//: net/mindview/util/BinaryFile.java
// Utility for reading files in binary form.
package net.mindview.util;
import java.io.*;

public class BinaryFile {
    public static byte[] read(File bFile) throws IOException{
        BufferedInputStream bf = new BufferedInputStream(
```

```
            new FileInputStream(bFile));
        try {
            byte[] data = new byte[bf.available()];
            bf.read(data);
            return data;
        } finally {
            bf.close();
        }
    }
    public static byte[] read(String bFile) throws IOException {
        return read(new File(bFile).getAbsoluteFile());
    }
} ///:~
```

Eine Version der überladenen `read()`-Methode erwartet ein `File`-, die andere ein `String`-Argument (den Dateinamen). Beide geben ein `byte`-Array zurück.

[88] Die `available()`-Methode wird verwendet, um ein Array passender Größe zu erzeugen und die gewählte Variante der `BufferedInputStream`-Methode `read()` füllt das Array.

Übungsaufgabe 19: (2) Schreiben Sie ein Programm, das mittels `BinaryFile` und eines *`Map<Byte, Integer>`*-Objektes die Vorkommen der verschiedenen Bytes in einer Datei zählt. ■

Übungsaufgabe 20: (4) Verifizieren Sie mittels `Directory.walk()` und `BinaryFile`, daß alle *.class* Dateien in einem Verzeichnisbaum mit den hexadezimalen Zeichen `CAFEBABE` beginnen. ■

19.8 Standardein-/ausgabe und -fehlerkanal

[89] Die Bezeichnungen „Standardeingabe-“, „Standardausgabe-“ und „Standardfehlerkanal“ beziehen sich auf das Unixkonzept, daß ein Programm über einen einzelnen Datenstrom Informationen austauscht (dieser Ansatz wurde bei Windows und verschiedenen anderen Betriebssysteme reproduziert). Alle Eingaben eines Programmes stammen aus der Standardeingabe, alle Ausgabe eines Programmes werden an die Standardausgabe und alle Fehlermeldungen an den Standardfehlerkanal gesendet. Der Vorteil der Standardein-/ausgabe besteht darin, daß Programme miteinander verknüpft werden können, wobei die Standardausgabe des einen Programms an die Standardeingabe des folgenden Programms übergeben wird. Damit steht ein mächtiger Mechanismus zur Verfügung.

19.8.1 Lesen von der Standardeingabe

[90] Dem Standardein-/ausgabekonzept folgend, stellt Java die Objekte `System.out`, `System.in` und `System.err` zur Verfügung. Sie haben an vielen Stellen in diesem Buch gesehen, wie Daten über `System.out` an die Standardausgabe übergeben werden. `System.out` und `System.err` sind als `PrintStream`-Objekt „vorverpackt“. `System.in` ist dagegen nicht vorverpackt und vom Typ `InputStream`. Das bedeutet, daß Sie `System.out` und `System.err` direkt verwenden können, während bei `System.in` stets eine Verpackung erforderlich ist, bevor Sie Daten aus dem Datenstrom lesen können.

[91] Eingaben werden typischerweise mittels `readLine()` zeilenweise gelesen. Sie müssen dazu `System.in` in einen `BufferedReader` verpacken (zuvor muß `System.in` per `InputStreamReader` in ein `Reader`-Objekt umgewandelt werden). Das folgende Beispiel wiederholt jede eingegebene Zeile:

```
//: io/Echo.java
// How to read from standard input.
```

```
// {RunByHand}
import java.io.*;

public class Echo {
    public static void main(String[] args) throws IOException {
        BufferedReader stdin = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = stdin.readLine()) != null && s.length()!= 0)
            System.out.println(s);
        // An empty line or Ctrl-Z terminates the program
    }
} ///:~
```

Die `throws`-Klausel ist vorhanden, da die `readLine()`-Methode eine Ausnahme vom Typ `IOException` auswerfen kann. Beachten Sie, daß `System.in`, wie die meisten Datenströme, gepuffert werden sollte.

Übungsaufgabe 21: (1) Schreiben Sie ein Programm, das von der Standardeingabe liest, alle Buchstaben in Großbuchstaben umwandelt und das Ergebnis über die Standardausgabe wieder ausgibt. Leiten Sie den Inhalt einer Datei in dieses Programm um (die Schreibweise für die Umleitung ist betriebssystemabhängig). ■

19.8.2 Vorschalten eines PrintWriters bei System.out

[92] `System.out` ist vom Typ `PrintStream`, der wiederum vom `OutputStream` abgeleitet ist. Die Klasse `PrintWriter` hat einen Konstruktor der ein Argument vom Typ `OutputStream` erwartet. Mit Hilfe dieses Konstruktors können Sie `System.out` ein `PrintWriter`-Objekt vorschalten:

```
//: io/ChangeSystemOut.java
// Turn System.out into a PrintWriter.
import java.io.*;

public class ChangeSystemOut {
    public static void main(String[] args) {
        PrintWriter out = new PrintWriter(System.out, true);
        out.println("Hello, world");
    }
} /* Output:
      Hello, world
   *////:~
```

Es kommt darauf an, die zweiargumentige Version des `PrintWriter`-Konstruktors zu verwenden und `true` als zweites Argument zu übergeben, um die automatische Leerung des Puffers zu aktivieren. Andernfalls bekommen Sie unter Umständen keine Ausgabe.

19.8.3 Umleiten der Standardein- und ausgabe

[93] Die statischen Methoden `setIn(InputStream)`, `setOut(PrintStream)` und `setErr(PrintStream)` der Klasse `java.lang.System` gestatten die Umleitung von Standardein-/ausgabe und -fehlerkanal.

[94] Das Umleiten der Ausgabe ist nützlich, wenn die Ausgabe eines Programms auf dem Bildschirm

umfangreich ist und schneller aus dem Fenster verschwindet, als Sie sie lesen können[3]. Das Umleiten der Eingabe ist dagegen sinnvoll, wenn Sie ein Programm von der Kommandozeile aus wiederholt mit bestimmten Argumenten aufrufen wollen. Das folgende Beispiel zeigt die Anwendung der obigen drei `System`-Methoden:

```
//: io/Redirecting.java
// Demonstrates standard I/O redirection.
import java.io.*;

public class Redirecting {
    public static void main(String[] args) throws IOException {
        PrintStream console = System.out;
        BufferedInputStream in = new BufferedInputStream(
            new FileInputStream("Redirecting.java"));
        PrintStream out = new PrintStream(
            new BufferedOutputStream(
                new FileOutputStream("test.out")));
        System.setIn(in);
        System.setOut(out);
        System.setErr(out);
        BufferedReader br = new BufferedReader(
            new InputStreamReader(System.in));
        String s;
        while((s = br.readLine()) != null)
            System.out.println(s);
        out.close(); // Remember this!
        System.setOut(console);
    }
} ///:~
```

Das Programm verknüpft die Standardeingabe mit einer Datei und verbindet Standardausgabe und -fehlerkanal mit einer anderen Datei. Beachten Sie, daß das Programm zu Beginn eine Referenz auf das ursprüngliche `System.out`-Objekt speichert und die Standardausgabe vor Programmende wieder auf die ursprüngliche Einstellung zurücksetzt.

[95] Die Ein-/Ausgabeumleitung wirkt auf byteorientierte Datenströme, nicht aber auf zeichenorientierte. Daher verwendet das obige Programm `InputStream` und `OutputStream` anstelle von `Reader` und `Writer`.

19.9 Prozeßsteuerung

[96] Es kommt häufig vor, daß Sie aus einem Java-Programm heraus ein Kommando Ihres Betriebssystems aufrufen und die Ein- und Ausgabe dieses Kommandos kontrollieren müssen. Die Java-Bibliothek stellt für solche Zwecke eigene Klassen zur Verfügung.

[97] Eine solche Aufgabe lautet, ein Kommando aufzurufen und seine Ausgabe über die Konsole auszugeben. Dieser Abschnitt stellt eine Hilfsklasse vor, um diese Aufgabe zu erleichtern.

[98] Beim Anwenden dieser Hilfsklasse können zwei Fehlersituationen eintreten: Normale Fehler, die Ausnahmen hervorrufen, werden in Ausnahmen vom Typ `RuntimeException` verpackt und erneut ausgeworfen. Fehler durch die Ausführung des Betriebssystemkommandos selbst werden mittels einer separaten Ausnahme gemeldet:

[3] In Kapitel 23 behandeln wir eine elegantere Lösung für dieses Problem, nämlich einen Textbereich mit Bildlaufleiste in einer graphischen Benutzeroberfläche.

```
//: net/mindview/util/OSExecuteException.java
package net.mindview.util;

public class OSExecuteException extends RuntimeException {
    public OSExecuteException(String why) { super(why); }
} ///:~
```

[99] Sie führen ein Kommando aus, in dem Sie der `command()`-Methode der Hilfsklasse `net.mindview.util.OSExecute` dieselbe Zeichenkette übergeben, die Sie zum Aufruf des Kommandos über die Konsole eingeben würden. Diese Zeichenkette wird dem Konstruktor der Klasse `java.lang.ProcessBuilder` übergeben (der eines oder mehrere `String`-Objekte erwartet). Anschließend wird das `ProcessBuilder`-Objekt gestartet:

```
//: net/mindview/util/OSExecute.java
// Run an operating system command
// and send the output to the console.
package net.mindview.util;
import java.io.*;

public class OSExecute {
    public static void command(String command) {
        boolean err = false;
        try {
            Process process = new ProcessBuilder(command.split(" ")).start();
            BufferedReader results = new BufferedReader(
                new InputStreamReader(process.getInputStream()));
            String s;
            while((s = results.readLine())!= null)
                System.out.println(s);
            BufferedReader errors = new BufferedReader(
                new InputStreamReader(process.getErrorStream()));
            // Report errors and return nonzero value
            // to calling process if there are problems:
            while((s = errors.readLine())!= null) {
                System.err.println(s);
                err = true;
            }
        } catch(Exception e) {
            // Compensate for Windows 2000, which throws an
            // exception for the default command line:
            if(!command.startsWith("CMD /C"))
                command("CMD /C " + command);
            else
                throw new RuntimeException(e);
        }
        if(err)
            throw new OSExecuteException("Errors executing " + command);
    }
} ///:~
```

Die Methode `getInputStream()` fängt die Standardausgabe des ausgeführten Kommandos ab (ein `InputStream` kann gelesen werden).

[100] Die Ausgabe des Kommandos erfolgt zeilenweise, kann also per `readLine()` gelesen werden. Die Hilfsklasse `OSExecute` gibt die gelesenen Zeilen einfach aus. Sie können die Ausgabe aber auch abfangen und als Rückgabewert der `command()`-Methode verwenden.

[101] Die Fehlermeldungen des Kommandos werden dem Standardfehlerkanal übergeben und zuvor per `getErrorStream()` abgefangen. Treten Fehler auf, so werden die entsprechenden Meldungen

sowohl ausgegeben als auch eine Ausnahme vom Typ `OSExecuteException` ausgeworfen, die vom aufrufenden Programm behandelt wird. Ein Anwendungsbeispiel für die Hilfsklasse `OSExecute`:

```
//: io/OSExecuteDemo.java
// Demonstrates standard I/O redirection.
import net.mindview.util.*;

public class OSExecuteDemo {
    public static void main(String[] args) {
        OSExecute.command(''javap OSExecuteDemo'');
    }
} /* Output:
      Compiled from ''OSExecuteDemo.java''
      public class OSExecuteDemo extends java.lang.Object{
      public OSExecuteDemo();
      public static void main(java.lang.String[]);
      }
   *///:~
```

Das Programm ruft `javap` auf (gehört zum Java Development Kit), um die Klasse `OSExecuteDemo` zu dekompilieren.

Übungsaufgabe 22: (5) Ändern Sie *OSExecute.java* so, daß das Programm die Ausgaben des aufgerufenen Kommandos nicht an die Standardausgabe sendet, sondern als *`List<String>`*-Objekt zurückgibt. ■

19.10 Die neue Ein-/Ausgabebibliothek

[102] Die mit Version 1.4 des Java Development Kits eingeführte „neue“ Ein-/Ausgabebibliothek (Package `java.nio`) hat ein wesentliches Ziel: Geschwindigkeit. Tatsächlich wurden sogar Teile des traditionellen `java.io`-Packages unter Anwendung des neuen `java.nio`-Packages umgeschrieben, um den Geschwindigkeitsvorteil zu nutzen. Sie profitieren also auf jeden Fall von der neuen Bibliothek, auch wenn Sie die Klassen nicht explizit verwenden. Der Geschwindigkeitszuwachs betrifft sowohl die in diesem Abschnitt beschriebene Dateiein-/ausgabe, als auch die in *Thinking in Enterprise Java* beschriebene Ein-/Ausgabe über ein Netzwerk hinweg („Netzwerkein-/ausgabe“).

[103] Die höhere Geschwindigkeit ergibt sich aus der Verwendung von Datenstrukturen, die dem Ein-/Ausgabesystem des Betriebssystems nahe sind: Kanäle (*channels*) und Puffer (*buffer*). Sie können sich die Funktionsweise ähnlich wie bei einem Bergwerk vorstellen: Der Kanal entspricht dem Schacht, der zur Lagerstätte (der Datei) führt und der Puffer entspricht der Lore, mit der die Kohle transportiert wird. Die Lore kommt vollbeladen ans Tageslicht und wird entladen. Sie kommunizieren also nicht direkt mit dem Kanal, sondern über den Puffer und senden ihn durch den Kanal hin und zurück. Der Kanal nimmt entweder Daten aus dem Puffer oder deponiert sie darin.

[104] Es gibt nur einen Puffertyp, der direkt mit dem Kanal kommuniziert, nämlich `java.nio.ByteBuffer`, welcher nicht aufbereitete (*raw*) Bytes transportiert. Ein Blick in die API-Dokumentation zeigt, daß ein Puffer vom Typ `ByteBuffer` leicht zu bedienen ist. Beim Erzeugen des Pufferobjektes geben Sie an, wieviel Speicherplatz allokiert werden soll und rufen anschließend die entsprechenden Methoden auf, um entweder unaufbereitete Bytes oder Werte primitiven Typs in den Puffer zu schreiben oder von dort zu entnehmen. Es ist nicht möglich, einem Puffer ein Objekt zu übergeben, nicht einmal beim Typ `String`. Der Ansatz ist ziemlich systemnah, eben weil sich die Operationen effizienter auf die meisten Betriebssysteme abbilden lassen.

[105] Drei Klassen der traditionellen Ein-/Ausgabebibliothek wurden überarbeitet, um ein `java-`

`.nio.channels.FileChannel`-Objekt erzeugen zu können: `FileInputStream`, `FileOutputStream` und `RandomAccessFile`. Beachten Sie, daß diese drei Klassen byteorientiert sind, im Einklang mit der Systemnähe des `java.nio`-Packages. Die zeichenorientierten Klassen aus den Hierarchien unter den Basisklassen `Reader` und `Writer` sind nicht in der Lage, ein `FileChannel`-Objekt zu erzeugen. Allerdings verfügt die Klasse `java.nio.channels.Channels` über Hilfsmethoden, um aus Kanälen `Reader`- und `Writer`-Objekte zu erhalten.

[106] Das folgende Beispiel gebraucht nacheinander alle drei Datenstromklassen (`FileInputStream`, `FileOutputStream` und `RandomAccessFile`), um ein nur schreibbares, ein les- und schreibbares und ein nur lesbares `FileChannel`-Objekt zu erzeugen:

```
//: io/GetChannel.java
// Getting channels from streams
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class GetChannel {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        // Write a file:
        FileChannel fc = new FileOutputStream("data.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text ".getBytes()));
        fc.close();
        // Add to the end of the file:
        fc = new RandomAccessFile("data.txt", "rw").getChannel();
        fc.position(fc.size()); // Move to the end
        fc.write(ByteBuffer.wrap("Some more".getBytes()));
        fc.close();
        // Read the file:
        fc = new FileInputStream("data.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        while(buff.hasRemaining())
            System.out.print((char)buff.get());
    }
} /* Output:
      Some text Some more
   *///:~
```

Bei allen drei Datenstromklassen gibt die `getChannel()`-Methode eine Referenz auf ein `FileChannel`-Objekt zurück. Ein `FileChannel`-Objekt ist leicht bedienbar: Sie können einen Puffer vom Typ `ByteBuffer` zum Lesen oder Schreiben übergeben und Abschnitte einer Datei exklusiv für den Zugriff durch nur einen Thread sperren (siehe Unterunterabschnitt 19.10.7.1).

[107] Eine Möglichkeit, um Bytes in einem Puffer vom Typ `ByteBuffer` zu deponieren besteht darin, sie mit einer der `put`-Methoden für einzelne Bytes, mehrere Bytes oder Werte primitiven Typs zu übergeben. Sie können aber auch, wie im obigen Beispiel, per `wrap()`-Methode ein bereits vorhandenes `byte`-Array in einem `ByteBuffer`-Objekt verpacken. Dabei wird das unterliegende `byte`-Array nicht kopiert, sondern als Speicherplatz des neu erzeugten `ByteBuffer`-Puffers verwendet. We say that the ByteBuffer is "backed by" the array.

[108] Die Datei *data.txt* wird per `RandomAccessFile` ein weiteres mal geöffnet. Beachten Sie, daß Sie den Dateizeiger über das `FileChannel`-Objekt positionieren können. Hier wird der Zeiger auf das Dateiende gesetzt, so daß neue Daten angehängt werden.

[109] Für Nur-Lesezugriff müssen Sie den Speicherplatz für den `ByteBuffer`-Puffer mit Hilfe der statischen Methode `allocate()` explizit allokieren. Das Ziel der neuen Ein-/Ausgabebibliothek von Java besteht darin, große Datenmengen schnell bewegen zu können, wobei sich die Puffergröße auswirkt. Das oben verwendete kB ist wahrscheinlich etwas zu klein (experimentieren Sie mit Ihrer Anwendung, um die beste Größe zu bestimmen).

[110] Sie erhalten eventuell einen noch größeren Geschwindigkeitszuwachs, wenn Sie statt der `allocate()`- die ebenfalls statische `allocateDirect()`-Methode wählen, um einen „direkten“ Puffer anzulegen, der unter Umständen noch besser mit dem Betriebssystem zusammenarbeitet. Allerdings sind die Unkosten beim Allokieren des Speichers höher und die Implementierung ist betriebssystemabhängig. Sie müssen wiederum mit Ihrer Anwendung experimentieren, um festzustellen, ob sich ein „direkter“ Puffer lohnt.

[111] Wenn Sie die `read()`-Methode aufgerufen haben, um das `FileChannel`-Objekt anzuweisen, Daten im `ByteBuffer`-Puffer zu speichern, müssen Sie den Puffer durch Aufrufen der `flip()`-Methode zur Entnahme der Daten umstellen (diese Forderung wirkt undurchdacht, bedenken Sie aber, daß die Operation systemnah stattfindet und der Geschwindigkeitsoptimierung dient). Würde der Puffer für eine weitere Leseoperation gebraucht, so müßten wir vor jedem `read()`-Aufruf die `clear()`-Methode aufrufen, um den Puffer vorzubereiten. Das nächste Beispiel ist ein einfaches Kopierprogramm für Dateien:

```
//: io/ChannelCopy.java
// Copying a file using channels and buffers
// {Args: ChannelCopy.java test.txt}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class ChannelCopy {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        ByteBuffer buffer = ByteBuffer.allocate(BSIZE);
        while(in.read(buffer) != -1) {
            buffer.flip(); // Prepare for writing
            out.write(buffer);
            buffer.clear(); // Prepare for reading
        }
    }
} ///:~
```

[112] Das Programm erzeugt je ein `FileChannel`-Objekt zum Lesen beziehungsweise zum Schreiben und allokiert einen Puffer. Das Ende der Eingabe ist erreicht, wenn die `FileChannel`-Methode `read()` den Wert -1 zurückgibt (zweifellos ein Überbleibsel von Unix und C). Nach jedem `read()`, also der Datenübergabe an den Puffer, bereitet `flip()` den Puffer für die Datenentnahme durch `write()` vor. Nach Verarbeitung der `write()`-Methode befinden sie die Daten noch immer im Puffer. Die `clear()`-Methode setzt alle internen Zeiger zurück, so daß der Puffer per `read()` neuen Daten aufnehmen kann.

[113] Das obige Programm ist noch nicht die optiomale Vorgehensweise für die geforderte Operation.

Die speziellen Methoden `transferTo()` und `transferFrom()` gestatten, einen Kanal direkt mit einem anderen zu verbinden:

```
//: io/TransferTo.java
// Using transferTo() between channels
// {Args: TransferTo.java TransferTo.txt}
import java.nio.channels.*;
import java.io.*;

public class TransferTo {
    public static void main(String[] args) throws Exception {
        if(args.length != 2) {
            System.out.println("arguments: sourcefile destfile");
            System.exit(1);
        }
        FileChannel
            in = new FileInputStream(args[0]).getChannel(),
            out = new FileOutputStream(args[1]).getChannel();
        in.transferTo(0, in.size(), out);
        // Or:
        // out.transferFrom(in, 0, in.size());
    }
} ///:~
```

Sie werden diesen Anwendungsfall nicht häufig brauchen, aber es ist gut, davon zu wissen.

19.10.1 Datenkonvertierung

[114] Im Beispiel *GetChannel.java* fällt auf, daß die Daten vor dem Schreiben in die Ausgabedatei Byte für Byte gelesen und von `byte` nach `char` umgewandelt werden. Das wirkt ein wenig primitiv. Die API-Dokumentation der `CharBuffer`-Methode `toString()` besagt, daß die Methode ein `String`-Objekt zurückgibt, welches die Zeichen des Puffers enthält. Da ein Puffer vom Typ `ByteBuffer` mit Hilfe der `asCharBuffer()`-Methode als `CharBuffer` betrachtet werden kann, liegt des nahe, diese `toString()`-Methode zu verwenden. Die erste Ausgabe des folgenden Beispiel zeigt jedoch, daß dieser Weg nicht gangbar ist.

```
//: io/BufferToText.java
// Converting text to and from ByteBuffers
import java.nio.*;
import java.nio.channels.*;
import java.nio.charset.*;
import java.io.*;

public class BufferToText {
    private static final int BSIZE = 1024;
    public static void main(String[] args) throws Exception {
        FileChannel fc =
            new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap("Some text".getBytes()));
        fc.close();
        fc = new FileInputStream("data2.txt").getChannel();
        ByteBuffer buff = ByteBuffer.allocate(BSIZE);
        fc.read(buff);
        buff.flip();
        // Doesn't work:
        System.out.println(buff.asCharBuffer());
        // Decode using this system's default Charset:
```

```
        buff.rewind();
        String encoding = System.getProperty("file.encoding");
        System.out.println("Decoded using " + encoding + ": "
            + Charset.forName(encoding).decode(buff));
        // Or, we could encode with something that will print:
        fc = new FileOutputStream("data2.txt").getChannel();
        fc.write(ByteBuffer.wrap(
            "Some text".getBytes("UTF-16BE")));
        fc.close();
        // Now try reading again:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
        // Use a CharBuffer to write through:
        fc = new FileOutputStream("data2.txt").getChannel();
        buff = ByteBuffer.allocate(24); // More than needed
        buff.asCharBuffer().put("Some text");
        fc.write(buff);
        fc.close();
        // Read and display:
        fc = new FileInputStream("data2.txt").getChannel();
        buff.clear();
        fc.read(buff);
        buff.flip();
        System.out.println(buff.asCharBuffer());
    }
} /* Output:
      ????
      Decoded using Cp1252: Some text
      Some text
      Some text
   *///:~
```

[115] Der Puffer enthält unaufbereitete Bytes (Rohdaten). Um diese Bytes in Zeichen umzuwandeln, müssen wir die Bytes entweder vor der Übergabe an den Puffer kodieren (so daß sie beim Auslesen bereits interpretierbar sind) oder nach der Entnahme dekodieren. Das läßt sich mit Hilfe der Klasse `java.nio.charset.Charset` bewerkstelligen, die über die erforderlichen Methoden verfügt, um Bytes bezüglich vieler verschiedener Zeichensätze zu kodieren:

```
//: io/AvailableCharSets.java
// Displays Charsets and aliases
import java.nio.charset.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class AvailableCharSets {
    public static void main(String[] args) {
        SortedMap<String,Charset> charSets = Charset.availableCharsets();
        Iterator<String> it = charSets.keySet().iterator();
        while(it.hasNext()) {
            String csName = it.next();
            printnb(csName);
            Iterator aliases = charSets.get(csName).aliases().iterator();
            if(aliases.hasNext())
                printnb(": ");
            while(aliases.hasNext()) {
```

```
                    printnb(aliases.next());
                    if(aliases.hasNext())
                        printnb(", ");
                }
                print();
            }
        }
    } /* Output:
          Big5: csBig5
          Big5-HKSCS: big5-hkscs, big5hk, big5-hkscs:unicode3.0, big5hkscs, Big5_HKSCS
          EUC-JP: eucjis, x-eucjp, csEUCPkdFmtjapanese, eucjp, \
          Extended_UNIX_Code_Packed_Format_for_Japanese, x-euc-jp, euc_jp
          EUC-KR: ksc5601, 5601, ksc5601_1987, ksc_5601, ksc5601-1987, euc_kr, \
          ks_c_5601-1987, euckr, csEUCKR
          GB18030: gb18030-2000
          GB2312: gb2312-1980, gb2312, EUC_CN, gb2312-80, euc-cn, euccn, x-EUC-CN
          GBK: windows-936, CP936
          ...
       *///:~
```

Wenn Sie im Beispiel *BufferToText.java* per `rewind()` an den Anfang des Puffers zurückkehren und den plattformspezifischen Standardzeichensatz wählen, um die Bytes per `decode()` zu dekodieren, läßt sich das resultierende `CharBuffer`-Objekt in menschlesbarer Form über die Konsole ausgeben. Sie können den Standardzeichensatz über die `System`-Methode `getProperty("file.encoding")` abfragen (das zurückerhaltene `String`-Objekt gibt den Namen des Zeichensatzes an). Die `Charset`-Methode `forName()` erzeugt, mit dem Standardzeichensatz aufgerufen, das `Charset`-Objekt mit dem Sie den Pufferinhalt dekodieren können.

[116] Eine andere Lösung besteht darin, den Pufferinhalt per `encode()`-Methode bezüglich eines Zeichensatzes zu kodieren, der dem Pufferinhalt beim Auslesen eine druckbare Gestalt gibt (dritter Abschnitt des Beispiels *BufferToText.java*). Hier wurde UTF-16BE gewählt, um die Daten in die Datei zu schreiben. Beim Auslesen genügt es, den Puffer in ein `CharBuffer`-Objekt umzuwandeln, um den erwarteten Text zu erhalten.

[117] Abschließend sehen Sie, was geschieht, wenn Sie über den `CharBuffer`-„Darstellungsmodus" Daten in einen Puffer vom Typ `ByteBuffer` schreiben (mehr dazu in Unterabschnitt 19.10.3). Beachten Sie, daß für den `ByteBuffer`-Puffer 24 Byte allokiert werden. Da jeder `char`-Wert zwei Byte beansprucht, genügt der angeforderte Speicherplatz für zwölf `char`-Werte, wobei „Some text" lediglich aus neun `char`-Werten besteht. *The remaining zero bytes still appear in the representation of the CharBuffer produced by its toString(), as you can see in the output.*

Übungsaufgabe 23: (6) Schreiben Sie eine Hilfsmethode, die den Inhalt eines Puffers vom Typ `ByteBuffer` ausgibt, bis das erste nicht druckbare Zeichen auftritt. ■

19.10.2 Abfragen von Werten primitiven Typs

[118] Obwohl ein Puffer vom Typ `ByteBuffer` nur Bytes enthält, verfügt die Klasse `ByteBuffer` über Methoden, um den Pufferinhalt in Werte der verschiedenen primitiven Typen umzuwandeln. Das folgende Beispiel zeigt die Anwendung dieser Methoden zur Übergabe beziehungsweise Entnahme von Werten für einige primitive Typen:

```
//: io/GetData.java
// Getting different representations from a ByteBuffer
import java.nio.*;
import static net.mindview.util.Print.*;
```

```
public class GetData {
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        // Allocation automatically zeroes the ByteBuffer:
        int i = 0;
        while(i++ < bb.limit())
            if(bb.get() != 0)
                print("nonzero");
        print("i = " + i);
        bb.rewind();
        // Store and read a char array:
        bb.asCharBuffer().put("Howdy!");
        char c;
        while((c = bb.getChar()) != 0)
            printnb(c + " ");
        print();
        bb.rewind();
        // Store and read a short:
        bb.asShortBuffer().put((short)471142);
        print(bb.getShort());
        bb.rewind();
        // Store and read an int:
        bb.asIntBuffer().put(99471142);
        print(bb.getInt());
        bb.rewind();
        // Store and read a long:
        bb.asLongBuffer().put(99471142);
        print(bb.getLong());
        bb.rewind();
        // Store and read a float:
        bb.asFloatBuffer().put(99471142);
        print(bb.getFloat());
        bb.rewind();
        // Store and read a double:
        bb.asDoubleBuffer().put(99471142);
        print(bb.getDouble());
        bb.rewind();
    }
} /* Output:
      i = 1025
      H o w d y !
      12390
      99471142
      99471142
      9.9471144E7
      9.9471142E7
   *///:~
```

Nach dem Allokieren des Puffers wird sein Inhalt Byte für Byte geprüft, um zu verifizieren, daß daß beim Allokieren alle Bytes auf Null gesetzt werden. Alle 1024 Bytes werden überprüft (bis zum „Limit“ des Buffers; siehe Unterabschnitt 19.10.5) und zurückgesetzt.

[119] Werte primitiven Typs lassen sich am einfachsten in einen Puffer vom Typ `ByteBuffer` eintragen, indem Sie per `asCharBuffer()`, `asShortBuffer()` und so weiter den passen „Darstellungsmodus“ (*view buffer*) des Puffers wählen und dessen `put()`-Methoden aufrufen. Im obigen Beispiel wird diese Vorgehensweise für jeden primitiven Typ einmal angewandt. Die einzige Ausnahme ist

die `put()`-Methode von `ShortBuffer`, die eine Typumwandlung benötigt (beachten Sie, daß der ursprüngliche Wert durch die Typumwandlung abgeschnitten und verändert wird). Bei den übrigen Darstellungsmodi ist keine Typumwandlung beim Argument der `put()`-Methode erforderlich.

19.10.3 Verschiedene Darstellungsmodi für ByteBuffer

[120] Ein Darstellungsmodus (*view buffer*) gestattet einen Blick auf den unterliegenden `ByteBuffer`-Puffer aus der Perspektive eines primitiven Typs. „Hinter" dem Darstellungsmodus steht nach wie vor ein Puffer vom Typ `ByteBuffer` als eigentlicher Speicher und jede Änderung in einem Darstellungsmodus vollzieht sich an den tatsächlichen Daten im Puffer. Das Beispiel im vorigen Unterabschnitt zeigt, daß Sie mit Hilfe der verschiedenen Darstellungsmodi mühelos Werte primitiven Typs in einen Puffer schreiben können. Darstellungsmodi gestatten auch die Entnahme eines (wie bei `ByteBuffer` selbst) oder mehrerer Werte primitiven Typs (als Array) aus einem Puffer. Das folgende Beispiel schreibt, ändert und liest `int`-Werte in einem Puffer vom Typ `ByteBuffer` über den Darstellungsmodus `IntBuffer`:

```
//: io/IntBufferDemo.java
// Manipulating ints in a ByteBuffer with an IntBuffer
import java.nio.*;

public class IntBufferDemo {
    private static final int BSIZE = 1024;
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.allocate(BSIZE);
        IntBuffer ib = bb.asIntBuffer();
        // Store an array of int:
        ib.put(new int[] { 11, 42, 47, 99, 143, 811, 1016 });
        // Absolute location read and write:
        System.out.println(ib.get(3));
        ib.put(3, 1811);
        // Setting a new limit before rewinding the buffer.
        ib.flip();
        while(ib.hasRemaining()) {
            int i = ib.get();
            System.out.println(i);
        }
    }
} /* Output:
      99
      11
      42
      47
      1811
      143
      811
      1016
   *///:~
```

Beim ersten Aufruf der überladenen `put()`-Methode wird ein `int`-Array im Puffer gespeichert. Die folgenden `get()`- und `put()`-Aufrufe greifen direkt auf die Positionen der `int`-Elemente im unterliegenden `ByteBuffer`-Puffer zu. Beachten Sie, daß Sie auch bei einem Puffer vom Typ `ByteBuffer` direkt auf die Positionen von Elementen primitiven Typs zugreifen können.

[121] Nachdem der unterliegende `ByteBuffer`-Puffer über den `int`-Darstellungsmodus mit `int`-Werten (beziehungsweise mit Werten eines anderen primitiven Typs über den entsprechenden Dar-

stellungsmodus) gefüllt ist, kann der Puffer direkt an den Kanal übergeben werden. Das Lesen eines `ByteBuffer`-Puffers aus dem Kanal und die Entnahme von Werten primitiven Typs ist genauso einfach. Das nächste Beispiel interpretiert eine Folge von Bytes mit Hilfe verschiedener Darstellungsmodi als `short`-, `int`-, `float`-, `long`- und `double`-Werte:

```
//: io/ViewBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class ViewBuffers {
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(new byte[] { 0, 0, 0, 0, 0, 0, 0, 'a' });
        bb.rewind();
        printnb("Byte Buffer ");
        while(bb.hasRemaining())
            printnb(bb.position()+ " -> " + bb.get() + ", ");
        print();
        CharBuffer cb = ((ByteBuffer) bb.rewind()).asCharBuffer();
        printnb("Char Buffer ");
        while(cb.hasRemaining())
            printnb(cb.position() + " -> " + cb.get() + ", ");
        print();
        FloatBuffer fb = ((ByteBuffer) bb.rewind()).asFloatBuffer();
        printnb("Float Buffer ");
        while(fb.hasRemaining())
            printnb(fb.position()+ " -> " + fb.get() + ", ");
        print();
        IntBuffer ib = ((ByteBuffer) bb.rewind()).asIntBuffer();
        printnb("Int Buffer ");
        while(ib.hasRemaining())
            printnb(ib.position()+ " -> " + ib.get() + ", ");
        print();
        LongBuffer lb = ((ByteBuffer) bb.rewind()).asLongBuffer();
        printnb("Long Buffer ");
        while(lb.hasRemaining())
            printnb(lb.position()+ " -> " + lb.get() + ", ");
        print();
        ShortBuffer sb = ((ByteBuffer) bb.rewind()).asShortBuffer();
        printnb("Short Buffer ");
        while(sb.hasRemaining())
            printnb(sb.position()+ " -> " + sb.get() + ", ");
        print();
        DoubleBuffer db = ((ByteBuffer) bb.rewind()).asDoubleBuffer();
        printnb("Double Buffer ");
        while(db.hasRemaining())
            printnb(db.position()+ " -> " + db.get() + ", ");
    }
} /* Output:
      Byte Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 0, 4 -> 0, 5 -> 0, 6 -> 0, 7 -> 97,
      Char Buffer 0 -> , 1 -> , 2 -> , 3 -> a,
      Float Buffer 0 -> 0.0, 1 -> 1.36E-43,
      Int Buffer 0 -> 0, 1 -> 97,
      Long Buffer 0 -> 97,
      Short Buffer 0 -> 0, 1 -> 0, 2 -> 0, 3 -> 97,
      Double Buffer 0 -> 4.8E-322,
   *///:~
```

Der `ByteBuffer`-Puffer wird durch Verpacken (`wrap()`) eines acht Byte langen Arrays erzeugt. Der Inhalt wird mit Hilfe des entsprechenden Darstellungsmodus' als Wert der verschiedenen primitiven Typen interpretiert und ausgegeben. Abbildung 19.1 veranschaulicht die vom Darstellungsmodus abhängige Interpretation des Pufferinhaltes. Vergleichen Sie die Abbildung mit der Ausgabe des Programms.

Übungsaufgabe 24: (1) Ändern *IntBufferDemo.java* so, daß das Programm `double`-Werte und den Darstellungsmodus `DoubleBuffer` verwendet. ■

19.10.3.1 Bytereihenfolge

[122] Verschiedene Rechner verwenden beim Speichern von Daten unterschiedliche Bytereihenfolgen. Das **Big-Endian-Format** platziert das Byte mit den höchstwertigen Bits an der kleinsten, das **Little-Endian-Format** dagegen an der größten Speicheradresse. Beim Speichern eines Wertes der größer als ein Byte ist, müssen Sie eventuell die Bytereihenfolge beachten. Ein Puffer vom Typ `ByteBuffer` speichert Daten im Big-Endian-Format. Über ein Netzwerk gesendete Daten sind stets Big-Endian-formatiert. Die Bytereihenfolge eines Puffers vom Typ `ByteBuffer` kann mit Hilfe der `order()`-Methode geändert werden. Die Methode erwartet entweder `ByteOrder.BIG_ENDIAN` oder `ByteOrder.LITTLE_ENDIAN` als Argument.

[123] Angenommen, ein Puffer vom Typ `ByteBuffer` enthält die beiden Bytes 00000000 und 01100001. Wenn Sie diese Daten über den Darstellungsmodus `ShortBuffer` als `short`-Wert einlesen, erhalten Sie die Zahl 97 (00000000 01100001). Wenn Sie die Bytereihenfolge auf `LITTLE_ENDIAN` umstellen, bekommen Sie dagegen 24832 (011000001 00000000).

[124] Das folgende Beispiel zeigt, wie sich die Bytereihenfolge bei `char`-Werten je nach `BIG_ENDIAN` beziehungsweise `LITTLE_ENDIAN` ändert:

```
//: io/Endians.java
// Endian differences and data storage.
import java.nio.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Endians {
    public static void main(String[] args) {
        ByteBuffer bb = ByteBuffer.wrap(new byte[12]);
```

```
+-----+-----+-----+-----+-----+-----+-----+------+
|  0  |  0  |  0  |  0  |  0  |  0  |  0  |  97  |   byte
+-----+-----+-----+-----+-----+-----+-----+------+
|           |           |           |     a      |   char
+-----------+-----------+-----------+------------+
|     0     |     0     |     0     |    97      |   short
+-----------+-----------+-----------+------------+
|           0           |           97           |   int
+-----------------------+-----------------------+
|          0.0          |       1.36e-43        |   float
+-----------------------+-----------------------+
|                      97                       |   long
+-----------------------------------------------+
|                  4.8e-322                     |   double
+-----------------------------------------------+
```

Abbildung 19.1: ...

```
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.BIG_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
        bb.rewind();
        bb.order(ByteOrder.LITTLE_ENDIAN);
        bb.asCharBuffer().put("abcdef");
        print(Arrays.toString(bb.array()));
    }
} /* Output:
      [0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
      [0, 97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102]
      [97, 0, 98, 0, 99, 0, 100, 0, 101, 0, 102, 0]
   *///:~
```

Ein Puffer vom Typ `ByteBuffer` verwendet ein `char`-Array als externen Speicherplatz, ist also groß genug, um alle Bytes aufzunehmen. Insbesondere kann die `array()`-Methode aufgerufen werden, um die unterliegenden Bytes zu erreichen. Die `array()`-Methode ist „optional“, das heißt sie steht nur dann zur Verfügung, wenn der Puffer ein externes Array als Speicherplatz verwendet. Andernfalls wird eine Ausnahme vom Typ `java.lang.UnsupportedOperationException` ausgeworfen.

[125] Das `char`-Array wird über den Darstellungsmodus `CharBuffer` in den `ByteBuffer`-Puffer eingetragen. Die Ausgabe der unterliegenden Bytes zeigt, daß die voreingestellte Bytereihenfolge mit der anschließenden Big-Endian-Formatierung übereinstimmt, während die Little-Endian-Formatierung die Anordnung der Bytes umkehrt.

19.10.4 Data manipulation with buffers

[126] Die Abbildung auf Seite 961 im Buch veranschaulicht die Beziehungen zwischen den Klassen des `java.nio`-Packages, so daß Sie ablesen können, wie Daten verschoben oder umgewandelt werden können. Wenn Sie beispielsweise ein `byte`-Array in eine Datei schreiben müssen, verpacken Sie es zunächst mittels `ByteBuffer.wrap()`, öffnen per `getChannel()`-Methode einen Ausgabekanal (`FileOutputStream`-Objekt) und übergeben den Puffer dem `FileChannel`-Objekt.

[127] Beachten Sie, daß ein Puffer vom Typ `ByteBuffer` die einzige Möglichkeit ist, um Daten an einen Kanal zu übergeben oder von dort zu erhalten und daß die `as`-Methoden des `ByteBuffer`-Objektes die einzige Möglichkeit sind, einen separaten Puffer für Werte primitiven Typs zu erzeugen. That is, you cannot convertn a primitive-typed buffer to a ByteBuffer. Andererseits können Sie per Darstellungsmodus Werte eines beliebigen primitiven Typ in einem Puffer vom Typ `ByteBuffer` speichern oder von dort entnehmen ,,, this ist not really a restriction.

19.10.5 Die vier Zeiger mark, position, limit und capacity

[128] Ein Puffer vom Typ `Buffer` besteht aus Daten und vier Zeigern, um diese Daten erreichen und bearbeiten zu können. Die vier Zeiger sind: `mark`, `position`, `limit` und `capacity`. Tabelle 19.7 zeigt die Methoden der Klassen `Buffer`, mit denen diese Zeiger abgefragt, geändert oder zurückgesetzt werden können. Die Abfrage- und Änderungsmethoden aktualisieren die Zeiger nach ihrer Ausführung.

[129] Das nächste Beispiel wendet einen sehr einfachen Algorithmus an (Vertauschen benachbarter Zeichen), um die Zeichen in einem Puffer vom Typ `CharBuffer` zu „verschlüsseln“ beziehungsweise

zu „entschlüsseln“:

```
//: io/UsingBuffers.java
import java.nio.*;
import static net.mindview.util.Print.*;

public class UsingBuffers {
    private static void symmetricScramble(CharBuffer buffer){
        while(buffer.hasRemaining()) {
            buffer.mark();
            char c1 = buffer.get();
            char c2 = buffer.get();
            buffer.reset();
            buffer.put(c2).put(c1);
        }
    }
    public static void main(String[] args) {
        char[] data = ''UsingBuffers''.toCharArray();
        ByteBuffer bb = ByteBuffer.allocate(data.length * 2);
        CharBuffer cb = bb.asCharBuffer();
        cb.put(data);
        print(cb.rewind());
        symmetricScramble(cb);
        print(cb.rewind());
        symmetricScramble(cb);
        print(cb.rewind());
    }
} /* Output:
      UsingBuffers
      sUniBgfuefsr
      UsingBuffers
   *///:~
```

Obwohl Sie per `wrap()`-Methode mit einem `char`-Array als Argument direkt einen Puffer vom Typ `CharBuffer` erzeugen könnten, allokieren wir einen Puffer vom Typ `ByteBuffer` und fordern einen `CharBuffer` als „View“ auf den `ByteBuffer`-Puffer an. Diese Vorgehensweise betont, daß die Operationen immer auf einem Puffer vom Typ `ByteBuffer` durchgeführt werden, da nur Puffer dieses Typs mit einem Kanal interagieren können.

[130] Beim erstmaligen Eintreten in die `symmetricScramble()`-Methode hat der Puffer den folgenden Zustand:

```
                                        cap
                                         +
| U | s | i | n | g | B | u | f | f | e | r | s |
  +                                      +
 pos                                    lim
```

Der `position`-Zeiger verweist auf das erste Element des Puffers, die Zeiger `capacity` und `limit` auf das letzte.

[131] Die `while`-Schleife in der `symmetricScramble()`-Methode wird so lange ausgeführt, bis `position` und `limit` übereinstimmen. Der `position`-Zeiger des Puffers ändert sich beim Aufrufen einer der relativen `get()`- beziehungsweise `put()`-Methoden auf „seinem“ Element. Die absoluten `get()`- beziehungsweise `put()`-Methoden erwarten einen Index als Argument, der das Element festlegt, auf dem die Methoden aufgerufen wird. Die absoluten Methoden beeinflussen den `position`-Zeiger des Puffers nicht.

[132] Beim Eintritt in die `while`-Schleife wird der `mark`-Zeiger mit Hilfe der `mark()`-Methode gesetzt. Der Zustand des Puffers ist nun:

```
 mar                                     cap
  +                                       +
| U | s | i | n | g | B | u | f | f | e | r | s |
  +                                       +
 pos                                     lim
```

Die beiden relativen `get()`-Aufrufe speichern die Werte der ersten beiden Zeichen in den Variablen `c1` und `c2`. Nach diesen beiden Operationen befindet sich der Puffer im folgenden Zustand:

```
 mar                                     cap
  +                                       +
| U | s | i | n | g | B | u | f | f | e | r | s |
          +                               +
         pos                             lim
```

Damit die Zeichen vertauscht werden, müssen wir `c2` an Position 0 und `c1` an Position 1 deponieren. Wir können dazu entweder die absolute `put()`-Methode verwenden, oder den `position`-Zeiger per `reset()`-Methode auf den Wert des `mark`-Zeigers zurücksetzen:

```
 mar                                     cap
  +                                       +
| U | s | i | n | g | B | u | f | f | e | r | s |
  +                                       +
 pos                                     lim
```

Die beiden `put()`-Aufrufe schreiben zuerst `c2` und dann `c1`:

```
 mar                                     cap
  +                                       +
| s | U | i | n | g | B | u | f | f | e | r | s |
```

Buffer-Methode	Funktion
`capacity()`	Gibt den Inhalt des `capacity`-Feldes des Puffers zurück.
`clear()`	Leert den Puffer, setzt das `position`-Feld auf Null und das `limit`-Feld) auf den Wert des `capacity`-Feldes. Sie rufen die `clear()`-Methode auf, um einen vorhandenen Puffer zu überschreiben.
`flip()`	Setzt das `limit`-Feld) auf den Wert des `position`-Feldes und den Wert des `position`-Feldes auf Null. Die `flip()`-Methode wird aufgerufen, um den Puffer nach der Übergabe von Daten zur Entnahme vorzubereiten.
`limit()`	Gibt den Inhalt des `limit`-Feldes des Puffers zurück.
`limit(int lim)`	Bewertet das `limit`-Feld des Puffers.
`mark()`	Bewertet das `mark`-Feld mit dem Inhalt des `position`-Feldes.
`position()`	Gibt den Inhalt des `position`-Feldes des Puffers zurück.
`position(int pos)`	Bewertet das `position`-Feld des Puffers.
`remaining()`	Gibt die Differenz zwischen den Inhalten der Felder `limit` und `position` des Puffers zurück.
`hasRemaining()`	Gibt `true` zurück, wenn sich noch Elemente zwischen `position` und `limit` befinden.

Tabelle 19.7: Abfrage- und Änderungsmethoden der vier Felder `mark`, `position`, `limit` und `capacity`.

```
            +                                   +
           pos                                 lim
```

Zu Beginn des nächsten Schleifendurchlaufs wird der `mark`-Zeiger auf den aktuellen Wert des `position`-Zeigers gesetzt:

```
            mar                                 cap
             +                                   +
| s | U | i | n | g | B | u | f | f | e | r | s |
             +                                   +
            pos                                 lim
```

[133] Der Prozeß schreitet fort, bis das Ende des Puffers erreicht ist. Nach der letzten Iteration verweist der `position`-Zeiger auf das Ende des Puffers. Wenn Sie den Pufferinhalt nun ausgeben, werden nur die Zeichen zwischen `position` und `limit` angezeigt. Sie müssen den `position`-Zeiger mittels `rewind()` an den Anfang des Puffers zurücksetzen, um den gesamten Inhalt auszugeben. Nach dem Aufrufen der `rewind()`-Methode befindet sich der Puffer im Zustand (der `mark`-Zeiger ist nicht definiert):

```
            mar                                 cap
             +                                   +
| s | U | i | n | g | B | u | f | f | e | r | s |
             +                                   +
            pos                                 lim
```

Wird die Methode `symmetricScramble()` erneut aufgerufen, so wird der Pufferinhalt noch einmal demselben Prozeß unterzogen und sein ursprünglicher Zustand wieder hergestellt.

19.10.6 Abbildung von Dateien in den Arbeitsspeicher

[134] Das Abbilden von Dateien in den Arbeitsspeicher gestattet das Erzeugen und Bearbeiten von Dateien, die für den Speicher zu groß sind. Durch die Abbildung einer Datei in den Arbeitsspeicher läßt sich vortäuschen, daß sich die gesamte Datei im Speicher befindet. Die Datei läßt sich wie ein sehr großes Array behandeln. Dieser Ansatz vereinfacht die Anweisungen zum Ändern einer Datei enorm. Ein kleines Beispiel:

```
//: io/LargeMappedFiles.java
// Creating a very large file using mapping.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class LargeMappedFiles {
    static int length = 0x8FFFFFF; // 128 MB
    public static void main(String[] args) throws Exception {
        MappedByteBuffer out =
            new RandomAccessFile(''test.dat'', ''rw'').getChannel()
                .map(FileChannel.MapMode.READ_WRITE, 0, length);
        for(int i = 0; i < length; i++)
            out.put((byte)'x');
        print(''Finished writing'');
        for(int i = length/2; i < length/2 + 6; i++)
            printnb((char)out.get(i));
    }
} ///:~
```

Wir wählen ein `RandomAccessFile`-Objekt, um sowohl Lese- als auch Schreibzugriff zu erhalten, fordern einen Kanal für die Datei an und rufen auf dem erhaltenen `FileChannel`-Objekt die `map()`-Methode auf, um ein `java.nio.MappedByteBuffer`-Objekt zu erzeugen (eine besondere Art von „direktem“ Puffer). Beachten Sie, daß Sie die Startposition und Länge des Abschnitts in der Datei angeben müssen, der in den Arbeitsspeicher abgebildet werden soll (Sie haben hierbei die Möglichkeit, nur einen kleinen Teil einer großen Datei zu wählen).

[135] Die Klasse `MappedByteBuffer` ist von `ByteBuffer` abgeleitet, verfügt also über alle `ByteBuffer`-Methoden. Das obige Beispiel zeigt nur einfache Anwendungen von `put()` und `get()`. Sie können aber auch die übrigen Methoden verwenden, zum Beispiel `asCharBuffer()`.

[136] Die mit dem obigen Beispiel erzeugte Datei ist 128 MB groß und damit wahrscheinlich größer, als Ihr Betriebssystem für eine einzelne Datei im Arbeitsspeicher erlaubt. Die Datei steht dennoch scheinbar komplett zur Verfügung, da sich stets ein Teil im Speicher befindet, während der Rest ausgelagert wird. Auf diese Weise können sehr große Dateien (bis zu zwei GB) mühelos bearbeitet werden. Die Abbildung von Dateien in den Arbeitsspeicher basiert auf den entsprechenden Möglichkeiten des unterliegenden Betriebssystems, um bestmögliche Performanz zu gewährleisten.

19.10.6.1 Performanz

[137] Obwohl die Performanz der traditionellen Ein-/Ausgabebibliothek durch die auf dem `java.nio`-Package basierende Reimplementierung verbessert wurde, ist der Zugriff auf in den Arbeitsspeicher abgebildete Dateien erheblich schneller. Das folgende Programm führt einen einfachen Geschwindigkeitsvergleich durch:

```
//: io/MappedIO.java
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class MappedIO {
    private static int numOfInts = 4000000;
    private static int numOfUbuffInts = 200000;
    private abstract static class Tester {
        private String name;
        public Tester(String name) { this.name = name; }
        public void runTest() {
            System.out.print(name + ": ");
            try {
                long start = System.nanoTime();
                test();
                double duration = System.nanoTime() - start;
                System.out.format("%.2f\n", duration/1.0e9);
            } catch(IOException e) {
                throw new RuntimeException(e);
            }
        }
        public abstract void test() throws IOException;
    }
    private static Tester[] tests = {
        new Tester("Stream Write") {
            public void test() throws IOException {
                DataOutputStream dos = new DataOutputStream(
                    new BufferedOutputStream(
                        new FileOutputStream(new File("temp.tmp"))));
```

```
            for(int i = 0; i < numOfInts; i++)
                dos.writeInt(i);
            dos.close();
        }
    },
    new Tester("Mapped Write") {
        public void test() throws IOException {
            FileChannel fc =
                new RandomAccessFile("temp.tmp", "rw").getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_WRITE, 0, fc.size()).asIntBuffer();
            for(int i = 0; i < numOfInts; i++)
                ib.put(i);
            fc.close();
        }
    },
    new Tester("Stream Read") {
        public void test() throws IOException {
            DataInputStream dis = new DataInputStream(
                new BufferedInputStream(
                    new FileInputStream("temp.tmp")));
            for(int i = 0; i < numOfInts; i++)
                dis.readInt();
            dis.close();
        }
    },
    new Tester("Mapped Read") {
        public void test() throws IOException {
            FileChannel fc = new FileInputStream(
                new File("temp.tmp")).getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_ONLY, 0, fc.size()).asIntBuffer();
            while(ib.hasRemaining())
                ib.get();
            fc.close();
        }
    },
    new Tester("Stream Read/Write") {
        public void test() throws IOException {
            RandomAccessFile raf = new RandomAccessFile(
                new File("temp.tmp"), "rw");
            raf.writeInt(1);
            for(int i = 0; i < numOfUbuffInts; i++) {
                raf.seek(raf.length() - 4);
                raf.writeInt(raf.readInt());
            }
            raf.close();
        }
    },
    new Tester("Mapped Read/Write") {
        public void test() throws IOException {
            FileChannel fc = new RandomAccessFile(
                new File("temp.tmp"), "rw").getChannel();
            IntBuffer ib = fc.map(
                FileChannel.MapMode.READ_WRITE, 0, fc.size()).asIntBuffer();
            ib.put(0);
            for(int i = 1; i < numOfUbuffInts; i++)
```

```
                    ib.put(ib.get(i - 1));
                fc.close();
            }
        }
    };
    public static void main(String[] args) {
        for(Tester test : tests)
            test.runTest();
    }
} /* Output: (90% match)
      Stream Write: 0.56
      Mapped Write: 0.12
      Stream Read: 0.80
      Mapped Read: 0.07
      Stream Read/Write: 5.32
      Mapped Read/Write: 0.02
   *///:~
```

Wir haben in diesem Buch bereits mehrere Anwendungen des *Templatemethod*-Entwurfsmusters besprochen. Die Schablonenmethode, hier `runTest()`, ist der „Algorithmus“ eines Testverfahrens, welcher verschiedene Implementierungen der `test()`-Methode („Einschubmethode“) in mehreren anonymen inneren Klassen aufruft. Jede dieser Klassen führt einen Test aus, das heißt die `test()`-Methoden sind Prototypen der verschiedenen Ein-/Ausgabeoperationen.

[138] *Although a mapped write would seem to use a FileOutputStream, all output in file mapping must use a RandomAccessFile, just as read/write does in the preceding code.* Beachten Sie, daß die zur Initialisierung der verschiedenen Ein-/Ausgabeoperationen benötigte Zeit mitgemessen wird. Obwohl das Aufsetzen einer in den Arbeitsspeicher abgebildeten Datei teuer sein kann, ist der im Vergleich mit traditioneller Ein-/Ausgabe erhaltene Geschwindigkeitszuwachs signifikant.

Übungsaufgabe 25: (6) Ersetzen Sie in den Beispielen dieses Abschnitts die `ByteBuffer`-Methode `allocate()` durch `allocateDirect()`. Zeigen Sie die Geschwindigkeitsunterschiede auf und beachten Sie, ob sich die zum Programmstart erforderliche Zeit merkbar ändert. ■

Übungsaufgabe 26: (3) Ändern Sie das Beispiel *strings/JGrep.java* so, daß das Programm eine in den Arbeitsspeicher abgebildete Datei verwendet. ■

19.10.7 Dateisperren

[139] Das Sperren von Dateien gestattet Ihnen, eine Datei wie eine gemeinsam genutzte Resource zu behandeln und den Zugriff zu synchronisieren („zeitlich aufeinander abzustimmen“). Zwei um eine Resource wetteifernde Threads können allerdings verschiedenen Laufzeitumgebungen angehören oder einer kann ein Java-Thread, der andere aber ein nativer Thread des Betriebssystems sein. Dateisperren sind aber für Prozesse des Betriebssystems sichtbar, da der Java-Sperrmechanismus direkt auf den Sperrmechanismus des unterliegenden Betriebssystems abgebildet ist.

[140] Ein einfaches Beispiel für das Sperren einer Datei:

```
//: io/FileLocking.java
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;

public class FileLocking {
    public static void main(String[] args) throws Exception {
        FileOutputStream fos = new FileOutputStream("file.txt");
```

```
            FileLock fl = fos.getChannel().tryLock();
            if(fl != null) {
                System.out.println("Locked File");
                TimeUnit.MILLISECONDS.sleep(100);
                fl.release();
                System.out.println("Released Lock");
            }
            fos.close();
        }
    } /* Output:
          Locked File
          Released Lock
       *///:~
```

Die argumentlosen `java.nio.channels.FileChannel`-Methoden `tryLock()` und `lock()` liefern ein `java.nio.channels.FileLock`-Objekt für die ganze Datei. (Bei den `java.nio.channels`-Klassen `SocketChannel`, `DatagramChannel` und `ServerSocketChannel` ist keine Sperre erforderlich, da ihre Objekte von Natur aus von höchstens einem Prozeß beansprucht werden. In der Regel teilen sich keine zwei Prozesse einen Netzwerksocket.) Die `tryLock()`-Methode blockiert nicht, sondern versucht, die Sperre an sich zu bringen und kehrt bei Mißerfolg (wenn ein anderer Prozeß die Sperre beansprucht und die Sperre nicht teilbar ist) einfach zurück. Die `lock()`-Methode blockiert dagegen, bis die Sperre verfügbar ist, der Thread, der die `lock()`-Methode aufgerufen hat, unterbrochen wird oder der Kanal (das `FileChannel`-Objekt) geschlossen wird, zu dem die aufgerufene `lock()`-Methode gehört. Eine Sperre wird mit Hilfe der `FileLock`-Methode `release()` wieder aufgehoben.

[141] Die andere (dreiargumentige) Version der `tryLock()`-Methode sperrt einen *Abschnitt fester Größe* der Datei:

```
tryLock(long position, long size, boolean shared)
```

Auch die `lock()`-Methode hat eine weitere (ebenfalls dreiargumentige) Version, die einen Abschnitt (ebenfalls) fester Größe der Datei sperrt:

```
lock(long position, long size, boolean shared)
```

In beiden Fällen gibt das dritte Argument an, ob die Sperre teilbar ist.

[142] Die von den argumentlosen Methoden `tryLock()` und `lock()` bewirkte Sperre paßt sich bei Änderung der Dateigröße an, nicht aber die von den argumentbehafteten Versionen induzierte Sperre. Bezieht sich eine Sperre auf einen durch `position` und `position+size` definierten Abschnitt fester Größe und die Datei wächst über das obere Ende dieses Abschnitts hinaus, so wird der überstehende Teil der Datei *nicht gesperrt*. Die von den argumentlosen Versionen bewirkte Sperre erstreckt sich stets auf die gesamte Datei und wächst erforderlichenfalls mit.

[143] Die Unterstützung exklusiver oder teilbarer Sperren hängt vom unterliegenden Betriebssystem ab. Unterstützt das Betriebssystem teilbare Sperren nicht, so wird gegebenenfalls stattdessen eine exklusive Sperre vergeben. Der Sperrtyp (exklusiv oder teilbar) kann mit Hilfe der `FileLock`-Methode `isShared()` abgefragt werden.

19.10.7.1 Abschnittsweise Sperrung einer in den Arbeitsspeicher abgebildeten Datei

[144] Das Abbilden von Dateien in den Arbeitsspeicher wird typischerweise bei sehr großen Dateien genutzt. Eventuell möchten Sie einige Abschnitte solcher großen Dateien sperren, während die nicht gesperrten Teile von andereren Prozessen modifiziert werden können. Diese Situation tritt bei Datenbanken auf, die stets für viele Benutzer verfügbar sein müssen.

[145] Im folgenden Beispiel sperren zwei Threads unterschiedliche Abschnitte einer Datei:

```
//: io/LockingMappedFiles.java
// Locking portions of a mapped file.
// {RunByHand}
import java.nio.*;
import java.nio.channels.*;
import java.io.*;

public class LockingMappedFiles {
    static final int LENGTH = 0x8FFFFFF; // 128 MB
    static FileChannel fc;
    public static void main(String[] args) throws Exception {
        fc = new RandomAccessFile("test.dat", "rw").getChannel();
        MappedByteBuffer out =
            fc.map(FileChannel.MapMode.READ_WRITE, 0, LENGTH);
        for(int i = 0; i < LENGTH; i++)
            out.put((byte)'x');
        new LockAndModify(out, 0, 0 + LENGTH/3);
        new LockAndModify(out, LENGTH/2, LENGTH/2 + LENGTH/4);
    }
    private static class LockAndModify extends Thread {
        private ByteBuffer buff;
        private int start, end;
        LockAndModify(ByteBuffer mbb, int start, int end) {
            this.start = start;
            this.end = end;
            mbb.limit(end);
            mbb.position(start);
            buff = mbb.slice();
            start();
        }
        public void run() {
            try {
                // Exclusive lock with no overlap:
                FileLock fl = fc.lock(start, end, false);
                System.out.println("Locked: "+ start +" to "+ end);
                // Perform modification:
                while(buff.position() < buff.limit() - 1)
                    buff.put((byte)(buff.get() + 1));
                fl.release();
                System.out.println("Released: "+start+" to "+ end);
            } catch(IOException e) {
                throw new RuntimeException(e);
            }
        }
    }
} ///:~
```

Die innere Thread-Klasse `LockAndModify` definiert einen Abschnitt des Puffers und erzeugt a `slice()` to be modified. Die `run()`-Methode sperrt das `FileChannel`-Objekt (der Puffer selbst kann nicht gesperrt werden, nur der Kanal). Der `lock()`-Aufruf ähnelt dem Sperren eines Objektes für Threads. Nach erfolgter Sperre erhalten Sie einen „kritischen Bereich“ mit exklusivem Zugriff im festgelegten Abschnitt der Datei (Kapitel 22 behandelt Threads in allen Einzelheiten).

[146] Die Sperre wird automatisch aufgehoben, wenn die Laufzeitumgebung beendet oder der Kanal zu dem die Sperre gehört geschlossen wird. Alternativ können Sie die `release()`-Methode des `FileLock`-Objektes explizit aufrufen (siehe oben).

19.11 Kompression

[147] Die Ein-/Ausgabebibliothek von Java enthält auch Klassen, die Datenströme in komprimierter Form lesen und schreiben können. Die anderen Ein-/Ausgabeklassen werden in einer solchen Kompressionsklasse verpackt, um die Kompressionsfunktionalität zu nutzen.

[148] Die Kompressionsklassen gehören nicht zu den zeichenorientierten Hierarchien unter den Basisklassen `Reader` und `Writer`, sondern zu den byteorientierten Hierarchien unter `InputStream` und `OutputStream`, da die Kompressionsbibliothek Bytes anstelle von Zeichen verarbeitet. Gelegentlich müssen Sie allerdings zeichen- und byteorientierte Datenströme kombinieren. (Denken Sie dabei an die Konvertierungsklassen `InputStreamReader` und `OutputStreamWriter`.) Unter den vielen Kompressionsverfahren werden Zip und GZIP wohl am häufigsten verwendet. Sie können Ihre komprimierten Daten also mit vielen Lese- und Schreibwerkzeugen für diese Formate bearbeiten.

19.11.1 Komprimieren einer einzelnen Datei per GZIP

[149] Die Schnittstellen der beiden Klassen `java.util.zip.GZIPInputStream` und `java.util.zip.GZIPOutputStream` sind einfach und daher angebracht, wenn Sie einen einzelnen Datenstrom komprimieren möchten (im Gegensatz zu mehreren verschiedenartigen Datenströmen). Das folgende Beispiel komprimiert eine einzelne Datei:

```
//: io/GZIPcompress.java
// {Args: GZIPcompress.java}
import java.util.zip.*;
import java.io.*;

public class GZIPcompress {
    public static void main(String[] args) throws IOException {
        if (args.length == 0) {
            System.out.println(
                ''Usage: \nGZIPcompress file\n'' +
                ''\tUses GZIP compression to compress '' +
                ''the file to test.gz'');
            System.exit(1);
        }
        BufferedReader in = new BufferedReader(
            new FileReader(args[0]));
        BufferedOutputStream out = new BufferedOutputStream(
            new GZIPOutputStream(
                new FileOutputStream(''test.gz''))));
        System.out.println(''Writing file'');
        int c;
        while((c = in.read()) != -1)
            out.write(c);
        in.close();
        out.close();
        System.out.println(''Reading file'');
        BufferedReader in2 = new BufferedReader(
            new InputStreamReader(new GZIPInputStream(
                new FileInputStream(''test.gz''))));
        String s;
        while((s = in2.readLine()) != null)
            System.out.println(s);
    }
} /* (Execute to see output) *////:~
```

Die Anwendung der Kompressionsklassen ist nicht schwer. Es genügt, den Ausgabestrom in einem `GZIPOutputStream`- oder `ZipOutputStream`-Objekt und den Eingabestrom in einem `GZIPInputStream`- oder `GZIPInputStream`-Objekt zu verpacken. Der Rest sind gewöhnliche Klassen zum Lesen beziehungsweise Schreiben aus der traditionellen Ein-/Ausgabebibliothek. *GZIPcompress.java* ist insbesondere ein Beispiel für die Kombination eines zeichenorientierten mit einem byteorientierten Datenstrom: Die lokale Variable `in` referenziert ein `Reader`-Objekt, während der Konstruktor der Klasse `GZIPOutputStream` nur ein `OutputStream`-, aber keine `Writer`-Objekte akzeptiert. Beim Öffnen der komprimierten Datei wird das `GZIPInputStream`-Objekt in ein `Reader`-Objekt umgewandelt.

19.11.2 Komprimieren vieler Dateien per Zip

[150] Der Teil der Kompressionsbibliothek, der das Zip-Format unterstützt, ist umfangreicher, gestattet viele Dateien zu archivieren und stellt sogar eine Klasse zur Verfügung, die das Einlesen einer *.zip* Datei erleichtert. Die Bibliothek unterstützt das Standard-Zip-Format, arbeitet also mit allen Zip-Werkzeugen zusammen, die im Internet zum Download zur Verfügung stehen. Das folgende Beispiel hat dieselbe Ausgabe wie das vorige (*GZIPcompress.java*), kann aber beliebig viele Kommandozeilenargumente verarbeiten. Darüber hinaus demonstriert das Programm die Verwendung der Prüfsummenklassen, um die Prüfsumme der *.zip* Datei zu berechnen und zu verifizieren:

```
//: io/ZipCompress.java
// Uses Zip compression to compress any
// number of files given on the command line.
// {Args: ZipCompress.java}
import java.util.zip.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ZipCompress {
    public static void main(String[] args) throws IOException {
        FileOutputStream f = new FileOutputStream("test.zip");
        CheckedOutputStream csum = new CheckedOutputStream(f, new Adler32());
        ZipOutputStream zos = new ZipOutputStream(csum);
        BufferedOutputStream out = new BufferedOutputStream(zos);
        zos.setComment("A test of Java Zipping");
        // No corresponding getComment(), though.
        for(String arg : args) {
            print("Writing file " + arg);
            BufferedReader in = new BufferedReader(new FileReader(arg));
            zos.putNextEntry(new ZipEntry(arg));
            int c;
            while((c = in.read()) != -1)
                out.write(c);
            in.close();
            out.flush();
        }
        out.close();
        // Checksum valid only after the file has been closed!
        print("Checksum: " + csum.getChecksum().getValue());
        // Now extract the files:
        print("Reading file");
        FileInputStream fi = new FileInputStream("test.zip");
        CheckedInputStream csumi = new CheckedInputStream(fi, new Adler32());
        ZipInputStream in2 = new ZipInputStream(csumi);
```

Kompressionsklasse	Funktion
`CheckedInputStream`	Die `getChecksum()`-Methode erzeugt die Prüfsumme eines beliebigen `InputStream`-Objektes (nicht nur Dekompression).
`CheckedOutputStream`	Die `getChecksum()`-Methode erzeugt die Prüfsumme eines beliebigen `OutputStream`-Objektes (nicht nur Dekompression).
`DeflaterOutputStream`	Basisklasse für Kompressionsklassen.
`ZipOutputStream`	Unterklasse von `DeflaterOutputStream`, komprimiert Daten ins *.zip* Dateiformat.
`GZIPOutputStream`	Unterklasse von `DeflaterOutputStream`, komprimiert Daten ins GZIP-Dateiformat.
`InflaterInputStream`	Basisklasse für Dekompressionsklassen.
`ZipInputStream`	Unterklasse von `InflaterInputStream`, dekomprimiert Daten, die zuvor im *.zip* Dateiformat gespeichert wurden.
`GZIPInputStream`	Unterklasse von `InflaterInputStream`, dekomprimiert Daten, die zuvor im GZIP-Dateiformat gespeichert wurden.

Tabelle 19.8: Kompressions- und Dekompressionsklassen im Package `java.util.zip`.

```
        BufferedInputStream bis = new BufferedInputStream(in2);
        ZipEntry ze;
        while((ze = in2.getNextEntry()) != null) {
            print("Reading file " + ze);
            int x;
            while((x = bis.read()) != -1)
                System.out.write(x);
        }
        if(args.length == 1)
            print("Checksum: " + csumi.getChecksum().getValue());
        bis.close();
        // Alternative way to open and read Zip files:
        ZipFile zf = new ZipFile("test.zip");
        Enumeration e = zf.entries();
        while(e.hasMoreElements()) {
            ZipEntry ze2 = (ZipEntry)e.nextElement();
            print("File: " + ze2);
            // ... and extract the data as before
        }
        /* if(args.length == 1) */
    }
} /* (Execute to see output) *////:~
```

Es gibt zwei Prüfsummenklassen: `java.util.zip.Adler32` (schneller) und `java.util.zip.CRC32` (langsamer, dafür aber genauer).

[151] Für jede Datei, die dem Archiv hinzugefügt werden soll, wird einmal die `putNextEntry()`-Methode mit einem `java.util.zip.ZipEntry`-Objekt als Argument aufgerufen. Das `ZipEntry`-Objekt besitzt eine umfangreiche Schnittstelle, über die Sie sämtliche Informationen über einen Archiveintrag abfragen oder ändern können: Datei- beziehungsweise Verzeichnisname, Größe im komprimierten und dekomprimierten Zustand, Datum, CRC-Prüfsumme, extra field data, Kommentar, Kompressionsverfahren sowie, ob es sich um ein Verzeichnis handelt. Das Zip-Format bein-

haltet ein Paßwort, welches aber von der Zip-Bibliothek in Java nicht unterstützt wird. Obwohl die Klassen `CheckedInputStream` und `CheckedOutputStream` beide Prüfsummen sowohl nach dem Adler32- als auch dem CRC32-Verfahren unterstützen, kennt die Klasse `ZipEntry` nur CRC32. ~~This is a restriction of the underlying Zip format~~, kann Sie aber darin hindern, das schnellere Adler32-Verfahren zu wählen.

[152] Die `ZipInputStream`-Methode `getNextEntry()` dient zum Extrahieren des nächsten Archiveintrages als `ZipEntry`-Objekt, sofern vorhanden. Enthält das Archiv keine weiteren Einträge, so gibt die Methode `null` zurück. Als kurze und bündige Alternative, können Sie die *.zip* Datei mit Hilfe eines `java.util.zip.ZipFile`-Objektes einlesen, dessen `entries()`-Methode eine Referenz auf ein *`Enumeration`*-Objekt von `ZipEntry`-Elementen zurückgibt.

[153] Das Auslesen der Prüfsumme setzt voraus, daß Sie Zugriff auf das entsprechende *`java.util.zip.Checksum`*-Objekt haben. Im obigen Beispiel werden Referenzen auf die `CheckedOutputStream`- und `CheckedInputStream`-Objekte aufbewahrt. Sie könnten aber auch einfach eine Referenz auf das *`Checksum`*-Objekt speichern.

[154] Die Klasse `ZipOutputStream` hat die rätselhafte (*baffling*) Methode `setComment()`. Wie Sie im obigen Beispiel sehen, können Sie beim Anlegen der *.zip* Datei zwar einen Kommentar speichern, aber die Klasse `ZipInputStream` verfügt über keine Methode, um den Kommentar abzufragen. Kommentare scheinen nur in der Klasse `ZipEntry` auf der Ebene der einzelnen Archiveeinträge in vollem Umfang unterstützt zu werden.

[155] Selbstverständlich sind Sie mit der Kompressionsbibliothek nicht auf Dateien eingeschränkt, sondern können alles komprimieren, inklusive Daten, die über eine Netzwerkverbindung gesendet werden.

19.11.3 Java-Archive (.jar Dateien)

[156] Das JAR-Dateiformat (Java Archive) basiert auf dem Zip-Format und gestattet, wie Zip, eine Gruppe von Dateien in einer einzelnen komprimierten Datei zu archivieren. Wie alles bei Java, sind auch *.jar* Dateien plattformunabhängig, so daß Sie sich nicht mit plattformspezifischen Fragen auseinandersetzen müssen. Eine *.jar* Datei kann neben *.class* Dateien auch Audio- und Graphikdateien enthalten.

[157] *.jar* Dateien sind im Zusammenhang mit dem Internet besonders praktisch. Vor den *.jar* Dateien mußte Ihr Webbrowser eine Reihe von Anfragen an den Webserver senden, um alle Bestandteile eines Applets herunterzuladen. Außerdem waren sämtliche Dateien unkomprimiert. Durch das Archivieren aller Bestandteile eines Applets zu einer einzigen *.jar* Datei ist nur noch eine Anfrage an den Webserver erforderlich und die Übertragung ist durch die Kompression schneller. Darüber hinaus kann jede Datei in einem *.jar* Archiv zur Sicherheit digital unterzeichnet werden.

[158] Ein *.jar* Archiv besteht aus einer einzigen Datei, welche eine Reihe von Zip-komprimierten Dateien und eine Manifest-Datei enthält. (Sie können selbst eine Manifest-Datei schreiben. Andernfalls nimmt Ihnen `jar` diese Arbeit ab.) In der Dokumentation des Java Development Kits finden Sie weiterführende Informationen über die Manifest-Dateien in *.jar* Archiven.

[159] Das in der Distribution des Java Development Kits enthaltene Hilfsprogramm `jar` komprimiert die zu archivierenden Dateien automatisch. Der Aufruf auf der Kommandozeile hat das Format:

```
jar [options] destination [manifest] inputfile(s)
```

Die Kommandozeilenschalter und -optionen sind einfach nur Buchstaben ohne führenden Bindestrich oder sonstige Kennzeichnung (siehe Tabelle 19.9). Unix/Linuxbenutzer werden die Ähnlichkeit

mit `tar` bemerken.

[160] Enthält die Liste der zu archivierenden Dateien ein Verzeichnis, so wird dieses, zusammen mit seinen Unterverzeichnissen automatisch inkludiert. Die Pfadinformation zu den Dateien in den Unterverzeichnissen bleibt dabei erhalten.

[161] Es folgen einige typische Beispiele für das `jar`-Kommando. Das erste Kommando erzeugt eine *.jar* Datei namens *myJarFile.jar*, die alle *.class* Dateien im aktuellen Verzeichnis sowie eine automatisch generierte Manifest-Datei enthält:

```
jar cf myJarFile.jar *.class
```

Das nächste Kommando entspricht dem vorigen, wobei nun eine vom Benutzer geschriebene Manifest-Datei verwendet wird:

```
jar cmf myJarFile.jar myManifestFile.mf *.class
```

Das folgende Kommando gibt das „Inhaltsverzeichnis“ der Datei *myJarFile.jar* aus:

```
jar tf myJarFile.jar
```

Der Schalter `v` bewirkt, daß das Kommando während seiner Arbeit detaillierte Meldungen über die archivierten Dateien ausgibt:

```
jar tvf myJarFile.jar
```

Sind *audio*, *classes* und *image* Verzeichnisse, so nimmt das nächste Kommando diese Verzeichnisse in die Archivdatei *myApp.jar* auf. Der Schalter `v` sorgt wiederum für ausführliche Meldungen, während das `jar`-Kommando arbeitet:

```
jar cvf myApp.jar audio classes image
```

Wenn Sie eine *.jar* Datei mit dem Schalter `o` erzeugen, können Sie sie in Ihren Klassenpfad aufnehmen:

Option	Beschreibung
`c`	Erzeugt ein neues oder leeres Archiv.
`t`	Zeigt den Archivinhalt an.
`x`	Packt das ganze Archiv aus.
`x file`	Packt die Datei *file* aus dem Archiv aus.
`f`	Teilt `jar` mit, daß es eine Archivdatei verwenden soll. Ohne die Option `f` setzt `jar` voraus, daß die Eingabe von der Standardeingabe kommt beziehungsweise an die Standardausgabe gesendet wird, wenn `jar` eine Archivdatei erzeugt.
`m`	Gibt an, daß das erste Argument eine vom Anwender erzeugt Manifest-Datei ist.
`v`	Gibt ausführliche Meldungen über die Verarbeitung des `jar`-Kommandos aus.
`o`	Archiviert die Dateien ohne Kompression (verwenden Sie diese Einstellung um eine *.jar* Datei zu erzeugen, die Sie in Ihren Klassenpfad eintragen.
`M`	Unterdrückt das automatische Erzeugen einer Manifest-Datei.

Tabelle 19.9: Kommandozeilenschalter und -optionen des `jar`-Kommandos.

```
CLASSPATH=''lib1.jar;lib2.jar;''
```

Nun sucht Java in *lib1.jar* und *lib2.jar* nach *.class* Dateien.

[162] Das `jar`-Kommando ist nicht so universell einsetzbar wie `zip`. Beispielsweise können Sie einer existierenden *.jar* Datei keine neuen Dateien hinzufügen oder archivierte Dateien aktualisieren. Sie können innerhalb einer Archivdatei keine archivierten Dateien verschieben und am ehemaligen Ort löschen. Andererseits können Sie eine auf einer beliebigen Plattform erzeugte *.jar* Datei mit dem `jar`-Kommando auf jeder beliebigen Plattform transparent lesen (dieses Problem tritt gelegentlich bei Programmen auf, die mit *.zip* Dateien arbeiten).

[163] Wie Sie in Unterabschnitt 23.11.5 lernen werden, dienen *.jar* Dateien auch zum Verpacken von JavaBeans.

19.12 Serialisierung

[164] Ein Objekt existiert während der Laufzeit eines Programms solange Sie es benötigen, nicht aber über die Laufzeit hinaus. Dies wirkt auf den ersten Blick durchaus sinnvoll, aber es gibt Situationen, in denen es nützlich wäre, wenn ein Objekt über die Laufzeit des Programms hinaus existieren und seinen Zustand erhalten könnte. In diesem Fall wäre ein solches Objekt beim nächsten Programmstart vorhanden und würde diesselbe Information enthalten, wie beim vorigen Programmaufruf. Sie bewirken selbstverständlich denselben Effekt, wenn Sie den Zustand des Objektes in eine Datei schreiben oder in einer Datenbank sichern. Aber vor dem Hintergrund, daß bei Java alles ein Objekt ist, wäre es folgerichtig, ein Objekt als „persistent“ zu deklarieren und sich nicht um die Einzelheiten kümmern zu müssen.

[165] Der Serialisierungsmechanismus von Java gestattet, jedes Objekt dessen Klasse das Interface *`java.io.Serializable`* implementiert, in eine Folge von Bytes umzuwandeln, aus der das ursprüngliche Objekt zu einem späteren Zeitpunkt rekonstruiert werden kann. Dies gilt sogar über ein Netzwerk hinweg, das heißt der Serialisierungsmechanismus gleicht die Unterschiede zwischen den beteiligten Betriebssystemen aus. Sie können also ein unter Windows serialisiertes Objekt über ein Netzwerk an einen Unixrechner senden, wo es korrekt deserialisiert wird. Sie brauchen sich nicht um die Darstellung der Daten bezüglich der jeweiligen Plattform, die Bytereihenfolge und andere Einzelheiten zu kümmern.

[166] Die Serialisierbarkeit von Objekten ist insbesondere dadurch interessant, daß sie Ihnen gestattet, *leichtgewichtige Persistenz* zu implementieren. Persistenz bedeutet, daß der Lebenszyklus eines Objektes nicht mehr davon abhängt, ob ein Programm ausgeführt wird oder nicht. Das Objekt „existiert“ vielmehr auch zwischen zwei Programmaufrufen. Der Persistenzeffekt tritt dadurch ein, daß Sie ein serialisierbares Objekt auf die Festplatte schreiben und während der nächsten Programmausführung rekonstruieren. Die Bezeichnung „leichtgewichtig“ rührt daher, daß Sie das Objekt nicht einfach mit einem Schlüsselwort als persistent bezeichnen und alle weiteren Einzelheiten dem System überlassen können (vielleicht wird es diese Möglichkeit in Zukunft einmal geben). Statt dessen müssen Sie Objekte in Ihrem Programm explizit serialisieren beziehungsweise deserialisieren. Wenn Sie ein technisch besser ausgefeiltes Persistenzverfahren brauchen, können Sie ein Framework wie Hibernate (*hibernate.sourceforge.net*) in Betracht ziehen. Zu den Details, siehe „Thinking in Enterprise Java“ (der Entwurf steht unter *www.mindview.net*) zum Herunterladen zur Verfügung.

[167] Der Serialisierungsmechanismus für Objekte wurde aus zwei Gründen in den Sprachumfang aufgenommen. Die Remote Method Invocation (RMI) gestattet Objekten, die in entfernten Laufzeitumgebungen liegen, sich wie Objekte Ihrer lokalen Laufzeitumgebung zu verhalten. Beim Aufrufen

von Methoden entfernter Objekte tritt die Serialisierung auf, um Argumente und Rückgabewerte transportieren zu können. Auch RMI wird in „Thinking in Enterprise Java“ diskutiert.

[168] Der Serialisierungsmechanismus für Objekte ist auch für JavaBeans erforderlich (siehe Kapitel 23). When a Bean is used, its state information is generally configured at design time. Diese Zustandsinformation muß gespeichert und beim nächsten Programmstart wieder hergestellt werden. Diese Aufgabe übernimmt die Serialisierung.

[169] Es ist nicht schwierig ein Objekt zu serialisieren, wenn seine Klasse das Interface *`Serializable`* implementiert (ein Markierungsinterface, das heißt *`Serializable`* deklariert keine Methoden). Anläßlich der Aufnahme des Serialisierungsmechanismus' in den Sprachumfang, wurden viele Klassen aus der Standardbibliothek mit diesem Interface gekennzeichnet, darunter alle Wrapperklassen der primitiven Typen, alle Kollektionsklassen und viele andere. Selbst Klassenobjekte sind serialisierbar.

[170] Wenn Sie ein Objekt serialisieren möchten, brauchen Sie ein `OutputStream`-Objekt, welches Sie in ein `ObjectOutputStream`-Objekt verpacken. Es genügt nun, die `writeObject()`-Methode aufzurufen, um Ihr Objekt zu serialisieren und an den Ausgabestrom zu übergeben. Die Serialisierung ist byteorientiert und baut daher auf den Hierarchien unter den Basisklassen `InputStream` beziehungsweise `OutputStream` auf. Wenn Sie ein Objekt deserialisieren möchten, brauchen Sie ein `InputStream`-Objekt, welches Sie in ein `ObjectInputStream`-Objekt verpacken und dessen `readObject()`-Methode Sie aufrufen müssen. Der Rückgabetyp der `readObject()`-Methode ist `Object` und muß in den entsprechenden Zieltyp umgewandelt werden, um Mißverständnisse zu vermeiden.

[171] Eine besonders elegante Eigenschaft der Serialisierung besteht darin, daß nicht nur das betrachtete Objekt, sondern auch alle von diesem referenzierten Objekte und wiederum die von diesen Objekten referenzierten Objekte und so weiter erfaßt werden. Hinsichtlich dieser Baum-/Waldstruktur sagt man gelegentlich, ein Objekt sei mit einem „Netz aus Objekten“ (*web of objects*) verbunden. Die Baum-/Waldstruktur beinhaltet sowohl „einfache“, von Feldern referenzierte Objekte, als auch Arrays von Objektreferenzen. Es wäre eine verwirrende und unübersichtliche Aufgabe, ein eigenes Serialisierungsverfahren zu entwickeln. Der Serialisierungsmechanismus von Java bewerkstelligt diese Aufgabe dagegen fehlerlos, wobei zweifellos ein optimierter Algorithmus verwendet wird, um die Baum-/Waldstruktur zu traversieren. Das folgende Beispiel führt den Serialisierungsmechanismus anhand eines „Wurms“ von miteinander verknüpften Objekten vor. Die Segmente des Wurms sind miteinander verbunden und jedes Segment referenziert ein Array von Objekten einer Hilfsklasse namens `io.Data`:

```
//: io/Worm.java
// Demonstrates object serialization.
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Data implements Serializable {
    private int n;
    public Data(int n) { this.n = n; }
    public String toString() { return Integer.toString(n); }
}

public class Worm implements Serializable {
    private static Random rand = new Random(47);
    private Data[] d = {
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10)),
        new Data(rand.nextInt(10))
    };
    private Worm next;
```

```
    private char c;
    // Value of i == number of segments
    public Worm(int i, char x) {
        print("Worm constructor: " + i);
        c = x;
        if(--i > 0)
            next = new Worm(i, (char)(x + 1));
    }
    public Worm() {
        print("Default constructor");
    }
    public String toString() {
        StringBuilder result = new StringBuilder(":");
        result.append(c);
        result.append("(");
        for(Data dat : d)
            result.append(dat);
        result.append(")");
        if(next != null)
            result.append(next);
        return result.toString();
    }
    public static void main(String[] args)
        throws ClassNotFoundException, IOException {
        Worm w = new Worm(6, 'a');
        print("w = " + w);
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("worm.out"));
        out.writeObject("Worm storage\n");
        out.writeObject(w);
        out.close(); // Also flushes output
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("worm.out"));
        String s = (String)in.readObject();
        Worm w2 = (Worm)in.readObject();
        print(s + "w2 = " + w2);
        ByteArrayOutputStream bout = new ByteArrayOutputStream();
        ObjectOutputStream out2 = new ObjectOutputStream(bout);
        out2.writeObject("Worm storage\n");
        out2.writeObject(w);
        out2.flush();
        ObjectInputStream in2 = new ObjectInputStream(
            new ByteArrayInputStream(bout.toByteArray()));
        s = (String)in2.readObject();
        Worm w3 = (Worm)in2.readObject();
        print(s + "w3 = " + w3);
    }
} /* Output:
      Worm constructor: 6
      Worm constructor: 5
      Worm constructor: 4
      Worm constructor: 3
      Worm constructor: 2
      Worm constructor: 1
      w = :a(853):b(119):c(802):d(788):e(199):f(881)
      Worm storage
      w2 = :a(853):b(119):c(802):d(788):e(199):f(881)
```

```
    Worm storage
    w3 = :a(853):b(119):c(802):d(788):e(199):f(881)
*///:~
```

Die Elemente des `Data`-Arrays in der Klasse `Worm` sind mit Zufallszahlen initialisiert, um das Beispiel interessanter zu machen. (Auf diese Weise wird der Verdacht entkräftet, daß der Compiler irgendwelche Meta-Informationen speichert.) Jedes Segment des Wurms (`Worm`-Objekt) ist mit einem Buchstaben gekennzeichnet, der beim rekursiven Erzeugen der verketteten Liste von `Worm`-Objekten automatisch vergeben wird. Beim Erzeugen eines `Worm`-Objektes stellen Sie die Anzahl die Länge des Wurms (Anzahl der Segmente) über den Konstruktor ein. Zur Bewertung des `next`-Feldes wird der `Worm`-Konstruktor mit der um 1 reduzierten Länge aufgerufen. Das letzte `next`-Feld referenziert `null` und zeigt dadurch das Ende des Wurms an.

[172] Beim Entwurf der Klasse `Worm` war es mein Ziel, eine hinreichend komplexe Struktur zu schaffen, die nicht *einfach* serialisiert werden kann. Das Starten des Serialisierungsvorgangs ist dagegen einfach. Nachdem das `ObjectOutputStream`-Objekt mit darin enthaltenem `OutputStream`-Objekt erzeugt ist, genügt es, die `writeObject()`-Methode aufzurufen, um die Serialisierung zu starten. Beachten Sie den `writeObject()`-Aufruf für das `String`-Objekt. Sie können während des Serialisierungsvorgangs Werte primitiven Typs übergeben, in dem Sie die Methoden aufrufen, deren Namen Sie aus der Klasse `DataOutputStream` kennen (sowohl `DataOutputStream` als auch `ObjectOutputStream` implementiert das Interface *`DataOutput`*).

[173] Das Beispiel hat zwei ähnliche Abschnitte. Der erste schreibt und liest eine Datei, der zweite zur Abwechslung ein `byte`-Array. Sie können jeden von `DataInputStream` beziehungsweise `DataOutputStream` abgeleiteten Typ verwenden, um ein Objekt über den Serialisierungsmechanismus zu lesen beziehungsweise zu schreiben, insbesondere auch ein Netzwerk (siehe „Thinking in Enterprise Java“).

[174] Die Ausgabe zeigt, daß das deserialisierte Objekt tatsächlich alle im ursprünglichen Objekt vorhandenen Verknüpfungen besitzt.

[175] Beachten Sie, daß während der Deserialisierung kein Konstruktor aufgerufen wird, auch nicht der Standardkonstruktor. Das gesamte Objekt wird aus den über den Eingabestrom gelesenen Daten rekonstruiert.

Übungsaufgabe 27: (1) Schreiben Sie eine serialisierbare Klasse, deren Objekte ein Objekt einer anderen serialisierbaren Klasse referenzieren. Erzeugen Sie ein Objekt Ihrer Klasse, serialisieren und deserialisieren sie es und verifizieren Sie, daß der Vorgang korrekt abgelaufen ist. ■

19.12.1 Deserialisierung: Die Suche nach dem Klassenobjekt

[176] Eventuell überlegen Sie bereits, was für die Rekonstruktion eines Objektes aus dem serialisierten Zustand benötigt wird. Stellen Sie sich beispielsweise vor, Sie serialisieren ein Objekt und senden es als Datei oder über ein Netzwerk an eine andere Laufzeitumgebung. Ist es möglich, daß ein Programm oder die andere Laufzeitumgebung das Objekt alleine anhand der serialisierten Informationen rekonstrukieren können?

[177] Die beste Möglichkeit, dieses Frage zu beantworten ist, wie gewöhnlich, ein praktisches Experiment. Die folgende Datei befindet sich im *io*-Verzeichnis dieses Kapitels:

```
//: io/Alien.java
// A serializable class.
import java.io.*;
public class Alien implements Serializable {} ///:~
```

Das Programm, das ein `io.Alien`-Objekt erzeugt und serialisiert, befindet sich ebenfalls im Verzeichnis *io*:

```
//: io/FreezeAlien.java
// Create a serialized output file.
import java.io.*;

public class FreezeAlien {
    public static void main(String[] args) throws Exception {
        ObjectOutput out = new ObjectOutputStream(
            new FileOutputStream("X.file"));
        Alien quellek = new Alien();
        out.writeObject(quellek);
    }
} ///:~
```

Statt Ausnahmen abzufangen und zu behandeln, habe ich bei diesem Programm den „Quick and Dirty"-Ansatz gewählt. Die Ausnahmen werden als Ausnahmeverhalten der `main()`-Methode deklariert und über die Konsole ausgegeben.

[178] Nach dem Übersetzen und Ausführen enthält das *io*-Verzeichnis eine Datei namens *X.file*. Das folgende Programm befindet sich im Unterverzeichnis *io/xfiles*:

```
//: io/xfiles/ThawAlien.java
// Try to recover a serialized file without the
// class of object that's stored in that file.
// {RunByHand}
import java.io.*;

public class ThawAlien {
    public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream(new File("..", "X.file")));
        Object mystery = in.readObject();
        System.out.println(mystery.getClass());
    }
} /* Output:
      class Alien
   *////:~
```

Selbst das Öffnen der Datei und das Einlesen des vom `mystery`-Feld nach der Deserialisierung referenzierten Objektes setzt das Klassenobjekt von `Alien` voraus. Die Laufzeitumgebung kann die Datei *Alien.class* nicht finden (es sei denn die Klasse liegt zufällig im Klassenpfad; sollte hier aber nicht zutreffen). Die Laufzeitumgebung wirft eine Ausnahme vom Typ `ClassNotFoundException` aus. (Wieder einmal haben sich die Hinweise auf außerirdisches Leben in Luft aufgelöst, bevor der Beweis seiner Existenz überprüft werden könnte.) Die Laufzeitumgebung muß die entsprechende *.class* Datei finden können.

19.12.2 Steuerung des Serialisierungsvorgangs

[179] Die Standardserialisierung ist völlig problemlos anzuwenden. Was können Sie aber tun, wenn Sie bei der Serialierung eines Objektes spezielle Anforderungen berücksichtigen müssen? Eventuell können Sie aus Sicherheitsgründen einen Teil des Objektes nicht serialisieren oder es ist einfach nicht nötig, ein referenziertes Komponentenobjekt ebenfalls zu serialisieren, da es beim Deserialisieren ohnehin neu erzeugt wird.

[180] Sie können in den Serialisierungsvorgang eingreifen, wenn Sie statt *`Serializable`* das Interface

java.io.Externalizable implementieren. Das Interface *Externalizable* ist von *Serializable* abgeleitet und deklariert die Methoden `writeExternal()`und `readExternal()`, die während der Serialisierung beziehungsweise Deserialisierung Ihres Objektes automatisch aufgerufen werden. Sie können spezielle Operationen also im Körper dieser beiden Methoden ausführen.

[181] Das folgende Beispiel zeigt eine einfache Implementierung des *Externalizable*-Interfaces. Beachten Sie, daß die Klassen `Blip1` und `Blip2` von einem subtilen Unterschied abgesehen, nahezu identisch sind (mal sehen, ob Sie den Unterschied entdecken):

```
//: io/Blips.java
// Simple use of Externalizable & a pitfall.
import java.io.*;
import static net.mindview.util.Print.*;

class Blip1 implements Externalizable {
    public Blip1() {
        print("Blip1 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip1.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip1.readExternal");
    }
}

class Blip2 implements Externalizable {
    Blip2() {
        print("Blip2 Constructor");
    }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip2.writeExternal");
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip2.readExternal");
    }
}

public class Blips {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        print("Constructing objects:");
        Blip1 b1 = new Blip1();
        Blip2 b2 = new Blip2();
        ObjectOutputStream o =
            new ObjectOutputStream(new FileOutputStream("Blips.out"));
        print("Saving objects:");
        o.writeObject(b1);
        o.writeObject(b2);
        o.close();
        // Now get them back:
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream("Blips.out"));
        print("Recovering b1:");
        b1 = (Blip1)in.readObject();
```

```
            // OOPS! Throws an exception:
            //! print("Recovering b2:");
            //! b2 = (Blip2)in.readObject();
        }
    } /* Output:
          Constructing objects:
          Blip1 Constructor
          Blip2 Constructor
          Saving objects:
          Blip1.writeExternal
          Blip2.writeExternal
          Recovering b1:
          Blip1 Constructor
          Blip1.readExternal
       *///:~
```

Das `Blip2`-Objekt wird nicht rekonstruiert, da zuvor eine Ausnahme hervorgerufen wird. Erkennen Sie den Unterschied zwischen `Blip1` und `Blip2`? Der `Blip1`-Konstruktor ist als `public` deklariert, nicht aber der Konstruktor von `Blip2` (dies ist die Ursache der Ausnahme beim Rekonstruieren des `Blip2`-Objektes). Deklarieren Sie versuchsweise den `Blip2`-Konstruktor als `public` und entfernen Sie die `//!`-Kommentarzeichen.

[182] Beim Rekonstruieren des von `b1` referenzierten `Blip1`-Objektes wird der Standardkonstruktor von `Blip1` aufgerufen, im Gegensatz zur Rekonstruktion eines Objektes dessen Klasse das Interface *`Serializable`* implementiert (das serialisierbare Objekt wird ohne Konstruktor und komplett aus den gespeicherten Daten rekonstruiert). Bei externalisierbaren Objekten erfolgt zunächst die normale Standardrekonstruktion (mit Initialisierung der Felder bei Definition) und *erst danach* wird die `readExternal()`-Methode aufgerufen. Beachten Sie diese Reihenfolge, vor allem die Tatsache, daß die Standardrekonstruktion *immer* stattfindet, um bei Ihren externalisierbaren Objekten das korrekte Verhalten zu gewährleisten.

[183] Das folgende Beispiel zeigt wie ein externalisierbares Objekt komplett gespeichert und anschließend rekonstruiert wird:

```
//: io/Blip3.java
// Reconstructing an externalizable object.
import java.io.*;
import static net.mindview.util.Print.*;

public class Blip3 implements Externalizable {
    private int i;
    private String s; // No initialization
    public Blip3() {
        print("Blip3 Constructor");
        // s, i not initialized
    }
    public Blip3(String x, int a) {
        print("Blip3(String x, int a)");
        s = x;
        i = a;
        // s & i initialized only in non-default constructor.
    }
    public String toString() { return s + i; }
    public void writeExternal(ObjectOutput out)
        throws IOException {
        print("Blip3.writeExternal");
        // You must do this:
```

```
        out.writeObject(s);
        out.writeInt(i);
    }
    public void readExternal(ObjectInput in)
        throws IOException, ClassNotFoundException {
        print("Blip3.readExternal");
        // You must do this:
        s = (String)in.readObject();
        i = in.readInt();
    }
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        print("Constructing objects:");
        Blip3 b3 = new Blip3("A String ", 47);
        print(b3);
        ObjectOutputStream o =
            new ObjectOutputStream(new FileOutputStream("Blip3.out"));
        print("Saving object:");
        o.writeObject(b3);
        o.close();
        // Now get it back:
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream("Blip3.out"));
        print("Recovering b3:");
        b3 = (Blip3)in.readObject();
        print(b3);
    }
} /* Output:
      Constructing objects:
      Blip3(String x, int a)
      A String 47
      Saving object:
      Blip3.writeExternal
      Recovering b3:
      Blip3 Constructor
      Blip3.readExternal
      A String 47
   *///:~
```

Die Felder `s` und `i` werden nur vom zweiten Konstruktor initialisiert, nicht aber vom Standardkonstruktor. Wenn Sie diese Felder nicht per `readExternal()` initialisieren, wird `s` der Wert `null` und `i` der Wert 0 zugewiesen (da der Speicherplatz eines Objektes im ersten Schritt der Objekterzeugung mit Nullen übergeschrieben wird). Wenn Sie die beiden Zeilen nach „You must do this: “ auskommentieren und das Programm aufrufen, sehen Sie, daß beim rekonstruierten Objekt `s` gleich `null` und `i` gleich 0 ist.

[184] Wenn Sie eine Klasse von einer externalisierbaren Klasse ableiten, rufen Sie typischerweise die Basisklassenversionen der Methoden `writeExternal()` und `readExternal()` auf, um die korrekte Speicherung beziehungsweise Rekonstruktion der Basisklassenfelder zu gewährleisten.

[185] Damit die Serialisierung per *`Externalizable`* korrekt funktioniert, müssen Sie nicht nur die benötigten Daten des serialisierbaren Objektes mit Hilfe der `writeExternal()`-Methode schreiben (es gibt kein Standardverhalten, das die von einem serialisierbaren Objekt referenzierten Objekte erfaßt), sondern auch über die `readExternal()`-Methode wieder einlesen. Das ist auf den ersten Blick verwirrend, da die Standardrekonstruktion externalisierbarer Objekte *scheinbar* automatisch Daten speichert und ausliest. Aber das trifft nicht zu.

Übungsaufgabe 28: (2) Kopieren Sie *Blips.java* und nennen Sie die Kopie *BlipCheck.java*. Benennen Sie in *BlipCheck.java* die Klasse die Klasse `Blip2` in `BlipCheck` um und deklarieren Sie `BlipCheck` als `public`, während Sie das `public`-Schlüsselwort bei der Klasse `Blips` entfernen. Entfernen Sie die `//!`-Kommentarzeichen und führen Sie das Programm aus (*mit* den zuvor auskommentierten Zeilen). Kommentieren Sie nun den Standardkonstruktor von `BlipCheck` aus. Führen Sie das Programm aus und erläutern Sie, warum es funktioniert. Beachten Sie, daß Sie das Programm nach dem Übersetzen mit `java Blips` aufrufen müssen, da sich die `main()`-Methode noch immer in `Blips` befindet. ■

Übungsaufgabe 29: (2) Kommentieren Sie in *Blip3.java* die beiden Zeilen nach „You must do this: “ aus und rufen Sie das Programm auf. Erklären Sie das Ergebnis und warum es abweicht, wenn die auskommentierten Zeilen einkommentiert sind. ■

19.12.2.1 Das transient-Schlüsselwort

[186] Wenn Sie in die Serialisierung eingreifen, gibt es unter Umständen ein Komponentenobjekt, dessen automatische Speicherung und Rekonstruktion Sie vermeiden möchten. Eine solche Situation ergibt sich häufig, wenn das Komponentenobjekt empfindliche (*sensitive*) Informationen enthält, etwa ein Paßwort, die Sie nicht serialisiert haben wollen. Selbst wenn die Informationen im Komponentenobjekt in einem als `private` deklarierten Feld steht, können sie im serialisierten Zustand aus der Datei gelesen oder durch Abhören der Netzwerkverbindung rekonstruiert werden.

[187] Eine Möglichkeit, um die Serialisierung empfindlicher Felder Ihres Objektes zu vermeiden besteht darin, daß die zugehörige Klasse das Interface ***`Externalizable`*** implementiert (siehe Beispiele *Blips.java* und *Blip3.java*). In diesem Fall wird nichts automatisch serialisiert und sie geben in der `writeExternal()`-Methode explizit an, welche Felder serialisiert werden sollen.

[188] Implementiert die Klasse Ihres Objektes das Interface ***`Serializable`***, so wird der Serialisierungsvorgang automatisch durchgeführt. Sie können in den Vorgang eingreifen, indem Sie einzelne Felder mit dem Schlüsselwort `transient` kennzeichnen, wodurch die Serialisierung des von einem solchen Feld referenzierten Objektes unterbleibt. Das Schlüsselwort `transient` informiert den Serialisierungsmechanismus darüber, daß er sich nicht mit der Speicherung und Rekonstruktion des entsprechenden Objektes auseinanderzusetzen braucht, sondern daß Sie sich selbst darum kümmern.

[189] Stellen Sie sich zum Beispiel ein Objekt vor, das die Anmeldedaten einer Sitzung repräsentiert. Angenommen, Sie möchten alle Informationen mit Ausnahme des Paßwortes nach der erfolgten Anmeldung speichern. Am einfachsten implementieren Sie dazu das Interface ***`Serializable`*** und deklarieren das Paßwortfeld als `transient`:

```
//: io/Logon.java
// Demonstrates the ''transient'' keyword.
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Logon implements Serializable {
    private Date date = new Date();
    private String username;
    private transient String password;
    public Logon(String name, String pwd) {
        username = name;
        password = pwd;
    }
```

```
    public String toString() {
        return ''logon info: \n username: '' + username +
            ''\n date: '' + date + ''\n password: '' + password;
    }
    public static void main(String[] args) throws Exception {
        Logon a = new Logon(''Hulk'', ''myLittlePony'');
        print(''logon a = '' + a);
        ObjectOutputStream o =
            new ObjectOutputStream(new FileOutputStream(''Logon.out''));
        o.writeObject(a);
        o.close();
        TimeUnit.SECONDS.sleep(1); // Delay
        // Now get them back:
        ObjectInputStream in =
            new ObjectInputStream(new FileInputStream(''Logon.out''));
        print(''Recovering object at '' + new Date());
        a = (Logon)in.readObject();
        print(''logon a = '' + a);
    }
} /* Output: (Sample)
      logon a = logon info:
      username: Hulk
      date: Sat Nov 19 15:03:26 MST 2005
      password: myLittlePony
      Recovering object at Sat Nov 19 15:03:28 MST 2005
      logon a = logon info:
      username: Hulk
      date: Sat Nov 19 15:03:26 MST 2005
      password: null
   *///:~
```

[190] Die Felder `date` und `username` sind „gewöhnliche“, das heißt keine transienten Felder und werden vom Serialisierungsmechanismus automatisch erfaßt. Das `password`-Feld ist ein transientes Feld und wird daher nicht auf der Festplatte gespeichert. Der Serialisierungsmechanismus unternimmt auch keinen Versuch, das referenzierte Objekt zu rekonstruieren. Nach der Rekonstruktion des `Logon`-Objektes enthält das `password`-Feld den Wert `null`. Beachten Sie, daß die `toString()`-Methode das `String`-Objekt per `+`-Operator zusammensetzt, wobei die `null`-Referenz automatisch in die Zeichenkette „null“ umgewandelt wird.

[191] Das `date`-Feld wird auf der Festplatte gespeichert und von dort gelesen, also nicht neu erzeugt.

[192] Da bei Objekten von externalisierbaren Klassen kein Feld automatisch gespeichert wird, ist das Schlüsselwort `transient` nur bei serialisierbaren Klassen erlaubt.

19.12.2.2 Eine Alternative zu Externalizable

[193] Wenn Sie nicht darauf erpicht sind, das Interface *`Externalizable`* zu implementieren, dann gibt es noch einen Ausweg. Sie können *`Serializable`* implementieren und zwei zusätzliche Methoden namens `writeObject()` und `readObject()` *anlegen* (beachten Sie, daß ich „zusätzlich anlegen“ und nicht „überschreiben“ oder „implementieren“ geschrieben habe), die während der Serialisierung beziehungsweise Deserialisierung automatisch aufgerufen werden. Genauer: Sind die beiden Methoden `writeObject()` und `readObject()` vorhanden, so werden sie *anstelle der Standardserialisierung* aufgerufen.

[194] Die beiden Methoden haben die folgenden Signaturen:

```
private void writeObject(ObjectOutputStream stream)
    throws IOException;

private void readObject(ObjectInputStream stream)
    throws IOException, ClassNotFoundException;
```

Vom Design her betrachtet, werden die Dinge hier seltsam: Eventuell ist Ihr erster Gedanken, daß diese beiden Methoden weder zu einer Basisklasse gehören noch im Interface *`Serializable`* deklariert sind (trivialerweise) und daher in einem oder mehreren eigenen Interfaces deklariert sein sollten. Beachten Sie aber, daß die Methoden als `private` deklariert sind, also nur von Komponenten ihrer eigenen Klasse aufgerufen werden können. Allerdings rufen Sie diese Methoden keineswegs selbst auf. Die private Methode `writeObject()`/`readObject()` wird nämlich von der `ObjectOutputStream`-Methode `writeObject()` beziehungsweise `readObject()` aufgerufen. (Beachten Sie meine enorme Beherrschung, mich angesichts der Wahl der Methodennamen nicht zu einer Hetzrede verleiten zu lassen.) Eventuell fragen Sie sich, wie das `ObjectOutputStream`- beziehungsweise `ObjectInputStream`-Objekt überhaupt eine *private* Methode Ihrer Klasse aufrufen kann. Wir können an dieser Stelle nur vermuten, daß der verantwortliche Mechanismus ein Teil der Serialisierungsmagie[4] ist.

[195] Jede in einem Interface deklarierte Komponente ist automatisch öffentlich (`public`). Daraus folgt, daß `writeObject()` und `readObject()` als *private* Methoden nicht Teil eines Interfaces sein können. Da Sie die Signaturen genau einhalten müssen, tun Sie effektiv dasselbe, als wenn Sie ein Interface implementieren.

[196] Wenn Sie die `ObjectOutputStream`-Methode `writeObject()` aufrufen, wird das übergebene serialisierbare Objekt per Reflexion dahingehend geprüft, ob es eine eigene `writeObject()`-Methode besitzt. Ist dies der Fall, so wird der normale Serialisierungsvorgang übersprungen und die objekteigene `writeObject()`-Methode aufgerufen. Dasselbe gilt für `readObject()`.

[197] Es gibt noch einen Kniff: Sie können im Körper Ihrer privaten `writeObject()`-Methode die Standardserialisierung auslösen, also die `ObjectOutputStream`-Version der `writeObject()`-Methode aufrufen, in dem Sie die `defaultWriteObject()`-Methode aufrufen. Analog steht Ihnen in Ihrer privaten `readObject()`-Methode eine Methode namens `defaultReadObject()` zur Verfügung. Das folgende einfache Beispiel zeigt das Eingreifen in den Speicher- und Rekonstruktionsvorgang bei einem serialisierbaren Objekt:

```
//: io/SerialCtl.java
// Controlling serialization by adding your own
// writeObject() and readObject() methods.
import java.io.*;

public class SerialCtl implements Serializable {
    private String a;
    private transient String b;
    public SerialCtl(String aa, String bb) {
        a = "Not Transient: " + aa;
        b = "Transient: " + bb;
    }
    public String toString() { return a + "\n" + b; }
    private void writeObject(ObjectOutputStream stream)
        throws IOException {
        stream.defaultWriteObject();
        stream.writeObject(b);
    }
    private void readObject(ObjectInputStream stream)
        throws IOException, ClassNotFoundException {
```

[4] Abschnitt 15.9 zeigt, wie private Methoden von außerhalb einer Klasse aufgerufen werden können.

```
        stream.defaultReadObject();
        b = (String)stream.readObject();
    }
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
        SerialCtl sc = new SerialCtl("Test1", "Test2");
        System.out.println("Before:\n" + sc);
        ByteArrayOutputStream buf= new ByteArrayOutputStream();
        ObjectOutputStream o = new ObjectOutputStream(buf);
        o.writeObject(sc);
        // Now get it back:
        ObjectInputStream in = new ObjectInputStream(
            new ByteArrayInputStream(buf.toByteArray()));
        SerialCtl sc2 = (SerialCtl)in.readObject();
        System.out.println("After:\n" + sc2);
    }
} /* Output:
      Before:
      Not Transient: Test1
      Transient: Test2
      After:
      Not Transient: Test1
      Transient: Test2
   *///:~
```

Das Beispiel hat ein gewöhnliches und eine transientes `String`-Feld, um zeigen zu können, daß das nicht-transiente Feld automatisch durch `defaultWriteObject()` gespeichert und rekonstruiert wird, während das transiente Feld explizit gespeichert und wiederhergestellt werden muß. Die Felder werden statt bei ihrer Definition über den Konstruktor initialisiert, um zeigen zu können, daß sie nicht während der Deserialisierung mit Hilfe eines automatischen Mechanismus' initialisiert werden.

[198] Wenn sie die Standardserialisierung zum Schreiben der nicht-transienten Felder Ihres Objektes verwenden, müssen Sie `defaultWriteObject()` als erste Operation Ihrer `writeObject()`-Methode aufrufen (analog `defaultReadObject()` als erste Operation Ihrer `readObject()`-Methode). Diese Methodenaufrufe sind sonderbar. Obwohl Sie beispielsweise `defaultWriteObject()` kein Objekt übergeben, kennt die Methode die Referenz des serialisierten Objektes und alle nicht-transienten Komponenten. Gespensterhaft.

[199] Die Anweisungen zum Speichern und Rekonstruieren der transienten Objekte sind vertrauter. Überlegen Sie sich dennoch, was geschieht. In der `main()`-Methode wird ein `io.SerialCtl`-Objekt erzeugt, serialisiert und dem Ausgabestrom übergeben. (Beachten Sie die Verwendung eines Puffers statt einer Datei. Für das `ObjectOutputStream`-Objekt ist beides gleich.) Der Serialisierungsvorgang beginnt mit der Zeile:

```
o.writeObject(sc);
```

Die `writeObject()`-Methode untersucht das von `sc` referenzierte Objekt, ob es eine eigene `writeObject()`-Methode besitzt. (Die Untersuchung bezieht sich weder auf ein Interface, welches ohnehin nicht existiert, noch auf die Klasse des Objektes, sondern basiert auf Reflexion.) Existiert die Methode, so wird sie aufgerufen. Dasselbe gilt für `readObject()`. Vielleicht war dies der einzige praktikable Weg, um das Problem zu lösen, aber es ist ein seltsamer Weg.

19.12.2.3 Serialisierung und Versionskontrolle

[200] Eventuell möchten Sie einmal die Version einer serialisierbaren Klasse ändern (während noch Objekte der ursprünglichen Klasse in einer Datenbank gespeichert sind). Versionsänderungen werden zwar unterstützt, aber der Bedarf hierfür zeigt sich nur in speziellen Situationen und erfordert zusätzliche Tiefe an Verständnis, die wir an dieser Stelle nicht anstreben können. Die Dokumentation des Java Development Kits behandelt dieses Thema sorgfältig.

[201] Viele Einträge in der JDK-Dokumentation beginnen mit der

> **Warnung:** "Serialized objects of this class will not be compatible with future Swing releases. The current serialization support is appropriate for short term storage or RMI between applications."
>
> Übersetzt etwa: „Serialisierbare Objekte dieser Klasse werden mit zukünftigen Versionen der Swing-Bibliothek nicht kompatibel sein. Das gegenwärtige Serialisierungsverfahren ist für kurzfristige Speicherung oder Methodenfernaufrufe per RMI geeignet."

Der Serialisierungsmechanismus ist zu einfach, um in jeder denkbaren Situation zuverlässig zu sein, insbesondere bei JavaBeans. Sun Microsystems arbeitet an einer Korrektur des Entwurfs. Daher die Warnung.

19.12.3 Using persistence

[202] Die Idee, per Serialisierung einen Teil des Zustandes eines Programms zu sichern, um den aktuellen Zustand zu einem späteren Zeitpunkt wieder herzustellen, hat einen gewissen Reiz. Zuvor müssen aber einige Fragen beantwortet werden. Was geschieht beispielsweise, wenn Sie zwei verschiedene Objekte serialisieren, die beide ein gemeinsames drittes Objekt referenzieren. Stellen Sie sich vor, Sie rekonstruieren die beiden ersteren Objekte aus dem serialisierten Zustand; do you get only one occurrence of the third object? Was geschieht, wenn die beiden ersteren Objekte in zwei separate Datei serialisiert werden und an verschiedenen Stellen im Programm deserialisiert werden?

[203] Das folgende Beispiel veranschaulicht das Problem:

```
//: io/MyWorld.java
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class House implements Serializable {}

class Animal implements Serializable {
    private String name;
    private House preferredHouse;
    Animal(String nm, House h) {
        name = nm;
        preferredHouse = h;
    }
    public String toString() {
        return name + "[" + super.toString() +
            "], " + preferredHouse + "\n";
    }
}

public class MyWorld {
    public static void main(String[] args)
        throws IOException, ClassNotFoundException {
```

```
        House house = new House();
        List<Animal> animals = new ArrayList<Animal>();
        animals.add(new Animal("Bosco the dog", house));
        animals.add(new Animal("Ralph the hamster", house));
        animals.add(new Animal("Molly the cat", house));
        print("animals: " + animals);
        ByteArrayOutputStream buf1 = new ByteArrayOutputStream();
        ObjectOutputStream o1 = new ObjectOutputStream(buf1);
        o1.writeObject(animals);
        o1.writeObject(animals); // Write a 2nd set
        // Write to a different stream:
        ByteArrayOutputStream buf2 = new ByteArrayOutputStream();
        ObjectOutputStream o2 = new ObjectOutputStream(buf2);
        o2.writeObject(animals);
        // Now get them back:
        ObjectInputStream in1 = new ObjectInputStream(
            new ByteArrayInputStream(buf1.toByteArray()));
        ObjectInputStream in2 = new ObjectInputStream(
            new ByteArrayInputStream(buf2.toByteArray()));
        List
            animals1 = (List)in1.readObject(),
            animals2 = (List)in1.readObject(),
            animals3 = (List)in2.readObject();
        print("animals1: " + animals1);
        print("animals2: " + animals2);
        print("animals3: " + animals3);
    }
} /* Output: (Sample)
      animals: [Bosco the dog[Animal@addbf1], House@42e816
      , Ralph the hamster[Animal@9304b1], House@42e816
      , Molly the cat[Animal@190d11], House@42e816
      ]
      animals1: [Bosco the dog[Animal@de6f34], House@156ee8e
      , Ralph the hamster[Animal@47b480], House@156ee8e
      , Molly the cat[Animal@19b49e6], House@156ee8e
      ]
      animals2: [Bosco the dog[Animal@de6f34], House@156ee8e
      , Ralph the hamster[Animal@47b480], House@156ee8e
      , Molly the cat[Animal@19b49e6], House@156ee8e
      ]
      animals3: [Bosco the dog[Animal@10d448], House@e0e1c6
      , Ralph the hamster[Animal@6ca1c], House@e0e1c6
      , Molly the cat[Animal@1bf216a], House@e0e1c6
      ]
   *///:~
```

Ein interessanter Aspekt dieses Beispiel ist, daß Sie per Serialisierung in Verbindung mit einem `byte`-Array eine rekursive Kopie eines serialisierten Objektes herstellen können. („Rekursive Kopie“ bedeutet, das gesamte Netz von Objekten zu duplizieren, im Gegensatz zum Ausgangsobjekt alleine.) Das Kopieren von Objekten wird in den online verfügbaren Anhängen zu diesem Buch behandelt. Jedes `Animal`-Objekt besitzt ein `House`-Feld. Die `main()`-Methode erzeugt ein *`List<Animal>`*-Objekt mit einigen Elementen und serialisiert es zweimal in ein und demselben Ausgabestrom und ein drittesmal in einen separaten Ausgabestrom. Nach der Deserialisierung erfolgt die Ausgabe (die Objekte liegen bei jedem Programm an anderen Speicheradressen).

[204] Sie erwarten zurecht, daß die serialisierten Objekte andere Speicheradressen haben, als die ursprünglichen Objekte. Beachten Sie aber, daß die Elemente in den deserialisierten, von `ani-`

`mal1` und `animal2` referenzierten *`List<Animal>`*-Objekten übereinstimmende Speicheradressen haben und dasselbe `House`-Objekt referenzieren. Dagegen hat die Laufzeitumgebung bei der Rekonstruktion von `animal3` keine Möglichkeit, um herauszufinden, daß diese Objekte mit den vorigen übereinstimmen und erzeugt daher ein neues Netz von Objekten.

[205] Wenn Sie genau einen Ausgabestrom zum Serialisieren verwenden, wird das Netz von Objekten im ursprünglichen Zustand wieder aufgebaut, ohne versehentlich Objekte zu duplizieren. Sie können den Zustand Ihrer Objekte während der Serialisierung selbstverständlich ändern, müssen in diesem Fall die Verantwortung aber selbst übernehmen. Die Objekte werden in ihrem Zustand und mit ihren Verknüpfungen zu anderen Objekten vor der Serialisierung rekonstruiert.

[206] Die sicherste Vorgehensweise, um den Zustand zu sichern, besteht darin, die Serialisierung als atomare Operation durchzuführen. Wenn Sie einen Teil serialisieren, etwas anderes machen, wieder einen Teil serialisieren und so weiter dann wird der Zustand Ihrer Objekte nicht konsistent gespeichert. Speichern Sie statt dessen alle Objekte, die für den Zustand Ihrer Anwendung ausschlaggebend sind, in einem Container und serialisieren Sie diesen in einer Operation. Der Container läßt sich anschließend mit nur einem Methodenaufruf rekonstruieren.

[207] Das folgende Beispiel demonstriert diesen Ansatz anhand einer imaginären CAD-Anwendung (Computer Aided Design). Das Beispiel behandelt insbesondere die Serialisierung statischer Felder. Nach der Dokumentation des Java Development Kits ist die Klasse `Class` serialisierbar und es sollte nicht schwer sein, statische Felder durch Serialisierung des Klassenobjektes zu erfassen. Die Vorgehensweise scheint sinnvoll zu sein:

```
//: io/StoreCADState.java
// Saving the state of a pretend CAD system.
import java.io.*;
import java.util.*;

abstract class Shape implements Serializable {
    public static final int RED = 1, BLUE = 2, GREEN = 3;
    private int xPos, yPos, dimension;
    private static Random rand = new Random(47);
    private static int counter = 0;
    public abstract void setColor(int newColor);
    public abstract int getColor();
    public Shape(int xVal, int yVal, int dim) {
        xPos = xVal;
        yPos = yVal;
        dimension = dim;
    }
    public String toString() {
        return getClass() +
            ''color['' + getColor() + ''] xPos['' + xPos +
            ''] yPos['' + yPos + ''] dim['' + dimension + '']\n'';
    }
    public static Shape randomFactory() {
        int xVal = rand.nextInt(100);
        int yVal = rand.nextInt(100);
        int dim = rand.nextInt(100);
        switch(counter++ % 3) {
        default:
        case 0: return new Circle(xVal, yVal, dim);
        case 1: return new Square(xVal, yVal, dim);
        case 2: return new Line(xVal, yVal, dim);
        }
    }
```

```
}

class Circle extends Shape {
    private static int color = RED;
    public Circle(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

class Square extends Shape {
    private static int color;
    public Square(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
        color = RED;
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

class Line extends Shape {
    private static int color = RED;
    public static void
        serializeStaticState(ObjectOutputStream os)
        throws IOException { os.writeInt(color); }
    public static void
        deserializeStaticState(ObjectInputStream os)
        throws IOException { color = os.readInt(); }
    public Line(int xVal, int yVal, int dim) {
        super(xVal, yVal, dim);
    }
    public void setColor(int newColor) { color = newColor; }
    public int getColor() { return color; }
}

public class StoreCADState {
    public static void main(String[] args) throws Exception {
        List<Class<? extends Shape>> shapeTypes =
            new ArrayList<Class<? extends Shape>>();
        // Add references to the class objects:
        shapeTypes.add(Circle.class);
        shapeTypes.add(Square.class);
        shapeTypes.add(Line.class);
        List<Shape> shapes = new ArrayList<Shape>();
        // Make some shapes:
        for(int i = 0; i < 10; i++)
            shapes.add(Shape.randomFactory());
        // Set all the static colors to GREEN:
        for(int i = 0; i < 10; i++)
            ((Shape)shapes.get(i)).setColor(Shape.GREEN);
        // Save the state vector:
        ObjectOutputStream out = new ObjectOutputStream(
            new FileOutputStream("CADState.out"));
        out.writeObject(shapeTypes);
        Line.serializeStaticState(out);
        out.writeObject(shapes);
        // Display the shapes:
        System.out.println(shapes);
```

```
    }
} /* Output:
      [class Circlecolor[3] xPos[58] yPos[55] dim[93]
      , class Squarecolor[3] xPos[61] yPos[61] dim[29]
      , class Linecolor[3] xPos[68] yPos[0] dim[22]
      , class Circlecolor[3] xPos[7] yPos[88] dim[28]
      , class Squarecolor[3] xPos[51] yPos[89] dim[9]
      , class Linecolor[3] xPos[78] yPos[98] dim[61]
      , class Circlecolor[3] xPos[20] yPos[58] dim[16]
      , class Squarecolor[3] xPos[40] yPos[11] dim[22]
      , class Linecolor[3] xPos[4] yPos[83] dim[6]
      , class Circlecolor[3] xPos[75] yPos[10] dim[42]
      ]
   *///:~
```

Die Klasse `io.Shape` ist serialisierbar, also auch jede von `Shape` abgeleitete Klasse. Jedes `Shape`-Objekt enthält Daten und jede von `Shape` abgeleitete Klasse besitzt ein statisches Feld, welches die Farbe aller Objekte der jeweiligen `Shape`-Unterklasse festlegt. (Das Anlegen des statischen Feldes in der Basisklasse würde bedeuten, daß das Feld nur in der Basisklasse existiert, da statische Felder nicht vererbt werden.) Die in der Basisklasse deklarierten abstrakten Abfrage- und Änderungsmethoden für die Farbe werden in den abgeleiteten Klassen überschrieben (statische Methoden ~~are not dynamically bound, so there are normal methods~~). Die `randomFactory()`-Methode erzeugt bei jedem Aufruf ein `Shape`-Objekt mit Zufallswerten für dessen Felder.

[208] Die Klassen `Circle` und `Square` sind einfache Ableitungen von `Shape`. Der einzige Unterschied besteht darin, daß `Circle` die Farbe zum Zeitpunkt der Felddefinition festlegt, `Square` dagegen per Konstruktor. Wir besprechen die Klasse `Line` später.

[209] In der `main()`-Methode speichert ein `ArrayList`-Objekt die Klassenobjekte und ein weiteres die `Shape`-Objekte.

[210] Die Rekonstruktion der Objekt ist einfach:

```
//: io/RecoverCADState.java
// Restoring the state of the pretend CAD system.
// {RunFirst: StoreCADState}
import java.io.*;
import java.util.*;

public class RecoverCADState {
    @SuppressWarnings("unchecked")
        public static void main(String[] args) throws Exception {
        ObjectInputStream in = new ObjectInputStream(
            new FileInputStream("CADState.out"));
        // Read in the same order they were written:
        List<Class<? extends Shape>> shapeTypes =
            (List<Class<? extends Shape>>)in.readObject();
        Line.deserializeStaticState(in);
        List<Shape> shapes = (List<Shape>)in.readObject();
        System.out.println(shapes);
    }
} /* Output:
      [class Circlecolor[1] xPos[58] yPos[55] dim[93]
      , class Squarecolor[0] xPos[61] yPos[61] dim[29]
      , class Linecolor[3] xPos[68] yPos[0] dim[22]
      , class Circlecolor[1] xPos[7] yPos[88] dim[28]
      , class Squarecolor[0] xPos[51] yPos[89] dim[9]
      , class Linecolor[3] xPos[78] yPos[98] dim[61]
```

```
    , class Circlecolor[1] xPos[20] yPos[58] dim[16]
    , class Squarecolor[0] xPos[40] yPos[11] dim[22]
    , class Linecolor[3] xPos[4] yPos[83] dim[6]
    , class Circlecolor[1] xPos[75] yPos[10] dim[42]
    ]
*///:~
```

Die Werte der Felder `xPos`, `yPos` und `dim` wurden korrekt gespeichert und wieder hergestellt, nicht aber die Werte der statischen Felder. Ursprünglich waren die statischen Felder alle mit 3 bewertet, aber die Feldinhalte wurden nicht überall wieder hergestellt. `Circle`-Objekte sind nun rot (der Wert 1 wird bei der Definition zugewiesen), `Square`-Objekte haben die nicht definierte Farbe 0 (initialisiert per Konstruktor). Werden die statischen Felder etwa nicht serialisiert? Ja, obwohl die Klasse `Class` serialisierbar ist, verhält sie sich anders als erwartet. Wenn Sie statische Felder serialisieren wollen, müssen Sie in den Serialisierungsvorgang eingreifen.

[211] Dieser Eingriff ist die Aufgabe der Methoden `serializeStaticState()` und `deserializeStaticState()` in der Klasse `Line`. Beide Methoden werden während des Speicher- beziehungsweise Rekonstruktionsvorgangs explizit aufgerufen. (Beachten Sie die Reihenfolge beim Schreiben/Lesen der Datei.) Folgende Korrekturen sind nötig, damit das Beispiel korrekt arbeitet:

- Legen Sie in jeder von `Shape` abgeleiteten Klasse die Methoden `serializeStaticState()` und `deserializeStaticState()` an.
- Löschen Sie das von `shapeTypes` referenzierte `ArrayList`-Objekt und alle diesbezüglichen Anwendungen.
- Rufen Sie die neuen statischen Methoden `serializeStaticState()` und `deserializeStaticState()` auf.

[212] Sicherheit ist ein Thema, mit dem Sie sich unter Umständen auseinander müssen, da die Serialisierung auch private Daten erfaßt. Ist Sicherheit in Ihrer Situation wichtig, so sollten die entsprechenden Felder als `transient` deklariert werden. In diesem Fall müssen Sie allerdings einen sicheren Weg finden, um diese Felder nach der Rekonstruktion zu bewerten.

Übungsaufgabe 30: (1) Korrigieren Sie das Programm *CADState.java*, wie im Text beschrieben. ■

19.13 XML

[213] Eine wesentliche Einschränkung der Serialisierung besteht darin, daß Sie an Java gebunden sind: Nur Java-Programme sind imstande, ein serialisiertes Objekt zu deserialisieren. Die Umwandlung von Daten in ein XML-konformes Format ist eine weniger einschränkende Lösung und gestattet die Auswertung unter vielen Betriebssystemen und Programmiersprachen.

[214] Aufgrund der Beliebtheit von XML existiert eine unüberschaubare Vielfalt von XML-Schnittstellen, darunter auch die `javax.xml.*`-Bibliotheken des Java Development Kits. Ich habe mich für Elliotte Rusty Harolds quelloffene XOM-Bibliothek entschieden (Sie können die Bibliothek und ihre Dokumentation von *www.xom.nu* herunterladen), da es die einfachste und übersichtlichste Möglichkeit zum Schreiben und Bearbeiten von XML-Dokumenten mit Java zu sein scheint. Insbesondere achtet XOM auf die Korrektheit des XML-Dokuments.

[215] Stellen Sie sich zum Beispiel vor, ein `xml.Person`-Objekt enthalte Vor- und Nachname einer Person und solle als XML formatiert werden. Die Klasse `Person` im folgenden Beispiel hat eine Methode namens `getXML()`, die per XOM den Inhalt eines `Person`-Objektes in das Äquivalent eines XML-Elementes (`nu.xom.Element`-Objekt) umwandelt, sowie einen Konstruktor, der ein solches

`Element`-Objekt erwartet und daraus die `Person`-Feldinhalte extrahiert (beachten Sie, daß die XML-Beispiele in einem eigenen Unterverzeichnis stehen):

```
//: xml/Person.java
// Use the XOM library to write and read XML
// {Requires: nu.xom.Node; You must install
// the XOM library from http://www.xom.nu }
import nu.xom.*;
import java.io.*;
import java.util.*;

public class Person {
    private String first, last;
    public Person(String first, String last) {
        this.first = first;
        this.last = last;
    }
    // Produce an XML Element from this Person object:
    public Element getXML() {
        Element person = new Element("person");
        Element firstName = new Element("first");
        firstName.appendChild(first);
        Element lastName = new Element("last");
        lastName.appendChild(last);
        person.appendChild(firstName);
        person.appendChild(lastName);
        return person;
    }
    // Constructor to restore a Person from an XML Element:
    public Person(Element person) {
        first = person.getFirstChildElement("first").getValue();
        last = person.getFirstChildElement("last").getValue();
    }
    public String toString() { return first + " " + last; }
    // Make it human-readable:
    public static void
        format(OutputStream os, Document doc) throws Exception {
        Serializer serializer = new Serializer(os,"ISO-8859-1");
        serializer.setIndent(4);
        serializer.setMaxLength(60);
        serializer.write(doc);
        serializer.flush();
    }
    public static void main(String[] args) throws Exception {
        List<Person> people = Arrays.asList(
            new Person("Dr. Bunsen", "Honeydew"),
            new Person("Gonzo", "The Great"),
            new Person("Phillip J.", "Fry"));
        System.out.println(people);
        Element root = new Element("people");
        for(Person p : people)
            root.appendChild(p.getXML());
        Document doc = new Document(root);
        format(System.out, doc);
        format(new BufferedOutputStream(
            new FileOutputStream("People.xml")), doc);
    }
} /* Output:
```

```
    [Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
    <?xml version=''1.0'' encoding=''ISO-8859-1''?>
    <people>
    <person>
    <first>Dr. Bunsen</first>
    <last>Honeydew</last>
    </person>
    <person>
    <first>Gonzo</first>
    <last>The Great</last>
    </person>
    <person>
    <first>Phillip J.</first>
    <last>Fry</last>
    </person>
    </people>
  *///:~
```

Die XOM-Methoden sind selbsterklärend (siehe API-Dokumentation).

[216] Die XOM-Bibliothek enthält eine Klasse namens `nu.xom.Serializer`, die in der `format()`-Methode angewendet wird, um den XML-Quelltext besser lesbar zu machen. If you just call toXML() you'll get everything run together, so the Serializer is a convenient tool.

[217] Die Deserialisierung der `Person`-Objekte aus einer *.xml* Datei ist genauso einfach:

```
//: xml/People.java
// {Requires: nu.xom.Node; You must install
// the XOM library from http://www.xom.nu }
// {RunFirst: Person}
import nu.xom.*;
import java.util.*;

public class People extends ArrayList<Person> {
    public People(String fileName) throws Exception {
        Document doc = new Builder().build(fileName);
        Elements elements = doc.getRootElement().getChildElements();
        for(int i = 0; i < elements.size(); i++)
            add(new Person(elements.get(i)));
    }
    public static void main(String[] args) throws Exception {
        People p = new People("People.xml");
        System.out.println(p);
    }
} /* Output:
      [Dr. Bunsen Honeydew, Gonzo The Great, Phillip J. Fry]
  *///:~
```

Der Konstruktor der Klasse `People` öffnet und liest eine *.xml* Datei mit Hilfe der `nu.xom.Builder`-Methode `build()` aus der XOM-Bibliothek. Die `getChildElements()`-Methode erzeugt ein listenartiges Objekt vom Typ `nu.xom.Elements`. (Dies ist kein Standard-*`List`*-Objekt, verfügt aber über die Methoden `size()` und `get()`. Harold wollte die Anwender seiner XOM-Bibliothek nicht zwingen, die SE 5 zu verwenden, aber einen typsicheren Container zur Verfügung stellen.) Jedes Element dieser Liste repräsentiert ein `Person`-Objekt und wird daher dem zweiten `Person`-Konstruktor übergeben. Diese Vorgehensweise setzt voraus, daß Sie die exakte Struktur der *.xml* Datei im Voraus kennen. Diese Voraussetzung ist bei derartigen Aufgaben aber in der Regel erfüllt. Entspricht die tatsächliche Struktur nicht Ihrer Annahme, so wird eine Ausnahme ausgeworfen. Es ist aber durchaus möglich auch komplexe Logik zu implementieren, um die XML-Struktur zu untersuchen,

statt Annahmen zu treffen, wenn Sie nur wenige konkrete Anhaltspunkte zu Struktur einer *.xml* Datei zur Verfügung haben.

[218] Sie müssen die *.jar* Archivdatei mit der XOM-Distribution in Ihren Klassenpfad aufnehmen, damit sich die beiden Beispiele übersetzen und ausführen lassen. Soviel als kurze Einführung in die XML-Programmierung unter Java mit der XOM-Bibliothek. Auf der Website *www.xom.nu* finden Sie weitere Informationen.

Übungsaufgabe 31: (2) Ergänzen Sie die Beispiele *Person.java* und *People.java* um Adressen. ■

Übungsaufgabe 32: (4) Schreiben Sie ein Programm, das mit Hilfe eines `Map<String,Integer>`-Objektes und der Hilfsklasse `net.mindview.util.TextFile` die Vorkommen der Wörter in einer Textdatei zählt. Übergeben Sie dem `TextFile`-Konstruktor den regulären Ausdruck \\`W+` als zweites Argument. Speichern Sie die Ergebnisse als *.xml* Datei ab. ■

19.14 Preferences

[219] Preferences liegen konzeptionell näher an der Persistenz, als an der Serialisierung, da die verwalteten Informationen automatisch gespeichert und wiederhergestellt werden. Sie sind allerdings auf vergleichsweise kleine Datensätze beschränkt: Sie können nur Werte primitiven Typs sowie `String`-Objekte verwalten, wobei die von einem `String`-Objekt repräsentierte Zeichenkette nicht länger als 8kB sein darf (nicht gerade winzig, aber die Wertefelder sind ohnehin nicht für komplexe Informationen gedacht). Preferences dienen dazu, benutzerdefinierte Voreinstellungen und anwendungsspezifische Parameter zu speichern beziehungsweise zu laden.

[220] Preferences sind in einer Knotenhierarchie gespeicherte Schlüssel/Wert-Paare (wie `Maps`). Obwohl sich über diese Knotenhierarchie komplizierte Strukturen abbilden lassen, wird typischerweise ein einzelner Knoten nach einer Klasse benannt und alle [für diese Klasse spezifischen] Informationen unterhalb dieses Knotens verwaltet. Ein Beispiel:

```
//: io/PreferencesDemo.java
import java.util.prefs.*;
import static net.mindview.util.Print.*;

public class PreferencesDemo {
    public static void main(String[] args) throws Exception {
        Preferences prefs = Preferences
            .userNodeForPackage(PreferencesDemo.class);
        prefs.put("Location", "Oz");
        prefs.put("Footwear", "Ruby Slippers");
        prefs.putInt("Companions", 4);
        prefs.putBoolean("Are there witches?", true);
        int usageCount = prefs.getInt("UsageCount", 0);
        usageCount++;
        prefs.putInt("UsageCount", usageCount);
        for(String key : prefs.keys())
            print(key + ": "+ prefs.get(key, null));
        // You must always provide a default value:
        print("How many companions does Dorothy have? " +
              prefs.getInt("Companions", 0));
    }
} /* Output: (Sample)
      Location: Oz
      Footwear: Ruby Slippers
      Companions: 4
```

```
    Are there witches?: true
    UsageCount: 53
    How many companions does Dorothy have? 4
*///:~
```

[221] Die Entscheidung für die Methode `userNodeForPackage()` ist willkürlich. Sie können ebenso gut die Methode `systemNodeForPackage()` wählen. Eigentlich soll aber `userNodeForPackage()` für benutzerdefinierte und `systemNodeForPackage()` für „systemweite“ anwendungsbezogene Voreinstellungen gewählt werden. Da die `main()`-Methode statisch ist, verwenden wir das Klassenliteral `PreferencesDemo.class`, um den Knoten zu identifizieren. Im Körper einer nicht-statischen Methode rufen Sie dagegen in der Regel die `getClass()`-Methode auf. Sie müssen den Namen der Klasse nicht als Knotennamen wählen, aber es ist so üblich.

[222] Ein erzeugter Knoten ist bereit, um Daten zu speichern oder von dort abzufragen. Das obige Beispiel lädt den Knoten mit seinen unterschiedlichen Schlüssel/Wert-Paaren und fordert die Schlüsselmenge an. Diese wird als `String`-Array zurückgegeben, verglichen mit den `keys()`-Methoden bei Kollektionen ein ungewöhnliches Rückgabeformat. Beachten Sie das zweite Argument der verschiedenen Abfragemethoden: Dies ist der von der Abfragemethode zurückgegebene Standardwert, falls zu diesem Schlüssel noch kein Wert vorhanden ist. ~~While iterating through a set of keys, you always know there's an entry, so using null as the default is safe, but normally you'll be fetching a named key, as in:~~

```
prefs.getInt("Companions", 0));
```

In der Regel geben Sie einen sinnvollen Standardwert vor. Eine typische Abfolge von Anwendungen ist:

```
int usageCount = prefs.getInt("UsageCount", 0);
usageCount++;
prefs.putInt("UsageCount", usageCount);
```

Auf dieser Weise hat `UsageCount` beim ersten Programmstart den Wert 0 und anschließend 1, 2 und so weiter.

[223] Beim wiederholten Ausführen des Programms *Preferences.java* werden Sie tatsächlich feststellen, daß `UsageCount` pro Aufruf um 1 erhöht wird. Wo aber wird der Zählerstand gespeichert? Nach dem ersten Programmstart existiert keine lokale Datei. Die ~~Preferences~~ werden an einer vom unterliegenden Betriebssystem abhängigen Stelle gespeichert. Bei Windowssystemen ist dies die Windows-Registratur (die bereits eine hierarchische Struktur von Schlüssel/Wert-Paaren ist). Der Knackpunkt besteht gerade darin, daß die Information wie von Zauberhand gespeichert wird, so daß Sie sich keine Gedanken darüber machen müssen, wie Ihre Laufzeitumgebung Ihre ~~Preferences~~ speichert.

[224] Die Funktionalität der ~~Preferences~~-API geht über den hier dargestellten Umfang hinaus. Konsultieren Sie zu weiteren Einzelheiten die Dokumentation des Java Development Kits.

Übungsaufgabe 33: (2) Schreiben Sie ein Programm, welches den aktuellen Wert des Schlüssels „base directory“ abfragt und vom Benutzer einen neuen Wert erwartet. Lesen Sie in der API-Dokumentation nach, um den neuen Wert zu sichern. ■

19.15 Zusammenfassung

[225] Die Ein-/Ausgabebibliothek von Java erfüllt die grundlegenden Anforderungen: Sie können Daten an die Konsole, Dateien, einen Block im Arbeitsspeicher und sogar über ein Netzwerk hin-

weg schreiben beziehungsweise von dort lesen. Sie können per Ableitung neue Typen von Ein-/Ausgabeobjekten erzeugen. Sie können die Klassen deren Objekte, einem Ausgabestrom übergeben werden sollen, ~~add simple extensibility~~, indem Sie die `toString()`-Methode überschreiben, die automatisch aufgerufen wird, wenn Sie einer Methode ein Objekt übergeben, die einen `String`-Parameter erwartet (beschränkte „automatische Typumwandlung“ bei Java).

[226] Es gibt Fragen, die weder von der Dokumentation noch vom Design der Ein-/Ausgabebibliothek beantwortet werden. Es wäre beispielsweise sinnvoll, eine Ausnahme auswerfen zu können, wenn eine bereits existierende Datei zum Schreiben geöffnet wird. Es gibt Programmiersprachen, die Ihnen ermöglichen, eine Datei nur dann zum Schreiben zu öffnen, wenn sie nicht bereits vorhanden ist. Bei Java müssen Sie ein `File`-Objekt verwenden, um festzustellen, ob eine Datei existiert, da sie beim Öffnen per `FileOutputStream`- oder `FileWriter`-Objekt stets überschrieben wird.

[227] Die Ein-/Ausgabebibliothek bewirkt gemischte Gefühle. Einerseits erfüllt sie ihren Zweck und ist portabel. Andererseits ist die Bibliothek nicht intuitiv anwendbar, ohne das *Decorator*-Entwurfsmuster verstanden zu haben, so daß zunächst ein zusätzlicher Lernaufwand erforderlich ist. Die Ein-/Ausgabebibliothek ist außerdem unvollständig. Hilfsklassen wie `TextFile` sollten eigentlich nicht nötig sein (die überarbeitete Klasse `PrintWriter` ist ein Schritt in die richtige Richtung, aber nur eine Teillösung). Die SE 5 beinhaltet eine erhebliche Verbesserung: Sun Microsystems hat endlich die formatierte Ausgabe in den Sprachumfang integriert, die von nahezu jeder anderen Programmiersprache unterstützt wird.

[228] Wenn Sie das *Decorator*-Entwurfsmuster erst einmal verinnerlicht haben und die Ein-/Ausgabebibliothek in Situationen einsetzen, in denen ihre Flexibilität erforderlich ist, werden Sie von diesem Design profitieren und der Aufwand einiger zusätzlicher Zeilen Quelltext stört nicht mehr.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 20

Aufzählungstypen

Inhaltsübersicht

[0] Das Schlüsselwort `enum` gestattet Ihnen, einen neuen Typ mit einer begrenzten Anzahl benannter Wert zu definieren und diese Werte als gewöhnliche Programmkomponenten zu verwenden. Aufzählungstypen („enumerierte Typen“) erweisen sich als sehr nützlich.[1]

[1] Aufzählungstypen wurden bereits in Abschnitt 6.9 kurz vorgestellt. Nachdem Sie nun auch einige tiefere Eigenschaften und Fähigkeiten von Java verstehen, können wir die Einzelheiten der seit Version 5 der Java Standard Edition (SE 5) verfügbaren Aufzählungstypen betrachten. Sie werden sehen, daß diese neuen Komponenten einige sehr interessante Möglichkeiten eröffnen. Dieses Kapitel liefert aber auch Einblicke in andere Spracheigenschaften, die Sie bereits kennengelernt haben, darunter generische Typen und den Reflexionsmechanismus. Außerdem werden einige neue Entwurfsmuster vorgestellt.

[1] Joshua Bloch hat mir beim Schreiben dieses Kapitels sehr geholfen.

20.1 Grundlegende Eigenschaften von Aufzählungstypen

[2] Sie haben in Abschnitt 6.9 gelernt, daß Sie mit Hilfe der `values()`-Methode eines Aufzählungstyps dessen konstante Werte der Reihe nach verarbeiten können. Die `values()`-Methode gibt ein Array zurück, welches die Werte des Aufzählungstyps in der Reihenfolge ihrer Deklaration enthält, so daß Sie den Rückgabewert beispielsweise in die erweiterte `for`-Schleife einsetzen können.

[3] Der Compiler erzeugt zu jedem Aufzählungstyp, den Sie definieren, eine Klasse. Diese Klasse ist automatisch von `java.lang.Enum` abgeleitet und stellt die im folgenden Beispiel demonstrierte Funktionalität zur Verfügung:

```
//: enumerated/EnumClass.java
// Capabilities of the Enum class
import static net.mindview.util.Print.*;

enum Shrubbery { GROUND, CRAWLING, HANGING }

public class EnumClass {
    public static void main(String[] args) {
        for(Shrubbery s : Shrubbery.values()) {
            print(s + " ordinal: " + s.ordinal());
            printnb(s.compareTo(Shrubbery.CRAWLING) + " ");
            printnb(s.equals(Shrubbery.CRAWLING) + " ");
            print(s == Shrubbery.CRAWLING);
            print(s.getDeclaringClass());
            print(s.name());
            print("-----------");
        }
        // Produce an enum value from a string name:
        for(String s : "HANGING CRAWLING GROUND".split(" ")) {
            Shrubbery shrub = Enum.valueOf(Shrubbery.class, s);
            print(shrub);
        }
    }
} /* Output:
      GROUND ordinal: 0
      -1 false false
      class Shrubbery
      GROUND
      -----------
      CRAWLING ordinal: 1
      0 true true
      class Shrubbery
      CRAWLING
      -----------
      HANGING ordinal: 2
      1 false false
      class Shrubbery
      HANGING
      -----------
      HANGING
      CRAWLING
      GROUND
   *///:~
```

Die Methode `ordinal()` gibt einen `int`-Wert („Ordinalwert“) zurück, der die Position der Konstanten bezüglich der deklarierten Reihenfolge bezeichnet (beginnend bei Null). Sie können die Werte eines Aufzählungstyps stets per `==`-Operator miteinander vergleichen. Die Methoden `equals()` und

`hashCode()` werden automatisch implementiert. Die Klasse `Enum` implementiert die Interfaces *`Serializable`* und *`Comparable`*, definiert also eine `compareTo()`-Methode.

[4] Die Methode `getDeclaringClass()` gibt, auf einer Konstanten aufgerufen, den Namen des Aufzählungstyps zurück.

[5] Die `name()`-Methode liefert, ebenso wie die Methode `toString()`, den deklarierten Namen einer Konstanten. Die statische `Enum`-Methode `valueOf()` erwartet ein Argument vom Typ `String` und gibt die Konstante mit dem entsprechenden Namen zurück oder wirft eine Ausnahme aus, wenn es keine passende Konstante gibt.

20.1.1 Statisches Importieren von Aufzählungstypen

[6] Das folgende Beispiel ist eine Variante des *Burrito.java*-Beispiels auf Seite 168 in Abschnitt 6.9:

```
//: enumerated/Spiciness.java
package enumerated;

public enum Spiciness {
    NOT, MILD, MEDIUM, HOT, FLAMING
} ///:~

//: enumerated/Burrito.java
package enumerated;
import static enumerated.Spiciness.*;

public class Burrito {
    Spiciness degree;
    public Burrito(Spiciness degree) { this.degree = degree;}
    public String toString() { return "Burrito is " + degree;}
    public static void main(String[] args) {
        System.out.println(new Burrito(NOT));
        System.out.println(new Burrito(MEDIUM));
        System.out.println(new Burrito(HOT));
    }
} /* Output:
      Burrito is NOT
      Burrito is MEDIUM
      Burrito is HOT
   *////:~
```

Der statische Import (beachten Sie die umgekehrte Reihenfolge: `import static`) integriert die Konstanten des Aufzählungstyps `Spiciness` in den lokalen Namensraum, so daß ihre Namen nicht qualifiziert zu werden brauchen. Ist das statische Importieren eine gute Idee oder wäre es besser, die Namen der Werte des Aufzählungstyps ausdrücklich voll zu qualifizieren? Das hängt wohl von der Komplexität des Quelltextes ab. Der Compiler erlaubt Ihnen mit Sicherheit nicht, den falschen Typ zu verwenden, so daß Ihre Bedenken nur der Frage gelten sollten, ob Ihr Quelltext den Leser verwirrt. Das statische Importieren eines Aufzählungstyps ist in vielen Situationen in Ordnung, Sie sollten sich aber je nach individueller Sachlage entscheiden.

[7] Beachten Sie, daß statisches Importieren nicht möglich ist, wenn der Aufzählungstyp in derselben Datei definiert ist oder unter Packagezugriff steht. (Anscheinend wurde die Frage, ob man das statische Importieren in diesem Fall zulassen sollte, bei Sun Microsystems diskutiert.)

20.2 Erweitern eines Aufzählungstyps um eigene Methoden

[8] Abgesehen von der Tatsache, daß Sie von einem Aufzählungstyp keine Klasse abgeleiten können, können Sie einen Aufzählungstyp wie eine gewöhnliche Klasse behandeln. Sie können insbesondere Methoden anlegen. Ein Aufzählungstyp kann sogar eine `main()`-Methode besitzen.

[9] Eventuell möchten Sie einen anderen Beschreibungstext für die Konstanten eines Aufzählungstyps ausgeben, als den voreingestellten Rückgabewert der `toString()`-Methode, also nur den Namen der Konstanten. Sie können zu diesem Zweck einen eigenen Konstruktor anlegen, der die zusätzlichen Informationen erfaßt und eigene Methoden schreiben, um den erweiterten Beschreibungstext auszugeben:

```
//: enumerated/OzWitch.java
// The witches in the land of Oz.
import static net.mindview.util.Print.*;

public enum OzWitch {
    // Instances must be defined first, before methods:
    WEST(''Miss Gulch, aka the Wicked Witch of the West''),
    NORTH(''Glinda, the Good Witch of the North''),
    EAST(''Wicked Witch of the East, wearer of the Ruby '' +
         ''Slippers, crushed by Dorothy's house''),
    SOUTH(''Good by inference, but missing''); // <--- Note semicolon
    private String description;
    // Constructor must be package or private access:
    private OzWitch(String description) {
        this.description = description;
    }
    public String getDescription() { return description; }
    public static void main(String[] args) {
        for(OzWitch witch : OzWitch.values())
            print(witch + '': '' + witch.getDescription());
    }
} /* Output:
      WEST: Miss Gulch, aka the Wicked Witch of the West
      NORTH: Glinda, the Good Witch of the North
      EAST: Wicked Witch of the East, wearer of the Ruby Slippers,
        crushed by Dorothy's house
      SOUTH: Good by inference, but missing
   *///:~
```

Beachten Sie, daß Sie die Aufzählung der Konstanten mit einem Semikolon (`;`) abschließen müssen, wenn Sie eigene Methoden oder Felder anlegen. Java erzwingt, daß die Konstanten zuerst definiert werden. Der Compiler meldet einen Fehler, wenn Sie versuchen, die Konstanten nach den Methoden oder Feldern zu definieren.

[10] Konstruktor und Methoden haben dasselbe Format wie bei einer gewöhnlichen Klasse, da ein Aufzählungstyp, abgesehen von einigen Einschränkungen, eine gewöhnliche Klasse *ist*. Sie können die Funktionalität eines Aufzählungstyps also fast beliebig erweitern. Im allgemeinen werden Aufzählungstypen aber auf einem einfachen Niveau belassen.

[11] Der Konstruktor im obigen Beispiel ist zwar als privat definiert, aber es kommt nicht darauf an, welchen Zugriff Sie festlegen. Der Konstruktor eines Aufzählungstyps kann nur dazu verwendet werden, um die Objekte zu den deklarierten Konstanten in der Definition zu erzeugen. Der Compiler erlaubt dagegen nicht, daß Sie den Konstruktor verwenden, um neue, über die Definition des Aufzählungstyps hinausgehende Konstanten zu erzeugen.

20.2.1 Überschreiben von Methoden der Klasse Enum

[12] Das nächste Beispiel zeigt eine weitere Möglichkeit, um die `String`-Darstellung der Konstanten eines Aufzählungstyps zu ändern. Wir bleiben diesesmal bei den Namen der Konstanten, formatieren aber ihre Darstellungsweise um. Das Überschreiben der `toString()`-Methode eines Aufzählungstyps entspricht dem Überschreiben einer Methode einer gewöhnlichen Klasse:

```
//: enumerated/SpaceShip.java
public enum SpaceShip {
    SCOUT, CARGO, TRANSPORT, CRUISER, BATTLESHIP, MOTHERSHIP;
    public String toString() {
        String id = name();
        String lower = id.substring(1).toLowerCase();
        return id.charAt(0) + lower;
    }
    public static void main(String[] args) {
        for(SpaceShip s : values()) {
            System.out.println(s);
        }
    }
} /* Output:
      Scout
      Cargo
      Transport
      Cruiser
      Battleship
      Mothership
   *///:~
```

Die `toString()`-Methode fragt per `name()` den Namen der jeweiligen `SpaceShip`-Konstanten ab und modifiziert das Ergebnis so, daß nur der erste Buchstabe als Großbuchstabe erscheint.

20.3 Aufzählungstypen in switch-Anweisungen

[13] Aufzählungstypen lassen sich auf eine sehr praktische Weise mit `switch`-Anweisungen kombinieren. Eine `switch`-Anweisung funktioniert für gewöhnlich nur mit einem ganzzahligen Wert. Da Aufzählungstypen ihren Konstanten einen ganzzahligen Ordinalwert zuordnen, der mit Hilfe der `ordinal()`-Methode abgefragt werden kann (der Compiler scheint diese Möglichkeit zu nutzen), können die Konstanten in die `case`-Zweige einer `switch`-Anweisung eingesetzt werden.

[14] Die in der Regel notwendige Qualifizierung der Konstanten durch den Namen des Aufzählungstyps entfällt bei der Verwendung in einer `switch`-Anweisung. Das folgende Beispiel implementiert einen kleinen endlichen Automaten:

```
//: enumerated/TrafficLight.java
// Enums in switch statements.
import static net.mindview.util.Print.*;

// Define an enum type:
enum Signal { GREEN, YELLOW, RED, }

public class TrafficLight {
   Signal color = Signal.RED;
   public void change() {
       switch(color) {
           // Note that you don't have to say Signal.RED
```

```
            // in the case statement:
            case RED:
                color = Signal.GREEN;
                break;
            case GREEN:
                color = Signal.YELLOW;
                break;
            case YELLOW:
                color = Signal.RED;
                break;
        }
    }
    public String toString() {
        return "The traffic light is " + color;
    }
    public static void main(String[] args) {
        TrafficLight t = new TrafficLight();
        for(int i = 0; i < 7; i++) {
            print(t);
            t.change();
        }
    }
} /* Output:
      The traffic light is RED
      The traffic light is GREEN
      The traffic light is YELLOW
      The traffic light is RED
      The traffic light is GREEN
      The traffic light is YELLOW
      The traffic light is RED
   *///:~
```

Der Compiler beschwert sich nicht, daß die `switch`-Anweisung keinen `default`-Zweig hat. Das hat allerdings *nichts* damit zu tun, daß der Compiler erkennt, daß Sie zu jeder Konstanten des Aufzählungstyps `Signal` einen `case`-Zweig angelegt haben. Der Compiler beschwert sich auch nicht, wenn Sie einen `case`-Zweig auskommentieren, das heißt Sie müssen selbst darauf achten, alle Fälle abzudecken. Enthält ein `case`-Zweig aber eine `return`-Anweisung, so *wird* sich der Compiler beschweren, wenn kein `default`-Zweig existiert, selbst wenn Sie alle Konstanten des Aufzählungstyps erfaßt haben.

Übungsaufgabe 1: (2) Legen Sie im Beispiel *TrafficLight.java* eine statische Importanweisung an, damit die einzelnen Konstanten des Aufzählungstyps nicht mehr qualifiziert angegeben werden müssen. ■

20.4 Die Herkunft der values()-Methode

[15] Wie bereits in Abschnitt 20.1 festgestellt, erzeugt der Compiler zu jedem Aufzählungstyp eine von `Enum` abgeleitete Klasse. Wenn Sie die Klasse `Enum` untersuchen, werden Sie allerdings beobachten, daß dort keine `values()`-Methode definiert wird, obwohl wir sie bereits verwendet haben. Gibt es noch weitere „verborgene“ Methoden? Das folgende kleine Programm stützt sich auf den Reflexionsmechanismus, um diese Frage zu beantworten:

```
//: enumerated/Reflection.java
// Analyzing enums using reflection.
```

```
import java.lang.reflect.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

enum Explore { HERE, THERE }

public class Reflection {
    public static Set<String> analyze(Class<?> enumClass) {
        print("--- Analyzing " + enumClass + " ---");
        print("Interfaces:");
        for(Type t : enumClass.getGenericInterfaces())
            print(t);
        print("Base: " + enumClass.getSuperclass());
        print("Methods: ");
        Set<String> methods = new TreeSet<String>();
        for(Method m : enumClass.getMethods())
            methods.add(m.getName());
        print(methods);
        return methods;
    }
    public static void main(String[] args) {
        Set<String> exploreMethods = analyze(Explore.class);
        Set<String> enumMethods = analyze(Enum.class);
        print("Explore.containsAll(Enum)? " +
              exploreMethods.containsAll(enumMethods));
        printnb("Explore.removeAll(Enum): ");
        exploreMethods.removeAll(enumMethods);
        print(exploreMethods);
        // Decompile the code for the enum:
        OSExecute.command("javap Explore");
    }
} /* Output:
      --- Analyzing class Explore ---
      Interfaces:
      Base: class java.lang.Enum
      Methods:
      [compareTo, equals, getClass, getDeclaringClass, hashCode, name, notify,
        notifyAll, ordinal, toString, valueOf, values, wait]
      --- Analyzing class java.lang.Enum ---
      Interfaces:
      java.lang.Comparable<E>
      interface java.io.Serializable
      Base: class java.lang.Object
      Methods:
      [compareTo, equals, getClass, getDeclaringClass, hashCode, name, notify,
        notifyAll, ordinal, toString, valueOf, wait]
      Explore.containsAll(Enum)? true
      Explore.removeAll(Enum): [values]
      Compiled from "Reflection.java"
      final class Explore extends java.lang.Enum{
      public static final Explore HERE;
      public static final Explore THERE;
      public static final Explore[] values();
      public static Explore valueOf(java.lang.String);
      static {};
      }
   *///:~
```

Die `values()`-Methode ist also eine statische Methode und wird vom Compiler angelegt. Sie sehen, daß auch die Methode `valueOf()` beim Erzeugen der Klasse für den Aufzählungstyp `Explore` angelegt wird. Das ist ein wenig verwirrend, weil die Klasse `Enum` bereits eine `valueOf()`-Methode mit zwei Argumenten definiert, während die vom Compiler erzeugte Version nur ein Argument erwartet. Da die `analyze()`-Methode nur auf Methodennamen achtet, nicht aber auf Signaturen, bleibt nach dem Aufruf „`Explore.removeAll(Enum)`“ nur die Methode `values()` übrig.

[16] Sie können an der Ausgabe ablesen, daß der Compiler die Klasse `Explore` als final definiert, so daß Sie von einem Aufzählungstyp keine Klasse ableiten können. Die vom Compiler generierte Klasse enthält außerdem einen statischen Initialisierungsblock, der aber umdefiniert werden kann (siehe später in diesem Kapitel // Buch: Noch nicht gefunden).

[17] Aufgrund der Typauslöschung (siehe Abschnitt 16.7) hat der Decompiler nicht alle Informationen über `Enum` zur Verfügung und zeigt die Basisklasse von `Explore` nur als den raw type `Enum` an, statt des eigentlichen Typs `Enum<Explore>`.

[18] Da der Compiler die statische Methode `values()` in die Definition des Aufzählungstyps einsetzt, steht die `values()`-Methode nach der Typumwandlung der entsprechenden Referenz in `Enum` nicht mehr zur Verfügung. Beachten Sie aber die `Class`-Methode `getEnumConstants()`. Auch wenn die `values()`-Methode nicht zur Schnittstelle der Klasse `Enum` gehört, können Sie die einzelnen Konstanten des Aufzählungsobjektes über das Klassenobjekt erreichen:

```
//: enumerated/UpcastEnum.java
// No values() method if you upcast an enum

enum Search { HITHER, YON }

public class UpcastEnum {
    public static void main(String[] args) {
        Search[] vals = Search.values();
        Enum e = Search.HITHER; // Upcast
        // e.values(); // No values() in Enum
        for(Enum en : e.getClass().getEnumConstants())
            System.out.println(en);
    }
} /* Output:
      HITHER
      YON
   *///:~
```

Da `getEnumConstants()` eine Methode des Klassenobjektes ist, können Sie sie auch bezüglich einer Klasse aufrufen, die keinen Aufzählungstyp repräsentiert:

```
//: enumerated/NonEnum.java
public class NonEnum {
    public static void main(String[] args) {
        Class<Integer> intClass = Integer.class;
        try {
            for(Object en : intClass.getEnumConstants())
                System.out.println(en);
        } catch(Exception e) {
            System.out.println(e);
        }
    }
} /* Output:
      java.lang.NullPointerException
   *///:~
```

Die Methode gibt `null` zurück, so daß eine Ausnahme hervorgerufen wird, wenn Sie den Rückgabewert verarbeiten.

20.5 Aufzählungstypen können Interfaces implementieren

[19] Ein Aufzählungstyp ist stets von der Klasse `java.lang.Enum` abgeleitet. Da Java keine Mehrfachvererbung unterstützt, können Sie einen Aufzählungstyp nicht von einer (weiteren) Klasse ableiten:

```
enum NotPossible extends Pet { ... // Won't work
```

Es *ist* dagegen möglich, daß ein Aufzählungstyp eines oder mehrere Interfaces implementiert:

```
//: enumerated/cartoons/EnumImplementation.java
// An enum can implement an interface
package enumerated.cartoons;
import java.util.*;
import net.mindview.util.*;

enum CartoonCharacter implements Generator<CartoonCharacter> {
    SLAPPY, SPANKY, PUNCHY, SILLY, BOUNCY, NUTTY, BOB;
    private Random rand = new Random(47);
    public CartoonCharacter next() {
        return values()[rand.nextInt(values().length)];
    }
}

public class EnumImplementation {
    public static <T> void printNext(Generator<T> rg) {
        System.out.print(rg.next() + ", ");
    }
    public static void main(String[] args) {
        // Choose any instance:
        CartoonCharacter cc = CartoonCharacter.BOB;
        for(int i = 0; i < 10; i++)
            printNext(cc);
    }
} /* Output:
      BOB, PUNCHY, BOB, SPANKY, NUTTY, PUNCHY, SLAPPY, NUTTY, NUTTY, SLAPPY,
   *///:~
```

Das Ergebnis ist ein wenig sonderbar, da Sie eine Konstante des Aufzählungstyps brauchen, um eine Methode des Aufzählungstyps aufrufen zu können. Andererseits können Sie nun jeder Methode die ein Argument vom Typ *`Generator`* erwartet (beispielsweise `printNext()`), eine Konstante des Aufzählungstyps `CartoonCharacter` übergeben.

Übungsaufgabe 2: (2) Ändern Sie `next()` in eine statische Methode um, statt das Interface *`Generator`* zu implementieren. Welche Vor- und Nachteile hat dieser Ansatz? ■

20.6 Zufällige Auswahl von Konstanten

[20] Viele Beispiele in diesem Kapitel verlangen zufällig ausgewählte Konstanten von Aufzählungstypen, wie bei der `next()`-Methode der Klasse `CartoonCharacter` im vorigen Abschnitt. Wir verallgemeinern diese Aufgabe nun mit Hilfe generischer Typen und deponieren das Ergebnis im Package

`net.mindview.util`:

```
//: net/mindview/util/Enums.java
package net.mindview.util;
import java.util.*;

public class Enums {
    private static Random rand = new Random(47);
    public static <T extends Enum<T>> T random(Class<T> ec) {
        return random(ec.getEnumConstants());
    }
    public static <T> T random(T[] values) {
        return values[rand.nextInt(values.length)];
    }
} ///:~
```

Die reichlich sonderbare Syntax `<T extends Enum<T>>` drückt aus, daß `T` eine Konstante eines Aufzählungstyps ist. Durch das Klassenobjekt (Argument vom Typ `Class<T>`) können die Konstanten des Aufzählungstyps abgefragt werden. Die überladene Methode `random()` braucht lediglich zu „wissen“, daß sie ein Array vom Typ `T` erhält, da sie keine Methoden aus der Klasse `Enum` benötigt. Die Methode gibt lediglich ein zufällig ausgewähltes Arrayelement zurück. Der Rückgabetyp ist der exakte Typ der Konstanten.

[21] Das folgende Programm ist ein einfacher Test für die `random()`-Methode:

```
//: enumerated/RandomTest.java
import net.mindview.util.*;

enum Activity { SITTING, LYING, STANDING, HOPPING,
                RUNNING, DODGING, JUMPING, FALLING, FLYING }

public class RandomTest {
    public static void main(String[] args) {
        for(int i = 0; i < 20; i++)
            System.out.print(Enums.random(Activity.class) + " ");
    }
} /* Output:
      STANDING FLYING RUNNING STANDING RUNNING STANDING LYING DODGING SITTING
        RUNNING HOPPING HOPPING HOPPING RUNNING STANDING LYING FALLING RUNNING
        FLYING LYING
   *///:~
```

Die Klasse `Enum` ist zwar nur eine kleine Klasse, verhindert aber, wie Sie in diesem Kapitel noch sehen werden, einen beträchtlichen Umfang an dupliziertem Quelltext. Das Duplizieren von Quelltext ist eine Fehlerquelle und sollte daher stets angestrebt werden.

20.7 Definition von Kategorien per Interface

[22] Es ist manchmal ärgerlich, daß Sie keine neue Klasse von einem Aufzählungstyp ableiten können. Die Motivation dazu stammt einerseits von dem Wunsch, die Anzahl der Konstanten des ursprünglichen Aufzählungstyps zu erweitern, und andererseits von dem Wunsch, durch Untertypen Unterkategorien bilden zu können.

[23] Sie können eine Kategorisierung dadurch erreichen, daß Sie mehrere Aufzählungstypen/Elemente in einem Interface gruppieren und einen neue Aufzählungstyp anlegen, der dieses Interface implementiert. Stellen Sie zum Beispiel die Aufteilung von Lebensmitteln in verschiedene Gruppen vor, wobei Sie

jede Gruppe als Aufzählungstyp darstellen möchten, jedes einzelne Lebensmittel aber auch zum Typ *Food* gehören soll:

```
//: enumerated/menu/Food.java
// Subcategorization of enums within interfaces.
package enumerated.menu;

public interface Food {
    enum Appetizer implements Food {
        SALAD, SOUP, SPRING_ROLLS;
    }
    enum MainCourse implements Food {
        LASAGNE, BURRITO, PAD_THAI,
            LENTILS, HUMMOUS, VINDALOO;
    }
    enum Dessert implements Food {
        TIRAMISU, GELATO, BLACK_FOREST_CAKE,
            FRUIT, CREME_CARAMEL;
    }
    enum Coffee implements Food {
        BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
            LATTE, CAPPUCCINO, TEA, HERB_TEA;
    }
} ///:~
```

Da das Implementieren von Interfaces die einzige Möglichkeit ist, einen Aufzählungstyp von einem Basistyp „abzuleiten“, implementiert jeder geschachtelte Aufzählungstyp das umgebende Interface *Food*. Damit hat jedes Lebensmittel den Typ *Food*, wie die folgenden Zeilen belegen:

```
//: enumerated/menu/TypeOfFood.java
package enumerated.menu;
import static enumerated.menu.Food.*;

public class TypeOfFood {
    public static void main(String[] args) {
        Food food = Appetizer.SALAD;
        food = MainCourse.LASAGNE;
        food = Dessert.GELATO;
        food = Coffee.CAPPUCCINO;
    }
} ///:~
```

Die aufwärts gerichtete Typumwandlung in *Food* funktioniert bei jedem Aufzählungstyp, der das Interface *Food* implementiert, so daß alle Konstanten vom Typ *Food* sind.

[24] Ein Interface ist allerdings weniger nützlich als ein Aufzählungstyp, wenn Sie einer Menge von Typen operieren wollen. Wenn Sie einen “ Aufzählungstyp von Aufzählungstypen“ brauchen, können Sie einen äußeren Aufzählungstyp anlegen, der eine Konstante für jeden Aufzählungstyp im Interface *Food* definiert:

```
//: enumerated/menu/Course.java
package enumerated.menu;
import net.mindview.util.*;

public enum Course {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
```

```
    private Course(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
} ///:~
```

Jeder der „inneren“ Aufzählungstypen `APPETIZER`, `MAINCOURSE`, `DESSERT` und `COFFEE` erwartet das korrespondierende Klassenobjekt als Argument des Konstruktors. Die die Konstanten jedes „inneren“ Aufzählungstyp werden mit Hilfe der Methode `getEnumConstants()` ausgewertet und gespeichert. Die Methode `randomSelection()` gibt anschließend einzelne Konstanten zurück, so daß wir zufällig kombinierte Mahlzeiten zusammenstellen können, indem wir eine *`Food`*-Konstante aus jeder Lebensmittelgruppe (`Course`) auswählen:

```
//: enumerated/menu/Meal.java
package enumerated.menu;

public class Meal {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Course course : Course.values()) {
                Food food = course.randomSelection();
                System.out.println(food);
            }
            System.out.println("--");
        }
    }
} /* Output:
      SPRING_ROLLS
      VINDALOO
      FRUIT
      DECAF_COFFEE
      --
      SOUP
      VINDALOO
      FRUIT
      TEA
      --
      SALAD
      BURRITO
      FRUIT
      TEA
      --
      SALAD
      BURRITO
      CREME_CARAMEL
      LATTE
      --
      SOUP
      BURRITO
      TIRAMISU
      ESPRESSO
      --
   *///:~
```

Der Wert des Aufzählungstyps von Aufzählungstypen liegt in diesem Beispiel darin, to iterate through each Course. Das Beispiel *VendingMachine.java* auf Seite 805 zeigt einen weiteren Ansatz

zur Kategorisierung, der von anderen Randbedingungen abhängt.

[25] Ein kompakterer Ansatz für das Kategorisierungsproblem ist die Schachtelung von Aufzählungstypen in der Definition eines Aufzählungstyps:

```
//: enumerated/SecurityCategory.java
// More succinct subcategorization of enums.
import net.mindview.util.*;

enum SecurityCategory {
    STOCK(Security.Stock.class), BOND(Security.Bond.class);
    Security[] values;
    SecurityCategory(Class<? extends Security> kind) {
        values = kind.getEnumConstants();
    }
    interface Security {
        enum Stock implements Security { SHORT, LONG, MARGIN }
        enum Bond implements Security { MUNICIPAL, JUNK }
    }
    public Security randomSelection() {
        return Enums.random(values);
    }
    public static void main(String[] args) {
        for(int i = 0; i < 10; i++) {
            SecurityCategory category = Enums.random(SecurityCategory.class);
            System.out.println(category + ": " + category.randomSelection());
        }
    }
} /* Output:
      BOND: MUNICIPAL
      BOND: MUNICIPAL
      STOCK: MARGIN
      STOCK: MARGIN
      BOND: JUNK
      STOCK: SHORT
      STOCK: LONG
      STOCK: LONG
      BOND: MUNICIPAL
      BOND: JUNK
   *///:~
```

Das Interface *Security* ist notwendig, um die inneren Aufzählungstypen `Stock` und `Bond` zu einem gemeinsamen Typ zusammenzufassen. Die inneren Konstanten werden unter den äußeren Konstanten gruppiert.

[26] Dieser Ansatz liefert, auf das Lebensmittelbeispiel angewendet, folgendes Ergebnis:

```
//: enumerated/menu/Meal2.java
package enumerated.menu;
import net.mindview.util.*;

public enum Meal2 {
    APPETIZER(Food.Appetizer.class),
    MAINCOURSE(Food.MainCourse.class),
    DESSERT(Food.Dessert.class),
    COFFEE(Food.Coffee.class);
    private Food[] values;
    private Meal2(Class<? extends Food> kind) {
        values = kind.getEnumConstants();
```

```
    }
    public interface Food {
        enum Appetizer implements Food {
            SALAD, SOUP, SPRING_ROLLS;
        }
        enum MainCourse implements Food {
            LASAGNE, BURRITO, PAD_THAI,
                LENTILS, HUMMOUS, VINDALOO;
        }
        enum Dessert implements Food {
            TIRAMISU, GELATO, BLACK_FOREST_CAKE,
                FRUIT, CREME_CARAMEL;
        }
        enum Coffee implements Food {
            BLACK_COFFEE, DECAF_COFFEE, ESPRESSO,
                LATTE, CAPPUCCINO, TEA, HERB_TEA;
        }
    }
    public Food randomSelection() {
        return Enums.random(values);
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++) {
            for(Meal2 meal : Meal2.values()) {
                Food food = meal.randomSelection();
                System.out.println(food);
            }
            System.out.println("--");
        }
    }
} /* Same output as Meal.java *////:~
```

Letztendlich ist es nur eine andere Anordnung des Quelltextes, kann aber die Struktur in einigen Fällen deutlicher hervortreten lassen.

Übungsaufgabe 3: (1) Ergänzen Sie das Beispiel *Course.java* (Seite 791) um einen weiteren Gang (`Course`) und zeigen Sie mit Hilfe des Beispiels *Meal.java* (Seite 792), daß der neue Gang „aufgetragen“ wird. ■

Übungsaufgabe 4: (1) Wiederholen Sie Übungsaufgabe 3 mit dem umstrukturierten Beispiel *Meal2.java* (Seite 793). ■

Übungsaufgabe 5: (4) Ändern Sie das Programm *VowelsAndConsonants.java* aus Abschnitt 5.8 (Seite 124), so daß es einen Aufzählungstyp mit drei Konstanten verwendet: `VOWEL`, `SOMETIMES_A_VOWEL` und `CONSONANT`. Der Konstruktor erwartet die verschiedenen Buchstaben der jeweiligen Kategorie. Tip: Verwenden Sie eine Parameterliste variabler Länge und denken Sie daran, daß eine solche Argumentliste automatisch ein Array erzeugt. ■

Übungsaufgabe 6: (3) Hat die Schachtelung der Aufzählungstypen `Appetizer`, `MainCourse`, `Dessert` und `Coffee` im Interface *`Food`* (Seite 791) einen speziellen Vorteil, außer daß sie sonst eigenständige Aufzählungstypen wären, die *`Food`* implementieren? ■

20.8 Die Klasse EnumSet ersetzt Bitvektoren

[27] Ein Container vom Typ *Set* repräsentiert eine Kollektion, die von jeder Sorte von Elementen höchstens ein Exemplar aufnimmt. Die Konstanten eines Aufzählungstyps müssen offensichtlich eindeutig sein, so daß Aufzählungstypen ein *Set*-ähnliches Verhalten haben. Da Sie bei einem Aufzählungstyp aber weder neue Konstanten ergänzen, noch vorhandene Konstanten entfernen können, taugen Aufzählungstypen nicht als *Set*-ähnliche Container. Die Klasse `EnumSet` wurde in der SE 5 aufgenommen, um in Kombination mit Aufzählungstypen die traditionellen `int`-basierten „Bit-Flags“ zu ersetzen. Solche Flags repräsentieren eine binärwertige Information (zum Beispiel „ein“/„aus“), wobei Sie Bits anstelle von Konzepten manipulieren, so daß leicht verwirrender Quelltext entsteht.

[28] Das Design der Klasse `EnumSet` ist geschwindigkeitsorientiert, da sie sich effektiv gegen Bit-Flags durchsetzen muß (die Operationen sind typischerweise erheblich schneller als bei `HashSet`). `EnumSet` verwendet intern (nach Möglichkeite) ein einzelnes `long`-Feld, welches als Bitvektor dient und ist daher extrem schnell und effizient. Der Vorteil der Klasse `EnumSet` besteht darin, daß Sie eine ausdrucksfähige Notation verwenden können, um die An- beziehungsweise Abwesenheit einer binären Eigenschaft anzuzeigen, ohne sich um die Performanz kümmern zu müssen.

[29] Die Elemente eines Containers vom Typ `EnumSet` müssen aus einem einzigen Aufzählungstyp stammen. Das folgende Beispiel definiert die Positionen von Alarmgebern in einem Gebäude als Konstanten eines Aufzählungstyps:

```
//: enumerated/AlarmPoints.java
package enumerated;
public enum AlarmPoints {
    STAIR1, STAIR2, LOBBY, OFFICE1, OFFICE2, OFFICE3,
    OFFICE4, BATHROOM, UTILITY, KITCHEN
} ///:~
```

Der `EnumSet`-Container kann nun verwendet werden, um den Status der einzelnen Alarmgeber zu überwachen:

```
//: enumerated/EnumSets.java
// Operations on EnumSets
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

public class EnumSets {
    public static void main(String[] args) {
        EnumSet<AlarmPoints> points =
            EnumSet.noneOf(AlarmPoints.class); // Empty set
        points.add(BATHROOM);
        print(points);
        points.addAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
        print(points);
        points = EnumSet.allOf(AlarmPoints.class);
        points.removeAll(EnumSet.of(STAIR1, STAIR2, KITCHEN));
        print(points);
        points.removeAll(EnumSet.range(OFFICE1, OFFICE4));
        print(points);
        points = EnumSet.complementOf(points);
        print(points);
    }
} /* Output:
      [BATHROOM]
```

```
        [STAIR1, STAIR2, BATHROOM, KITCHEN]
        [LOBBY, OFFICE1, OFFICE2, OFFICE3, OFFICE4, BATHROOM, UTILITY]
        [LOBBY, BATHROOM, UTILITY]
        [STAIR1, STAIR2, OFFICE1, OFFICE2, OFFICE3, OFFICE4, KITCHEN]
    *///:~
```

Der Aufzählungstyp `AlarmPoints` wird statisch importiert, um die Verwendung der Konstanten zu vereinfachen. Die Methodennamen sind weitestgehend selbsterklärend und Sie finden sämtliche Einzelheiten in der API-Dokumentation der Klasse `EnumSet`. Die Dokumentation verdeutlicht die Geschwindigkeitsorientierung am Beispiel der mehrfach überladenen Methode `of()`: Es gibt eine Version mit Parameterliste variabler Länge sowie Versionen für zwei bis fünf Argumente. Die Überladung der `of()`-Methode ist ein Hinweis auf das Gewicht der Performanz beim Entwurf der Klasse `EnumSet`. Eine einzige `of()`-Methode mit Parameterliste variabler Länge hätte zwar dieselbe Funktion erfüllt, wäre aber etwas weniger effizient als die Lösung mit expliziten Parameterlisten. Wenn Sie `of()` mit zwei bis fünf Argumenten aufrufen, wird also eine der expliziten (etwas schnelleren) Versionen aufgerufen, bei einem oder mehr als fünf Argumenten dagegen die variable Version. Beachten Sie, daß der Compiler kein Array für den variablen Anteil der Argumentliste erzeugt, wenn Sie `of()` mit nur einem Argument aufrufen, das heißt der Aufruf mit einem Argument bewirkt keinen zusätzlichen Unkosten.

[30] Die Klasse `EnumSet` baut auf dem primitiven Typ `long` auf. Ein `long`-Feld beansprucht 64 Bit und jede Konstante des verknüpften Aufzählungstyps benötigt ein Bit, um An- oder Abwesenheit anzuzeigen. Sie können also einen Container vom Typ `EnumSet` mit einem Aufzählungstyp von bis zu 64 Konstanten verknüpfen, ohne mehr als einen einzigen `long`-Wert zu verbrauchen. Was geschieht, wenn Ihr Aufzählungstyp mehr als 64 Konstanten definiert?

```
//: enumerated/BigEnumSet.java
import java.util.*;

public class BigEnumSet {
    enum Big { A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10,
               A11, A12, A13, A14, A15, A16, A17, A18, A19, A20, A21,
               A22, A23, A24, A25, A26, A27, A28, A29, A30, A31, A32,
               A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43,
               A44, A45, A46, A47, A48, A49, A50, A51, A52, A53, A54,
               A55, A56, A57, A58, A59, A60, A61, A62, A63, A64, A65,
               A66, A67, A68, A69, A70, A71, A72, A73, A74, A75 }
    public static void main(String[] args) {
        EnumSet<Big> bigEnumSet = EnumSet.allOf(Big.class);
        System.out.println(bigEnumSet);
    }
} /* Output:
      [A0, A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16,
        A17, A18, A19, A20, A21, A22, A23, A24, A25, A26, A27, A28, A29, A30, A31,
        A32, A33, A34, A35, A36, A37, A38, A39, A40, A41, A42, A43, A44, A45, A46,
        A47, A48, A49, A50, A51, A52, A53, A54, A55, A56, A57, A58, A59, A60, A61,
        A62, A63, A64, A65, A66, A67, A68, A69, A70, A71, A72, A73, A74, A75]
   *///:~
```

Offensichtlich hat der `EnumSet`-Container keine Schwierigkeiten mit einem Aufzählungstyp, der mehr als 64 Elemente enthält. Wir können also annehmen (beachten Sie hierzu Übungsaufgabe 7), daß der Container bei Bedarf einen weiteren `long`-Speicherplatz beansprucht.

Übungsaufgabe 7: (3) Suchen Sie den Quelltext der Klasse `EnumSet` und erläutern Sie seine Funktionsweise. ■

20.9 Die Klasse EnumMap

[31] Die Klasse `EnumMap` repräsentiert einen speziellen *`Map`*-Container, dessen Schlüssel in einem einzigen Aufzählungstyp definiert sein müssen. Aufgrund der Einschränkungen für Aufzählungstypen kann *`Map`* intern mit Hilfe eines Arrays implementiert werden. Container vom Typ `EnumMap` sind daher extrem schnell und können zum Nachschlagen von Schlüsseln verwendet werden, die in einem Aufzählungstyp definiert sind.

[32] Sie können mit der `put()`-Methode nur solche Schlüssel/Wert-Paare anlegen, deren Schlüssel im verknüpften Aufzählungstyp definiert sind. Davon abgesehen, verhält sich ein Container vom Typ `EnumMap` wie ein Container vom Typ *`Map`*.

[33] Das folgende Beispiel demonstriert die Anwendung des Entwurfsmusters *Command*. Das Entwurfsmuster beginnt mit der Definition eines Interfaces, welches (typischerweise) eine einzelne Methode deklariert und in mehrere Implementierungen mit unterschiedlichem Verhalten dieser Methode mündet. Sie erzeugen *Kommandoobjekte*, die das Programm bei Bedarf aufruft:

```
//: enumerated/EnumMaps.java
// Basics of EnumMaps.
package enumerated;
import java.util.*;
import static enumerated.AlarmPoints.*;
import static net.mindview.util.Print.*;

interface Command { void action(); }

public class EnumMaps {
    public static void main(String[] args) {
        EnumMap<AlarmPoints,Command> em =
            new EnumMap<AlarmPoints,Command>(AlarmPoints.class);
        em.put(KITCHEN, new Command() {
                public void action() { print("Kitchen fire!"); }
            });
        em.put(BATHROOM, new Command() {
                public void action() { print("Bathroom alert!"); }
            });
        for(Map.Entry<AlarmPoints,Command> e : em.entrySet()) {
            printnb(e.getKey() + ": ");
            e.getValue().action();
        }
        try { // If there's no value for a particular key:
            em.get(UTILITY).action();
        } catch(Exception e) {
            print(e);
        }
    }
} /* Output:
      BATHROOM: Bathroom alert!
      KITCHEN: Kitchen fire!
      java.lang.NullPointerException
   *///:~
```

Wie beim Containertyp `EnumSet` richtet sich die Reihenfolge der Elemente bei `EnumMap` nach der Anordnung der Schlüssel im verknüpften Aufzählungstyp. Der letzte Abschnitt der `main()`-Methode zeigt, daß zu jeder Konstante des verknüpften Aufzählungstyps ein Schlüssel existiert, der zugeordnete Wert aber solange `null` ist, bis Sie dem Schlüssel per `put()`-Methode einen Wert zuweisen.

[34] Verglichen mit den im folgenden Abschnitt beschriebenen konstantenspezifischen Methoden hat

der Containertyp `EnumMap` den Vorteil, daß Sie die Werte ändern können, während die konstantenspezifischen Methoden zur Übersetzungszeit fixiert werden.

[35] In Abschnitt 20.11 lernen Sie, daß Sie Container vom Typ `EnumMap` verwenden können, um *multiple dispatching* in Situationen zu bewerkstelligen, in denen *multiple types of enums* miteinander interagieren.

20.10 Konstantenspezifische Methoden

[36] Die Aufzählungstypen von Java haben die sehr interessante Eigenschaft, daß Sie jeder Konstanten durch individuelle Methoden ein eigenes Verhalten geben können. Sie definieren in einem solchen Aufzählungstyp eine oder mehrere abstrakte Methoden und implementieren diese Methode(n) bei jeder einzelnen Konstanten. Ein Beispiel:

```
//: enumerated/ConstantSpecificMethod.java
import java.util.*;
import java.text.*;

public enum ConstantSpecificMethod {
    DATE_TIME {
        String getInfo() {
            return DateFormat.getDateInstance().format(new Date());
        }
    },
    CLASSPATH {
        String getInfo() {
            return System.getenv("CLASSPATH");
        }
    },
    VERSION {
        String getInfo() {
            return System.getProperty("java.version");
        }
    };
    abstract String getInfo();
    public static void main(String[] args) {
        for(ConstantSpecificMethod csm : values())
            System.out.println(csm.getInfo());
    }
} /* (Execute to see output) *////:~
```

Sie können die Methoden über die mit ihnen verknüpfte Konstante erreichen und aufrufen. Dieser Mechanismus wird häufig als *table-driven code* bezeichnet (beachten Sie die Ähnlichkeit mit dem auf Seite 797 erwähnten Entwurfsmuster *Command*).

[37] In der objektorientierten Programmierung wird unterschiedliches Verhalten mit unterschiedlichen Klassen assoziiert. Die Tatsache, daß jede Konstante eines Aufzählungstyps durch konstantenspezifische Methoden eigenes Verhalten haben kann, deutet an, daß jede Konstante einen eigenen Typ repräsentiert. Im obigen Beispiel wird jede Konstante als dem „Basistyp“ `ConstantSpecificMethod` zugehörig behandelt, aber Sie erhalten beim Aufrufen der Methode `getInfo()` polymorphes Verhalten.

[38] Damit ist die Reichweite der Ähnlichkeit allerdings erschöpft. Sie können die Konstanten eines Aufzählungstyps nicht wie Klassen behandeln:

```
//: enumerated/NotClasses.java
// {Exec: javap -c LikeClasses}
import static net.mindview.util.Print.*;

enum LikeClasses {
    WINKEN { void behavior() { print("Behavior1"); } },
    BLINKEN { void behavior() { print("Behavior2"); } },
    NOD { void behavior() { print("Behavior3"); } };
    abstract void behavior();
}

public class NotClasses {
    // void f1(LikeClasses.WINKEN instance) {} // Nope
} /* Output:
      Compiled from "NotClasses.java"
      abstract class LikeClasses extends java.lang.Enum{
      public static final LikeClasses WINKEN;

      public static final LikeClasses BLINKEN;

      public static final LikeClasses NOD;
      ...
   *///:~
```

Die Methode `f1()` zeigt, daß der Compiler Ihnen nicht erlaubt, eine Konstante eines Aufzählungstyps als Klasse zu verwenden. Das leuchtet ein, wenn Sie den vom Compiler generierten Code betrachten: Jede Konstante ist ein statisches finales Feld vom Typ `LikeClasses`.

[39] Aufgrund ihres statischen Charakters verhalten sich die Konstanten innerer Aufzählungstypen nicht wie gewöhnliche innere Klassen. Nicht-statische Felder oder Methoden in der äußeren Klasse sind nicht erreichbar.

[40] Das folgende interessantere Beispiel modelliert eine Autowaschanlage. Jeder Kunde hat mehrere Wahlmöglichkeiten für seinen Waschvorgang und jede Option bewirkt einen anderen Teilvorgang. Mit jeder Option kann eine konstantenspezifische Methoden verknüpft und ein Container vom Typ `EnumSet` verwendet werden, um die gewählten Einstellungen des Kunden zu speichern:

```
//: enumerated/CarWash.java
import java.util.*;
import static net.mindview.util.Print.*;

public class CarWash {
    public enum Cycle {
        UNDERBODY {
            void action() { print("Spraying the underbody"); }
        },
        WHEELWASH {
            void action() { print("Washing the wheels"); }
        },
        PREWASH {
            void action() { print("Loosening the dirt"); }
        },
        BASIC {
            void action() { print("The basic wash"); }
        },
        HOTWAX {
            void action() { print("Applying hot wax"); }
        },
        RINSE {
```

```
                void action() { print("Rinsing"); }
            },
            BLOWDRY {
                void action() { print("Blowing dry"); }
            };
            abstract void action();
        }
        EnumSet<Cycle> cycles = EnumSet.of(Cycle.BASIC, Cycle.RINSE);
        public void add(Cycle cycle) { cycles.add(cycle); }
        public void washCar() {
            for(Cycle c : cycles)
                c.action();
        }
        public String toString() { return cycles.toString(); }
        public static void main(String[] args) {
            CarWash wash = new CarWash();
            print(wash);
            wash.washCar();
            // Order of addition is unimportant:
            wash.add(Cycle.BLOWDRY);
            wash.add(Cycle.BLOWDRY); // Duplicates ignored
            wash.add(Cycle.RINSE);
            wash.add(Cycle.HOTWAX);
            print(wash);
            wash.washCar();
        }
    } /* Output:
          [BASIC, RINSE]
          The basic wash
          Rinsing
          [BASIC, HOTWAX, RINSE, BLOWDRY]
          The basic wash
          Applying hot wax
          Rinsing
          Blowing dry
       *///:~
```

Die Syntax zur Definition einer konstantenspezifischen Methode ist effektiv eine verkürzte Variante der Syntax einer anonymen inneren Klasse.

[41] Dieses Beispiel zeigt außerdem Eigenschaften des Containertyps `EnumSet`. Als *`Set`*-artiger Typ enthält ein solcher Container höchstens ein Exemplar einer Konstanten, ignoriert also mehrfaches Aufrufen der `add()`-Methode mit demselben Argument (dieses Verhalten ist sinnvoll, da Sie ein Bit nur einmal in den Zustand „ein“ umschalten können). Außerdem ist die Reihenfolge in der Sie die Konstanten übergeben unerheblich. Die Ausgabereihenfolge entspricht der Anordnung der Konstanten in der Definition des Aufzählungstyps.

[42] Konstantenspezifische Methoden können überschrieben werden, statt abstrakte Methoden zu implementieren:

```
//: enumerated/OverrideConstantSpecific.java
import static net.mindview.util.Print.*;

public enum OverrideConstantSpecific {
    NUT, BOLT,
    WASHER {
        void f() { print("Overridden method"); }
    };
```

```
        void f() { print("default behavior"); }
        public static void main(String[] args) {
            for(OverrideConstantSpecific ocs : values()) {
                printnb(ocs + ": ");
                ocs.f();
            }
        }
    } /* Output:
          NUT: default behavior
          BOLT: default behavior
          WASHER: Overridden method
       *///:~
```

Obwohl bei Aufzählungstypen einige Syntaxen nicht erlaubt sind, können Sie sie im allgemeinen wie Klassen behandeln.

20.10.1 Das Entwurfsmuster Chain of Responsibility

[43] Das Entwurfsmuster *Chain-of-Responsibility* beschreibt eine Anzahl verschiedener Lösungsmöglichkeiten eines Problems und verkettet sie miteinander. Trifft eine Anfrage ein, so durchwandert sie die Kette, bis eine der Lösungen die Anfrage verarbeiten kann.

[44] Konstantenspezifische Methoden gestatten, eine solche „Zuständigkeitskette“ mühelos zu implementieren. Stellen Sie sich die Modellierung eines Postamtes vor, bei dem jede Postsendung so allgemein wie möglich behandelt wird, die Verarbeitung aber solange versucht werden muß, bis sich die Postsendung als unzustellbar erweist. Sie können sich jeden Versuch als Strategie vorstellen (*Strategy* ist ebenfalls ein Entwurfsmuster) und die gesamte Liste als Zuständigkeitskette.

[45] Wir beginnen mit der Beschreibung der Postsendung. Alle verschiedenen Eigenschaften lassen sich in Form von Aufzählungstypen ausdrücken. Da die `Mail`-Objekte zufällig generiert werden läßt sich die Wahrscheinlichkeit, daß eine Postwendung beispielsweise als `GeneralDelivery` eingeordnet wird, am einfachsten dadurch reduzieren, daß *to create more non-YES instances*. Die Definitionen der Aufzählungstypen sehen daher ein wenig seltsam aus.

[46] Die Methode `randomMail()` der Klasse `Mail` erzeugt zufällige Postsendungen zum Testen. Die Methode `generator()` liefert ein Objekt vom Typ *`Iterable`*, welches mit Hilfe der `randomMail()`-Methode bei jedem Aufruf der `next()`-Methode des Iterators ein `Mail`-Objekt zurückgibt. Diese Konstruktion ermöglicht per `Mail.generator()` die Verwendung einer einfachen erweiterten `for`-Schleife:

```
//: enumerated/PostOffice.java
// Modeling a post office.
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

class Mail {
    // The NO's lower the probability of random selection:
    enum GeneralDelivery {YES,NO1,NO2,NO3,NO4,NO5}
    enum Scannability {UNSCANNABLE,YES1,YES2,YES3,YES4}
    enum Readability {ILLEGIBLE,YES1,YES2,YES3,YES4}
    enum Address {INCORRECT,OK1,OK2,OK3,OK4,OK5,OK6}
    enum ReturnAddress {MISSING,OK1,OK2,OK3,OK4,OK5}
    GeneralDelivery generalDelivery;
    Scannability scannability;
    Readability readability;
```

```
    Address address;
    ReturnAddress returnAddress;
    static long counter = 0;
    long id = counter++;
    public String toString() { return ”Mail ” + id; }
    public String details() {
        return toString() +
            ”, General Delivery: ” + generalDelivery +
            ”, Address Scanability: ” + scannability +
            ”, Address Readability: ” + readability +
            ”, Address Address: ” + address +
            ”, Return address: ” + returnAddress;
    }
    // Generate test Mail:
    public static Mail randomMail() {
        Mail m = new Mail();
        m.generalDelivery= Enums.random(GeneralDelivery.class);
        m.scannability = Enums.random(Scannability.class);
        m.readability = Enums.random(Readability.class);
        m.address = Enums.random(Address.class);
        m.returnAddress = Enums.random(ReturnAddress.class);
        return m;
    }
    public static Iterable<Mail> generator(final int count) {
        return new Iterable<Mail>() {
            int n = count;
            public Iterator<Mail> iterator() {
                return new Iterator<Mail>() {
                    public boolean hasNext() { return n-- > 0; }
                    public Mail next() { return randomMail(); }
                    public void remove() { // Not implemented
                        throw new UnsupportedOperationException();
                    }
                };
            }
        };
    }
}

public class PostOffice {
    enum MailHandler {
        GENERAL_DELIVERY {
            boolean handle(Mail m) {
                switch(m.generalDelivery) {
                case YES:
                    print(”Using general delivery for ” + m);
                    return true;
                default: return false;
                }
            }
        },
        MACHINE_SCAN {
            boolean handle(Mail m) {
                switch(m.scannability) {
                case UNSCANNABLE: return false;
                default:
                    switch(m.address) {
                    case INCORRECT: return false;
```

```
                    default:
                        print(''Delivering ''+ m + '' automatically'');
                        return true;
                    }
                }
            }
        },
        VISUAL_INSPECTION {
            boolean handle(Mail m) {
                switch(m.readability) {
                case ILLEGIBLE: return false;
                default:
                    switch(m.address) {
                    case INCORRECT: return false;
                    default:
                        print(''Delivering '' + m + '' normally'');
                        return true;
                    }
                }
            }
        },
        RETURN_TO_SENDER {
            boolean handle(Mail m) {
                switch(m.returnAddress) {
                case MISSING: return false;
                default:
                    print(''Returning '' + m + '' to sender'');
                    return true;
                }
            }
        };
        abstract boolean handle(Mail m);
    }
    static void handle(Mail m) {
        for(MailHandler handler : MailHandler.values())
            if(handler.handle(m))
                return;
        print(m + '' is a dead letter'');
    }
    public static void main(String[] args) {
        for(Mail mail : Mail.generator(10)) {
            print(mail.details());
            handle(mail);
            print(''*****'');
        }
    }
} /* Output:
      Mail 0, General Delivery: NO2, Address Scanability: UNSCANNABLE,
        Address Readability: YES3, Address Address: OK1, Return address: OK1
      Delivering Mail 0 normally
      *****
      Mail 1, General Delivery: NO5, Address Scanability: YES3,
        Address Readability: ILLEGIBLE, Address Address: OK5, Return address: OK1
      Delivering Mail 1 automatically
      *****
      Mail 2, General Delivery: YES, Address Scanability: YES3,
        Address Readability: YES1, Address Address: OK1, Return address: OK5
```

```
Using general delivery for Mail 2
*****
Mail 3, General Delivery: NO4, Address Scanability: YES3,
  Address Readability: YES1, Address Address: INCORRECT, Return address: OK4
Returning Mail 3 to sender
*****
Mail 4, General Delivery: NO4, Address Scanability: UNSCANNABLE,
  Address Readability: YES1, Address Address: INCORRECT, Return address: OK2
Returning Mail 4 to sender
*****
Mail 5, General Delivery: NO3, Address Scanability: YES1,
  Address Readability: ILLEGIBLE, Address Address: OK4, Return address: OK2
Delivering Mail 5 automatically
*****
Mail 6, General Delivery: YES, Address Scanability: YES4,
  Address Readability: ILLEGIBLE, Address Address: OK4, Return address: OK4
Using general delivery for Mail 6
*****
Mail 7, General Delivery: YES, Address Scanability: YES3,
  Address Readability: YES4, Address Address: OK2, Return address: MISSING
Using general delivery for Mail 7
*****
Mail 8, General Delivery: NO3, Address Scanability: YES1,
  Address Readability: YES3, Address Address: INCORRECT,
  Return address: MISSING
Mail 8 is a dead letter
*****
Mail 9, General Delivery: NO1, Address Scanability: UNSCANNABLE,
  Address Readability: YES2, Address Address: OK1, Return address: OK4
Delivering Mail 9 normally
*****
*///:~
```

Der Aufzählungstyp `MailHandler` repräsentiert die Zuständigkeitskette und die Anordnung der Konstanten bestimmt die Reihenfolge in der die Strategien auf jede Postsendung angewendet werden. Die Strategien werden eine nach der anderen versucht, bis eine funktioniert oder alle gescheitert sind (im letzteren Fall ist die Postsendung unzustellbar).

Übungsaufgabe 8: (6) Ändern Sie das Beispiel *PostOffice.java*, so daß Postsendungen nachgesendet (*to forward mail*) werden können. ■

Übungsaufgabe 9: (5) Ändern Sie die Klasse `PostOffice`, so daß Sie einen Container vom Typ `EnumMap` verwendet. ■

Projekt[2]**:** Spezialisierte Programmiersprachen wie Prolog verwenden einen Ansatz namens Rückwärtsverkettung (*backward chaining*), um Probleme wie das obige Postamt zu beschreiben. Recherchieren Sie solche Sprachen und entwickeln Sie ein Programm, das die einfache Ergänzung neuer Regeln zum System gestattet (orientieren Sie sich am Beispiel *PostOffice.java*). ■

[2]Projekte sind Vorschläge für Semester- oder Halbjahresarbeiten. Der *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können, enthält *keine* Lösungsvorschläge für Projekte.

20.10.2 Modellierung endlicher Automaten mit Aufzählungstypen

[47] Aufzählungstypen eignen sich ideal zur Modellierung endlicher Automaten (*state machine*). Ein endlicher Automat kann eine endliche Anzahl von Zuständen einnehmen und geht in der Regel in Abhängigkeit von einer Eingabe von einem Zustand in einen anderen über. Es gibt aber auch flüchtige Zustände (*transient states*), die der Automat beendet, wenn seine Aufgabe erledigt ist.

[48–49] Jeder Zustand erlaubt bestimmte Eingaben und verschiedene Eingaben überführen den Automaten in unterschiedliche neue Zustände. Aufzählungstypen beschränken die Auswahl möglicher Fälle und sind beim Aufzählen der verschiedenen Zustände und Eingaben recht nützlich. Jeder Zustand ist typischerweise auch mit einer Ausgabe verknüpft.

[50] Ein Verkaufsautomat ist ein gutes Beispiel für einen endlichen Automaten. Wir definieren zuerst die verschiedenen Eingabemöglichkeiten als Aufzählungstyp:

```
//: enumerated/Input.java
package enumerated;
import java.util.*;

public enum Input {
    NICKEL(5), DIME(10), QUARTER(25), DOLLAR(100),
    TOOTHPASTE(200), CHIPS(75), SODA(100), SOAP(50),
    ABORT_TRANSACTION {
        public int amount() { // Disallow
            throw new RuntimeException("ABORT.amount()");
        }
    },
    STOP { // This must be the last instance.
        public int amount() { // Disallow
            throw new RuntimeException("SHUT_DOWN.amount()");
        }
    };
    int value; // In cents
    Input(int value) { this.value = value; }
    Input() {}
    int amount() { return value; }; // In cents
    static Random rand = new Random(47);
    public static Input randomSelection() {
        // Don't include STOP:
        return values()[rand.nextInt(values().length - 1)];
    }
} ///:~
```

Beachten Sie, daß jede Konstante des Aufzählungstyps `Input` ein `amount`-Feld besitzt, in dem der Münzwert beziehungsweise der Preis gespeichert wird. Die Methode `amount()` ist in der Schnittstelle von `Input` definiert, um das `amount`-Feld abfragen zu können. Dieses Standardverhalten der `amount()`-Methode ist bei den Konstanten `ABORT_TRANSACTION` und `STOP` ungebracht. Der obige Ansatz ist zwar ein wenig sonderbar (Definition einer Methode in der Schnittstelle des Aufzählungstyps einerseits, Auswerfen einer Ausnahme, wenn diese Methode auf bestimmten Konstanten aufgerufen wird andererseits), ist uns aber durch die Einschränkungen der Aufzählungstypen auferlegt.

[51] Die Klasse `VendingMachine` (der Automat) ordnet diese Eingabemöglichkeiten zuerst mit Hilfe des Aufzählungstyps `Category` in verschiedene Kategorien ein, um die Eingaben anschließend mit Hilfe von `switch`-Anweisungen nach Kategorie verarbeiteten zu können. Das folgende Beispiel zeigt, wie Aufzählungstypen den Quelltext klarer und leichter handhabbar machen:

```
//: enumerated/VendingMachine.java
```

```
// {Args: VendingMachineInput.txt}
package enumerated;
import java.util.*;
import net.mindview.util.*;
import static enumerated.Input.*;
import static net.mindview.util.Print.*;

enum Category {
    MONEY(NICKEL, DIME, QUARTER, DOLLAR),
    ITEM_SELECTION(TOOTHPASTE, CHIPS, SODA, SOAP),
    QUIT_TRANSACTION(ABORT_TRANSACTION),
    SHUT_DOWN(STOP);
    private Input[] values;
    Category(Input... types) { values = types; }
    private static EnumMap<Input,Category> categories =
        new EnumMap<Input,Category>(Input.class);
    static {
        for(Category c : Category.class.getEnumConstants())
            for(Input type : c.values)
                categories.put(type, c);
    }
    public static Category categorize(Input input) {
        return categories.get(input);
    }
}

public class VendingMachine {
    private static State state = State.RESTING;
    private static int amount = 0;
    private static Input selection = null;
    enum StateDuration { TRANSIENT } // Tagging enum
    enum State {
        RESTING {
            void next(Input input) {
                switch(Category.categorize(input)) {
                case MONEY:
                    amount += input.amount();
                    state = ADDING_MONEY;
                    break;
                case SHUT_DOWN:
                    state = TERMINAL;
                default:
                }
            }
        },
        ADDING_MONEY {
            void next(Input input) {
                switch(Category.categorize(input)) {
                case MONEY:
                    amount += input.amount();
                    break;
                case ITEM_SELECTION:
                    selection = input;
                    if(amount < selection.amount())
                        print("Insufficient money for " + selection);
                    else state = DISPENSING;
                    break;
                case QUIT_TRANSACTION:
```

```
                    state = GIVING_CHANGE;
                    break;
                case SHUT_DOWN:
                    state = TERMINAL;
                default:
                }
            }
        },
        DISPENSING(StateDuration.TRANSIENT) {
            void next() {
                print("here is your " + selection);
                amount -= selection.amount();
                state = GIVING_CHANGE;
            }
        },
        GIVING_CHANGE(StateDuration.TRANSIENT) {
            void next() {
                if(amount > 0) {
                    print("Your change: " + amount);
                    amount = 0;
                }
                state = RESTING;
            }
        },
        TERMINAL { void output() { print("Halted"); } };
        private boolean isTransient = false;
        State() {}
        State(StateDuration trans) { isTransient = true; }
        void next(Input input) {
            throw new RuntimeException(
                "Only call next(Input input) for non-transient states");
        }
        void next() {
            throw new RuntimeException(
                "Only call next() for StateDuration.TRANSIENT states");
        }
        void output() { print(amount); }
    }
    static void run(Generator<Input> gen) {
        while(state != State.TERMINAL) {
            state.next(gen.next());
            while(state.isTransient)
                state.next();
            state.output();
        }
    }
    public static void main(String[] args) {
        Generator<Input> gen = new RandomInputGenerator();
        if(args.length == 1)
            gen = new FileInputGenerator(args[0]);
        run(gen);
    }
}

// For a basic sanity check:
class RandomInputGenerator implements Generator<Input> {
    public Input next() { return Input.randomSelection(); }
}
```

```
// Create Inputs from a file of ';'-separated strings:
class FileInputGenerator implements Generator<Input> {
    private Iterator<String> input;
    public FileInputGenerator(String fileName) {
        input = new TextFile(fileName, ";").iterator();
    }
    public Input next() {
        if(!input.hasNext())
            return null;
        return Enum.valueOf(Input.class, input.next().trim());
    }
} /* Output:
      25
      50
      75
      here is your CHIPS
      0
      100
      200
      here is your TOOTHPASTE
      0
      25
      35
      Your change: 35
      0
      25
      35
      Insufficient money for SODA
      35
      60
      70
      75
      Insufficient money for SODA
      75
      Your change: 75
      0
      Halted
   *///:~
```

[53] Da die Konstanten eines Aufzählungstyps meistens per `switch`-Anweisung ausgewertet werden (beachten Sie zusätzlichen Aufwand, um die Kombination von Aufzählungstyp und `switch` in Java-Programmen zu erleichtern), lautet die wichtigste Frage bei einem Ansatz mit mehreren Aufzählungstypen, welches Kriterium durch eine `switch`-Anweisung implementiert wird. Der Verkaufsautomat wertet die Eingabemöglichkeiten der einzelnen Zustände per `switch` nach den Basiskategorien „Münzeinwurf“, „Produktwahl“, „Transaktionsabbruch“ und „Automat abschalten“ aus. Jede Kategorie umfaßt mehrere Münzsorten beziehungsweise Produkte. Der Aufzählungstyp `Category` gruppiert die zehn verschiedenen Eingabemöglichkeiten und die Methode `categorize()` liefert in den `switch`-Anweisungen die Kategorie der ihr übergebenen Eingabe. Die `categorize()`-Methode fragt einen `EnumMap`-Container ab, um die Kategorie nachzuschlagen.

[54] Beim Durchsehen der Klasse `VendingMachine` sehen Sie, daß jeder Zustand individuell definiert ist und anders auf Eingaben reagiert. Beachten Sie auch die beiden flüchtigen Zustände `DISPENSING` und `GIVING_CHANGE`: Die `run()`-Methode wartet auf Eingaben und durchläuft solange einen Zustand nach dem anderen, bis der Automat einen nicht-flüchtigen Zustand einnimmt.

[55] Der Verkaufsautomat kann auf zwei Arten getestet werden, genauer mit zwei verschiedenen

Generatorobjekten. Die Klasse `RandomInputGenerator` liefert eine Eingabe nach der anderen, außer `SHUT_DOWN`. Indem Sie das Programm eine längere Zeit laufen lassen, bekommen Sie einen Anhaltspunkt dafür, daß der Automat nicht in einen unerwünschten Zustand gerät. Die Klasse `FileInputGenerator` liest eine Datei ein, welche Eingaben in Textform beschreibt, wandelt sich in Konstanten eines Aufzählungstyps um und erzeugt `Input`-Objekte. Die obige Ausgabe stammt von der folgenden Textdatei:

```
QUARTER; QUARTER; QUARTER; CHIPS;
DOLLAR; DOLLAR; TOOTHPASTE;
QUARTER; DIME; ABORT_TRANSACTION;
QUARTER; DIME; SODA;
QUARTER; DIME; NICKEL; SODA;
ABORT_TRANSACTION;
STOP;
```

Das Design der Klasse `VendingMachine` hat die Einschränkung, daß die von den Konstanten des Aufzählungstyps `State` abgefragten oder geänderten Felder statische Felder sein müssen, das heißt Sie können nur ein einziges Objekt der Klasse `VendingMachine` nutzen. Andererseits hält sich diese Beschränkung im Rahmen, wenn Sie sich eine tatsächliche Implementierung (embedded Java) vorstellen, da in diesem Fall wahrscheinlich nur eine Anwendung pro Automat betrieben wird.

Übungsaufgabe 10: (7) Ändern Sie die Klasse `VendingMachine`, so daß nur `EnumMap`-Container verwendet werden, damit das Programm mehrere `VendingMachine`-Objekte unterstützt. ■

Übungsaufgabe 11: (7) Bei einem realistischen Verkaufsautomaten müssen die verkauften Produkte einfach ergänzt und geändert werden können, so daß die Einschränkungen durch den Charakter von `Input` als Aufzählungstyp nicht praktikabel sind (denken Sie daran, daß ein Aufzählungstyp nur eine beschränkte Auswahl von Typen/Konstanten gestattet). Ändern Sie das Beispiel *VendingMachine.java*, so daß die angebotenen Produkte in Form einer Klasse statt eines Aufzählungstyps erfaßt werden und initialisieren Sie einen Container vom Typ `ArrayList` mittels einer Textdatei mit den Produktnamen (verwenden Sie dazu die Hilfsklasse `net.mindview.util.TextFile`). ■

Projekt[3]: Entwerfen Sie ein Design für einen internationalisierten Verkaufsautomaten, so daß ein Gerät mühelos an jedes Land angepaßt werden kann. ■

20.11 Multiple Dispatching

[56] Wenn Sie mit mehreren miteinander wechselwirkenden Typen hantieren, kann ein Programm schnell unübersichtlich werden. Stellen Sie sich beispielsweise ein Programm vor, das mathematische Ausdrücke parst und auswertet. Sie erwarten Ausdrücke der Form `Number.plus(Number)`, `Number.multiply(Number)` und so weiter, wobei `Number` die Basisklasse einer Familie von Klassen ist, die Objekte numerischen Inhaltes repräsentieren. Die Aufgabe lautet, einen Ausdruck wie `a.plus(b)` auszuwerten, wobei die exakten Typen der Operanden `a` und `b` nicht bekannt sind.

[57] Die Lösung dieser Aufgabe beginnt mit einer Spracheigenschaft, an die Sie wahrscheinlich in diesem Zusammenhang nicht nachdenken: Java unterstützt nur single dispatching, das heißt, wenn eine Operation mehr als einen Operanden unbekannten Typs beinhaltet, kann Java die dynamische Bindung nur bezüglich eines dieser Typen bewerkstelligen. Damit ist das beschriebene Problem nicht gelöst, so daß Sie letztendlich einige Typen manuell ermitteln und das dynamische Bindungsverhalten effektiv nachprogrammieren müssen.

[3] Projekte sind Vorschläge für Semester- oder Halbjahresarbeiten. Der *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können, enthält *keine* Lösungsvorschläge für Projekte.

[58] Die Lösung ist sogenanntes multiple dispatching. (Im vorliegenden Fall sind nur zwei dispatches notwendig, sogenanntes double dispatching.) Polymorphie tritt nur bei Methodenaufrufen auf. Double dispatching verlangt somit zwei Methodenaufrufe, je einen, um die beiden unbekannten Typen zu ermitteln. Multiple dispatching setzt je einen virtuellen Methodenaufruf pro Typ voraus: Stammen die unbekannten Typen aus zwei verschiedenen miteinander wechselwirkenden Hierarchien, so brauchen Sie einen Methodenaufruf für jede der beiden Hierarchien. In der Regel gestalten Sie das Programm so, daß ein einzelner Methodenaufruf mehr als einen virtuellen Methodenaufruf verursacht und dadurch mehr als einen Typ ermittelt. Sie brauchen mehr als eine Methode, um diesen Effekt herbeizuführen: Eine Methode pro dispatch. Die Methoden `compete()` und `eval()` des folgenden Beispiels (das Spiel „Schere, Stein, Papier“) gehören demselben Typ an und liefern einen von drei möglichen Ausfällen: [4]

```
//: enumerated/Outcome.java
package enumerated;
public enum Outcome { WIN, LOSE, DRAW } ///:~
```

```
//: enumerated/RoShamBo1.java
// Demonstration of multiple dispatching.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

interface Item {
    Outcome compete(Item it);
    Outcome eval(Paper p);
    Outcome eval(Scissors s);
    Outcome eval(Rock r);
}

class Paper implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return DRAW; }
    public Outcome eval(Scissors s) { return WIN; }
    public Outcome eval(Rock r) { return LOSE; }
    public String toString() { return ''Paper''; }
}

class Scissors implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return LOSE; }
    public Outcome eval(Scissors s) { return DRAW; }
    public Outcome eval(Rock r) { return WIN; }
    public String toString() { return ''Scissors''; }
}

class Rock implements Item {
    public Outcome compete(Item it) { return it.eval(this); }
    public Outcome eval(Paper p) { return WIN; }
    public Outcome eval(Scissors s) { return LOSE; }
    public Outcome eval(Rock r) { return DRAW; }
    public String toString() { return ''Rock''; }
}

public class RoShamBo1 {
    static final int SIZE = 20;
    private static Random rand = new Random(47);
    public static Item newItem() {
```

[4] Dieses Beispiel existierte mehrere Jahre in meinen Bücher zu C++ und Java (in *Thinking in Patterns*; siehe *http://www.mindview.net*), bevor es ohne Zuschreibung in einem Buch anderer Autoren erschien.

```
            switch(rand.nextInt(3)) {
                default:
                case 0: return new Scissors();
                case 1: return new Paper();
                case 2: return new Rock();
            }
        }
        public static void match(Item a, Item b) {
            System.out.println(a + " vs. " + b + ": " + a.compete(b));
        }
        public static void main(String[] args) {
            for(int i = 0; i < SIZE; i++)
                match(newItem(), newItem());
        }
    } /* Output:
          Rock vs. Rock: DRAW
          Paper vs. Rock: WIN
          Paper vs. Rock: WIN
          Paper vs. Rock: WIN
          Scissors vs. Paper: WIN
          Scissors vs. Scissors: DRAW
          Scissors vs. Paper: WIN
          Rock vs. Paper: LOSE
          Paper vs. Paper: DRAW
          Rock vs. Paper: LOSE
          Paper vs. Scissors: LOSE
          Paper vs. Scissors: LOSE
          Rock vs. Scissors: WIN
          Rock vs. Paper: LOSE
          Paper vs. Rock: WIN
          Scissors vs. Paper: WIN
          Paper vs. Scissors: LOSE
          Paper vs. Scissors: LOSE
          Paper vs. Scissors: LOSE
          Paper vs. Scissors: LOSE
       *///:~
```

[59] *Item* ist das Interface der multiply dispatched Typen in diesem Beispiel. Die `match()`-Methode der Klasse `RoShamBo1` erwartet zwei Argumente vom Typ *Item* und beginnt den double dispatching-Prozeß mit einem Aufruf der `compete()`-Methode ihres ersten Argumentes. Der virtuelle Mechanismus bestimmt den Typ von `a` und führt zur `compete()`-Methode des eigentlichen Typs des von `a` referenzierten Objektes. Die `compete()`-Methode bewerkstelligt den zweiten dispatch-Schritt, indem sie die `eval()`-Methode des verbleibenden Argumentes `b` aufruft. Die Übergabe der Selbstreferenz `this` an die überladene `eval()`-Methode erhält die Typinformationen aus dem ersten dispatch-Schritt. Nachdem zweiten dispatch-Schritt kennen Sie die exakten Typen beider *Item*-Objekte.

[60] Die Vorbereitung für multiple dispatching erfordert einigen Umstand. Vergessen Sie darüber nicht den Vorteil der eleganten Syntax beim Aufrufen einer Operationen mit verschiedenen unbekannten Typen. Statt ungelenkten Quelltextes, um den Typ eines oder mehrerer Objekte während eines Methodenaufrufs zu bestimmen, wählen Sie einfach zwei Objekte aus, kümmern Sie nicht um ihren Typen und die Interaktion zwischen den Objekten funktioniert. Machen Sie sich bewußt, daß diese Form von Eleganz für Sie wichtig ist, bevor sie mit multiple dispatching auseinandersetzen.

20.11.1 Lösung per Aufzählungstyp

[61] Eine direkte Übertragung des Beispiels *RoShamBo1.java* in eine Lösung auf der Basis von Aufzählungstypen ist schwierig, da die Konstanten eines Aufzählungstyps keine Typen sind, die überladene `eval()`-Methode also nicht funktioniert (Sie können die Konstanten eines Aufzählungstyps nicht als Argumenttypen verwenden). Es gibt dennoch verschiedene Ansätze, um multiple dispatching mit Aufzählungstypen zu implementieren.

[62] Einer dieser Ansätze verwendet einen Konstruktor, um jede Konstante mit a "row" von Ausfällen zu initialisieren. Zusammengenommen ergibt sich eine Art Tabelle zum Nachschlagen:

```
//: enumerated/RoShamBo2.java
// Switching one enum on another.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo2 implements Competitor<RoShamBo2> {
    PAPER(DRAW, LOSE, WIN),
    SCISSORS(WIN, DRAW, LOSE),
    ROCK(LOSE, WIN, DRAW);
    private Outcome vPAPER, vSCISSORS, vROCK;
    RoShamBo2(Outcome paper,Outcome scissors,Outcome rock) {
        this.vPAPER = paper;
        this.vSCISSORS = scissors;
        this.vROCK = rock;
    }
    public Outcome compete(RoShamBo2 it) {
        switch(it) {
            default:
            case PAPER: return vPAPER;
            case SCISSORS: return vSCISSORS;
            case ROCK: return vROCK;
        }
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo2.class, 20);
    }
} /* Output:
      ROCK vs. ROCK: DRAW
      SCISSORS vs. ROCK: LOSE
      SCISSORS vs. ROCK: LOSE
      SCISSORS vs. ROCK: LOSE
      PAPER vs. SCISSORS: LOSE
      PAPER vs. PAPER: DRAW
      PAPER vs. SCISSORS: LOSE
      ROCK vs. SCISSORS: WIN
      SCISSORS vs. SCISSORS: DRAW
      ROCK vs. SCISSORS: WIN
      SCISSORS vs. PAPER: WIN
      SCISSORS vs. PAPER: WIN
      ROCK vs. PAPER: LOSE
      ROCK vs. SCISSORS: WIN
      SCISSORS vs. ROCK: LOSE
      PAPER vs. SCISSORS: LOSE
      SCISSORS vs. PAPER: WIN
      SCISSORS vs. PAPER: WIN
      SCISSORS vs. PAPER: WIN
```

```
        SCISSORS vs. PAPER: WIN
    *///:~
```

Nachdem beide Typen im Körper der `compete()`-Methode identifiziert sind, bleibt nur noch das entsprechende Ergebnis (`Outcome`) zurückzugeben. Sie könnten ebenfalls eine weitere Methode aufrufen, beispielsweise selbst über das dem Konstruktor übergebene Kommandoobjekt.

[63] *RoShamBo2.java* ist viel kleiner und einfacher als das ursprüngliche Beispiel und läßt sich leichter im Auge behalten. Beachten Sie, daß noch immer zwei dispatches erforderlich sind, um die Typen beider Objekte zu bestimmen. Im Beispiel *RoShamBo1.java* wurden beide dispatches mit Hilfe virtueller Methodenaufrufe bewerkstelligt. Hier beinhaltet nur die erste Stufe einen virtuellen Methodenaufruf. Sie zweite Stufe verwendet eine `switch`-Anweisung, ist aber sicher, da die Auswahlmöglichkeiten in der `switch`-Anweisung durch den Aufzählungstyp `RoShamBo2` beschränkt sind.

[64] Die Anweisungen, die den Aufzählungstyp `RoShamBo2` steuern, wurden ausgelagert, um sie auch in anderen Beispielen nutzen zu können. Das Interface *`Competitor`* beschreibt einen Typ der gegen einen anderes Objekt desselben Typs antritt:

```
//: enumerated/Competitor.java
// Switching one enum on another.
package enumerated;

public interface Competitor<T extends Competitor<T>> {
    Outcome compete(T competitor);
} ///:~
```

Anschließend definieren wir in der Klasse `RoShamBo` zwei statische Methoden (static to avoid having to specify the parameter type explicitly). Die Methode `match()` ruft die `compete()`-Methode eines Objektes vom Typ *`Competitor`* mit einem weiteren Objekt dieses Typs. In diesem Fall genügt es, wenn der Parametertyp zu *`Competitor<T>`* paßt. Die Methode `play()` erwartet dagegen einen Parametertyp, der sowohl zu `Enum<T>` als auch zum Typ *`Competitor<T>`* paßt, da er sowohl der statischen `Enums`-Methode `random()` als auch der zuvor definierten `match()`-Methode übergeben wird:

```
//: enumerated/RoShamBo.java
// Common tools for RoShamBo examples.
package enumerated;
import net.mindview.util.*;

public class RoShamBo {
    public static <T extends Competitor<T>>
    void match(T a, T b) {
        System.out.println(a + " vs. " + b + ": " + a.compete(b));
    }
    public static <T extends Enum<T> & Competitor<T>>
    void play(Class<T> rsbClass, int size) {
        for(int i = 0; i < size; i++)
            match(Enums.random(rsbClass),Enums.random(rsbClass));
    }
} ///:~
```

Die `play()`-Methode hat keinen Rückgabewert, der den Typparameter `T` beinhaltet, so daß Sie anscheinend auch Platzhalter bei `Class<T>` einsetzen könnten, statt die führende Beschreibung der Parameter zu verwenden. Platzhalter können aber nicht mehr als einen Basistyp erweitern, so daß wir an die obigen Notation gebunden sind.

20.11.2 Lösung mit konstantenspezifischen Methoden

[65] Da Sie mit Hilfe konstantenspezifischer Methoden jeder Konstanten eines Aufzählungstyps ein individuelles Verhalten geben können, scheinen diese eine perfekte Lösung für das ~~multiple dispatching~~ Problem zu sein. Obwohl Sie konstantenspezifisches Verhalten implementieren können, repräsentieren die Konstanten eines Aufzählungstyps keine Typen, können also nicht als Argumenttypen in eine Methodensignatur eingesetzt werden. Das Beste, was Sie im Rahmen unseres „Schere, Stein, Papier"-Beispiels tun können, besteht darin, eine `switch`-Anweisung zu wählen:

```
//: enumerated/RoShamBo3.java
// Using constant-specific methods.
package enumerated;
import static enumerated.Outcome.*;

public enum RoShamBo3 implements Competitor<RoShamBo3> {

    PAPER {
        public Outcome compete(RoShamBo3 it) {
            switch (it) {
                default: // To placate the compiler
                case PAPER: return DRAW;
                case SCISSORS: return LOSE;
                case ROCK: return WIN;
            }
        }
    },
    SCISSORS {
        public Outcome compete(RoShamBo3 it) {
            switch (it) {
                default:
                case PAPER: return WIN;
                case SCISSORS: return DRAW;
                case ROCK: return LOSE;
            }
        }
    },
    ROCK {
        public Outcome compete(RoShamBo3 it) {
            switch (it) {
                default:
                case PAPER: return LOSE;
                case SCISSORS: return WIN;
                case ROCK: return DRAW;
            }
        }
    };

    public abstract Outcome compete(RoShamBo3 it);

    public static void main(String[] args) {
        RoShamBo.play(RoShamBo3.class, 20);
    }

} /* Same output as RoShamBo2.java *////:~
```

Diese Lösung funktioniert zwar und ist auch keineswegs unvernünftig, aber der Ansatz des Beispiels *RoShamBo2.java* erfordert beim Hinzufügen eines weiteren Typs weniger Zeilen und wirkt einfacher.

[66] Beispiel *RoShamBo3.java* läßt sich aber vereinfachen und komprimieren:

```
//: enumerated/RoShamBo4.java
package enumerated;

public enum RoShamBo4 implements Competitor<RoShamBo4> {
    ROCK {
        public Outcome compete(RoShamBo4 opponent) {
            return compete(SCISSORS, opponent);
        }
    },
    SCISSORS {
        public Outcome compete(RoShamBo4 opponent) {
            return compete(PAPER, opponent);
        }
    },
    PAPER {
        public Outcome compete(RoShamBo4 opponent) {
            return compete(ROCK, opponent);
        }
    };
    Outcome compete(RoShamBo4 loser, RoShamBo4 opponent) {
        return ((opponent == this) ? Outcome.DRAW
                   : ((opponent == loser) ? Outcome.WIN
                       : Outcome.LOSE));
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo4.class, 20);
    }
} /* Same output as RoShamBo2.java *////:~
```

Hier wird die zweite dispatcher-Stufe mit Hilfe der zweiparametrigen Version der `compete()`-Methode bewerkstelligt, die eine Reihe von Vergleichsoperationen ausführt und somit der Funktion der `switch`-Anweisung ähnelt. Diese Lösung ist kleiner aber etwas verwirrender. Dieser Einwand kann diesen Ansatz bei einer größeren Anwendung entkräften.

20.11.3 Lösung per Aufzählungstyp

[67] Die speziell für die äußerst effiziente Kombination mit Aufzählungstypen entwickelte Klasse `EnumMap` ermöglicht „echtes“ double dispatching. Da unser Ziel lautet, nach zwei unbekannten Typen zu verzweigen, liefert ein `EnumMap`-Container von `EnumMap`-Containern double dispatching:

```
//: enumerated/RoShamBo5.java
// Multiple dispatching using an EnumMap of EnumMaps.
package enumerated;
import java.util.*;
import static enumerated.Outcome.*;

enum RoShamBo5 implements Competitor<RoShamBo5> {
    PAPER, SCISSORS, ROCK;
    static EnumMap<RoShamBo5,EnumMap<RoShamBo5,Outcome>> table =
        new EnumMap<RoShamBo5, EnumMap<RoShamBo5,Outcome>>(RoShamBo5.class);
    static {
        for(RoShamBo5 it : RoShamBo5.values())
            table.put(it, new EnumMap<RoShamBo5,Outcome>(RoShamBo5.class));
        initRow(PAPER, DRAW, LOSE, WIN);
        initRow(SCISSORS, WIN, DRAW, LOSE);
        initRow(ROCK, LOSE, WIN, DRAW);
    }
```

```
    static void
        initRow(RoShamBo5 it, Outcome vPAPER, Outcome vSCISSORS, Outcome vROCK) {
        EnumMap<RoShamBo5,Outcome> row = RoShamBo5.table.get(it);
        row.put(RoShamBo5.PAPER, vPAPER);
        row.put(RoShamBo5.SCISSORS, vSCISSORS);
        row.put(RoShamBo5.ROCK, vROCK);
    }
    public Outcome compete(RoShamBo5 it) {
        return table.get(this).get(it);
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo5.class, 20);
    }
} /* Same output as RoShamBo2.java *///:~
```

Der von `tables` referenzierte `EnumMap`-Container wird mit Hilfe eines statischen Initialisierungsblocks bewertet. Sie erkennen die tabellenartige Struktur an den Aufrufen der `initRow()`-Methode. Beachten Sie die `compete()`-Methode, in der beide dispatching-Stufen in einer einzelnen Anweisung stattfinden.

20.11.4 Lösung mit zweidimensionalem Array

[68] Unter Berücksichtigung der Tatsache, daß jeder Konstanten eines Aufzählungstyps ein fester Index zugeordnet ist (abhängig von der Deklarationsreihenfolge) und die `ordinal()`-Methode diesen Index zurückgibt, läßt sich die Lösung noch weiter vereinfachen. Ein zweidimensionales Array, welches die Spieler auf die Ausfälle abbildet, liefert die kürzeste und einfachste Lösung. (Wohl auch die schnellste Lösung. Denken Sie aber daran, daß der Containertyp `EnumMap` intern auf Arrays aufbaut.):

```
//: enumerated/RoShamBo6.java
// Enums using ''tables'' instead of multiple dispatch.
package enumerated;
import static enumerated.Outcome.*;

enum RoShamBo6 implements Competitor<RoShamBo6> {
    PAPER, SCISSORS, ROCK;
    private static Outcome[][] table = {
        { DRAW, LOSE, WIN }, // PAPER
        { WIN, DRAW, LOSE }, // SCISSORS
        { LOSE, WIN, DRAW }, // ROCK
    };
    public Outcome compete(RoShamBo6 other) {
        return table[this.ordinal()][other.ordinal()];
    }
    public static void main(String[] args) {
        RoShamBo.play(RoShamBo6.class, 20);
    }
} ///:~
```

Das von `table` referenzierte Array und die Aufrufe der `initRow()`-Methode im vorigen Beispiel repräsentieren genau dieselbe Anordnung.

[69] Die geringe Länge des Quelltext macht diese Lösung im Vergleich mit den andern sehr attraktiv. Das liegt teilweise daran, daß sie leichter zu verstehen und zu ändern ist, aber auch daran, daß sie einfacher ist. Andererseits ist diese Lösung weniger „sicher“ als die vorigen Beispiele, da sie auf einem Array aufbaut. Bei einem größeren Array schleicht sich eventuell ein Fehler in der Länge der beiden

Dimensionen ein und wenn Ihre Tests nicht alle Möglichkeiten abdecken, bleibt eventuell ein Fehler unentdeckt.

[70] Alle Lösungen in diesem Abschnitt repräsentieren unterschiedliche Arten von Tabellen, aber es lohnt sich, die verschiedenen Ausdrucksformen zu untersuchen, um die am besten passende Variante zu finden. Die obige Lösung ist zwar die kompakteste, andererseits aber auch starr, da sie nur eine konstante Ausgabe zu einer konstanten Eingabe liefern kann. Andererseits hält Sie nichts davon ab, aus der Tabelle ein *function object* aufzubauen. Das Konzept des *table-driven code* ist bei gewissen Problemtypen sehr mächtig.

20.12 Zusammenfassung

[71] Aufzählungstypen sind selbst nicht übermäßig kompliziert, aber durch die Kombinationsmöglichkeiten mit Polymorphie, generischen Typen und dem Reflexionsmechanismus steht dieses Kapitel im letzten Drittel des Buches.

[72] Obwohl die Aufzählungstypen von Java etwas ausgefeilter sind, als bei C oder C++, sind sie eine „kleine“ Komponente und die Sprache hat, wenn auch ein wenig ungeschickt, mehrere Jahre ohne Aufzählungstypen überlebt. Dennoch dokumentiert dieses Kapitel wie wertvoll die Auswirkungen einer „kleinen“ Komponente sein können: Hin und wieder liefert eine solche Spracheigenschaft oder Fähigkeit den richtigen Ansatz, um ein Problem elegant und nachvollziehbar zu lösen. Ich habe in diesem Buch immer wieder darauf hingewiesen, daß Eleganz wichtig ist und die Nachvollziehbarkeit den Unterschied zwischen einer erfolgreichen und einer gescheiterten Lösung bedeuten kann, weil andere Programmierer sie nicht verstehen.

[73] Apropos Nachvollziehbarkeit: Die armseelige Wahl der Bezeichnung „Enumeration“ (bei Java 1.0), anstelle des gängigen und akzeptierten Begriffs „Iterator“, zur Benennung eines Objektes, welches die Elemente einer Reihe nacheinander auswählt (siehe Unterabschnitt 18.13.1), ist eine bedauerliche Quelle der Verwirrung. Einige Programmiersprachen bezeichnen Aufzählungstypen gar als „Enumeratoren“! Dieser Fehltritt wurde bei Java ausgebessert. Das Interface `java.util.Enumeration` kann dagegen nicht einfach entfernt werden und kommt noch immer in älteren (manchmal auch neuen) Quelltexten, der Standardbibliothek und Dokumentation vor.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 21

Annotationen

Inhaltsübersicht

[0] Annotationen („Metadaten“) repräsentieren einen formalisierten Ansatz, um Ihren Quelltext und Bytecode mit zusätzlichen Informationen zu versehen, die Sie zu einem späteren Zeit mühelos auswerten können.[1]

[1] Annotationen sind zum Teil durch die allgemeine Entwicklungsrichtung motiviert, Metadaten und Quelltextdateien zu kombinieren, statt in separaten Dateien zu pflegen. Teilweise sind sie auch eine Reaktion auf den Druck durch die Eigenschaften und Fähigkeiten anderer Sprachen wie C#.

[2] Annotation gehören zu den fundamentalen Änderungen der Programmiersprache im Rahmen von Version 5 der Java Standard Edition (SE 5) und stellen Informationen dar, die zur vollständigen Beschreibung eines Programmes notwendig sind, sich aber in Java nicht ausdrücken lassen. Annotationen gestatten Ihnen, zusätzliche Informationen über Ihr Programm in einem Format zu speichern, das vom Compiler geprüft und verifiziert wird. Sie können Annotationen beispielsweise nutzen, um

[1] Jeremy Meyer hat mich in Crested Butte besucht und zwei Wochen mit mir an diesem Kapitel gearbeitet. Seine Hilfe war von unschätzbarem Wert.

Deployment-Deskriptoren und sogar neue Klassendefinitionen zu generieren und erleichtern das Anlegen von Quelltextvorlagen. Mit Annotationen verbleiben diese Metadaten im Java-Quelltext und Sie haben die Vorteile des saubereren Quelltextes, der Typprüfung zur Übersetzungszeit und einer Programmierschnittstelle, zur Unterstützung der Verarbeitung Ihrer Annotationen auf Ihrer Seite. Die SE 5 definiert zwar einige Annotationstypen vor, aber in der Regel bestimmen Sie den Typ und die Auswertung Ihrer Annotationen selbst.

[3–4] Annotationen sind syntaktisch relativ einfach und erweitern die Sprache lediglich um das `@`-Symbol. Die SE 5 beinhaltet drei universelle eingebaute Annotationen, die im Package `java.lang` definiert sind:

- `@Override` zeigt an, daß eine Methodendefinition eine Methode der Basisklasse überschreiben soll. Die Annotation bewirkt, daß der Compiler eine Fehlermeldung ausgibt, wenn Sie versehentlich den Methodennamen falsch schreiben oder die Signatur fehlerhaft wiedergeben.[2]
- `@Deprecated` veranlaßt den Compiler, eine Warnung auszugeben, wenn ein mit dieser Annotation markiertes Element verwendet wird.
- `@SuppressWarnings` schaltet unangebrachte Warnungen ab. Diese Annotation ist vor der SE 5 zwar erlaubt, wird aber nicht unterstützt (sondern ignoriert).

Vier weitere Annotationstypen unterstützen das Definieren neuer Annotationen und werden in Unterabschnitt 21.1.2 vorgestellt.

[5] Immer wenn Sie *descriptor classes or interfaces* anlegen, die viele sich wiederholende Arbeitsschritte enthalten, können Sie in der Regel Annotationen verwenden, um den Prozeß zu automatisieren und zu vereinfachen. Beispielsweise wurde ein großer Teil der Arbeit bei den Enterprise JavaBeans (EJBs) durch den Einsatz von Annotationen in Version EJB 3.0 eliminiert.

[6] Annotation können vorhandene Hilfsprogramme wie XDoclet ablösen, ein unabhängiges Werkzeug für Doclets (siehe Anhang in *http://www.mindview.net/Books/BetterJava*), welches entwickelt wurde, um annotationsähnliche Doclets zu erzeugen. Annotationen sind dagegen ein echter Bestandteil der Sprache, sind strukturiert und werden zur Übersetzungszeit typgeprüft. Das Erfassen aller Informationen im eigentlichen Quelltext, statt in Kommentaren, vermittelt dem Quelltext eine saubere Erscheinungsform und erleichtert die Pflege. Die Anwendung und Erweiterung der Annotations-API und Hilfsmittel beziehungsweise externe Bibliotheken zur Manipulation des Bytecodes liefern mächtige Untersuchungs- und Eingriffsmöglichkeiten für Quell- und Bytecode.

21.1 Grundsätzliche Syntax

[7] Im folgenden Beispiel ist die Methode `testExecute()` mit der Annotation `@Test` bezeichnet. Die Annotation hat keine eigene Funktionalität, aber der Compiler garantiert, daß Ihr Klassenpfad eine Definition für `@Test` enthält. Sie lernen in Unterabschnitt 21.5.3, wie Sie mit Hilfe des Reflexionsmechanismus' die mit `@Test` annotierten Methoden aufrufen können:

```
//: annotations/Testable.java
package annotations;
import net.mindview.atunit.*;

public class Testable {
    public void execute() {
```

[2] Die Anregung stammt zweifellos von einer ähnlichen Fähigkeit bei C#. Die C#-Fähigkeit ist ein Schlüsselwort, statt einer Annotation und wird vom Compiler erzwungen, das heißt beim Überschreiben einer Methoden C# müssen Sie das Schlüsselwort `override` angeben, während die `@Override`-Annotation bei Java optional ist.

```
        System.out.println("Executing..");
    }
    @Test void testExecute() { execute(); }
} ///:~
```

Annotierte Methoden unterscheiden sich nicht von gewöhnlichen Methoden. Die Annotation `@Test` aus diesem Beispiel kann mit jedem Modifikator wie `public`, `static` oder `void` kombiniert werden. Annotationen werden syntaktisch weitestgehend wie Modifikatoren verwendet.

21.1.1 Definition von Annotationen

[8] Das nächste Beispiel zeigt die Definition der obigen Annotation. Sie sehen, daß die Definition einer Annotation der Definition eines Interfaces ähnelt. Der Compiler übersetzt Annotationen analog zu Interfaces in Klassendateien (*.class* Dateien):

```
//: net/mindview/atunit/Test.java
// The @Test tag.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Test {} ///:~
```

Abgesehen vom Symbol `@` sieht die Definition von `@Test` wie die Definition eines leeren Interfaces aus. Die Definition einer Annotation beinhaltet stets die *Metaannotationen* `@Target` und `@Retention`. `@Target` beschreibt, wo Sie die definierte Annotation verwenden können (beispielsweise bei einer Methode oder einem Feld). `@Retention` beschreibt ob die definierte Annotation im Quelltext (`source`), in der Klassendatei (`class`) oder zur Laufzeit (`runtime`) verfügbar sein soll.

[9–10] Annotationen haben in der Regel **Elemente** in Form von Name/Wert-Paaren. Ein Programm oder Werkzeug kann diese Parameter beim Verarbeiten Ihrer Annotationen auswerten. Elemente wirken optisch wie in einem Interface deklarierte Methoden, können aber mit Standardwerten versehen werden. Eine Annotation ohne Elemente, wie `@Test` im obigen Beispiel, heißt **Markierungsannotation** (*marker annotation*).

[11] Das folgende Beispiel zeigt eine einfache Annotation zur Beobachtung der Anwendungsfälle (*use cases*) eines Projektes. Die Programmierer annotieren jede Methode beziehungsweise jeden Satz von Methoden, der die Anforderungen eines Anwendungsfalls erfüllt. Ein Projektleiter kann sich durch Zählen der implementierten Anwendungsfälle einen Eindruck über den Fortschritt des Projektes verschaffen und die für die spätere Pflege des Projektes verantwortlichen Entwickler können Anwendungsfälle schnell finden, wenn Geschäftsregeln (*business rules*) im System geändert oder auf Fehler untersucht werden müssen:

```
//: annotations/UseCase.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface UseCase {
    public int id();
    public String description() default "no description";
} ///:~
```

Beachten Sie, daß die Elemente `id` und `description` an Methodendeklarationen erinnern. Da der Compiler `id` einer Typprüfung unterzieht, kann sowohl die Dokumentation der Anwendungsfälle als

auch der Quelltext zuverlässig mit einer tracking database verknüpft werden. Das Element `description` hat einen **Standardwert**, den der Annotationsprozessor verwendet, wenn beim Annotieren einer Methode kein anderer Wert angegeben wird.

[12] Die folgende Klasse annotiert drei Methoden mit `@UseCase`:

```
//: annotations/PasswordUtils.java
import java.util.*;

public class PasswordUtils {
    @UseCase(id = 47,
             description = ''Passwords must contain at least one numeric'')
    public boolean validatePassword(String password) {
        return (password.matches(''\\w*\\d\\w*''));
    }
    @UseCase(id = 48)
    public String encryptPassword(String password) {
        return new StringBuilder(password).reverse().toString();
    }
    @UseCase(id = 49,
             description = ''New passwords can't equal previously used ones'')
    public boolean
        checkForNewPassword(List<String> prevPasswords, String password) {
        return !prevPasswords.contains(password);
    }
} ///:~
```

Die Elemente der Annotation werden als Name/Wert-Paare in Klammern nach dem Namen `@UseCase` deklariert. Die Annotation bei `encryptPassword()` enthält keinen Beschreibungstext, so daß der in `@Interface @UseCase` festgelegte Standardwert verwendet wird, wenn die Klasse einen Annotationsprozessor durchläuft.

[13] Sie könnten ein solches System verwenden, um eine Anwendung zu entwerfen und die Funktionalität beim Aufbau nach und nach einzusetzen.

21.1.2 Meta-Annotationen

[14] Java hat zur Zeit nur drei Standardannotationen (siehe Kapiteleinleitung) und vier Metaannotationen. Die Metaannotationen dienen zum Annotieren von Annotationen:

- `@Target` gibt an, wo die definierte Annotation verwendet werden kann. Der Aufzählungstyp `java.lang.annotation.ElementType` definiert die folgenden zulässigen Konstanten:
  - `constructor`: bei Konstruktordefinitionen.
  - `field`: bei Felddefinitionen (inklusive Aufzählungstypen).
  - `local_variable`: bei der Definition lokaler Variablen.
  - `method`: bei Methodendefinitionen.
  - `package`: bei `package`-Anweisungen.
  - `parameter`: bei Parameterdefinitionen.
  - `type`: bei Klassen- und Interfacedefinitionen (inklusive Annotationen) sowie bei der Definition von Aufzählungstypen.

- `@Retention` gibt an, wie lange die durch die definierte Annotation hinterlegte Information verfügbar ist. Der Aufzählungstyp `java.lang.annotation.RetentionPolicy` definiert die folgenden zulässigen Konstanten:
  - `source`: Die Annotationen werden vom Compiler entfernt.
  - `class`: Der Compiler beläßt die Annotationen in der Klassendatei, aber sie werden von der Laufzeitumgebung entfernt.
  - `runtime`: Die Annotationen sind zur Laufzeit verfügbar und können mit Hilfe des Reflexionsmechanimus abgefragt werden.
- `@Documented` gibt an, daß die Annotation in der Javadoc-Dokumentation angezeigt werden soll.
- `@Inherited` gestattet abgeleiteten Klassen, Annotationen ihrer Basisklasse zu erben.

Sie werden zumeist eigene Annotationen definieren und eigene Prozessoren schreiben, um sie zu verarbeiten.

21.2 Entwicklung eines Annotationsprozessors

[15] Annotationen sind ohne Hilfsmittel zu ihrer Auswertung nicht nützlicher als Kommentare. Ein wichtiger Anteil des Nutzungsprozesses von Annotationen besteht darin, Annotationsprozessoren zu entwickeln. Die SE 5 beinhaltet Erweiterungen der Programmierschnittstelle des Reflexionsmechanismus', um Sie beim Entwickeln solcher Hilfsmittel zu unterstützen. Die SE 5 liefert außerdem ein externes Werkzeug namens `apt` zum Parsen von Java-Quelltext mit Annotationen.

[16] Das folgende Beispiel ist ein sehr einfacher Prozessor für Annotationen, der die annotierte Klasse `PasswordUtils` einliest und per Reflexionsmechanismus nach `@UseCase`-Tags sucht. Das Programm wertet die `id`-Elemente aus und zeigt an welche Anwendungsfälle gefunden wurden, sowie welche Fälle fehlen:

```
//: annotations/UseCaseTracker.java
import java.lang.reflect.*;
import java.util.*;

public class UseCaseTracker {
    public static void trackUseCases(List<Integer> useCases, Class<?> cl) {
        for(Method m : cl.getDeclaredMethods()) {
            UseCase uc = m.getAnnotation(UseCase.class);
            if(uc != null) {
                System.out.println("Found Use Case:" + uc.id() +
                                   " " + uc.description());
                useCases.remove(new Integer(uc.id()));
            }
        }
        for(int i : useCases) {
            System.out.println("Warning: Missing use case-" + i);
        }
    }
    public static void main(String[] args) {
        List<Integer> useCases = new ArrayList<Integer>();
        Collections.addAll(useCases, 47, 48, 49, 50);
        trackUseCases(useCases, PasswordUtils.class);
    }
```

```
} /* Output:
      Found Use Case:47 Passwords must contain at least one numeric
      Found Use Case:48 no description
      Found Use Case:49 New passwords can't equal previously used ones
      Warning: Missing use case-50
   *///:~
```

Die Methode `trackUseCase()` stützt sich sowohl auf die `Class`-Methode `getDeclaredMethods()` (Reflexion) als auch auf die im Interface *`java.lang.reflect.AnnotatedElement`* deklarierte Methode `getAnnotation()` (Klassen wie `Class`, `Method` und `Field` implementieren *`AnnotatedElement`*). Die `getAnnotation()`-Methode gibt ein Objekt vom Typ *`Annotation`* zurück, das die für die entsprechende Methode deklarierte Annotation des angeforderten Typs (hier `@UseCase`) repräsentiert. Ist bei der Methode keine Annotation dieses Typs deklariert, so gibt die Methode `null` zurück. Die Elementwerte werden mit Hilfe der Methoden `id()` und `description()` abgefragt. Denken Sie daran, daß die `@UseCase`-Annotation bei der Methode `encryptPassword()` keine Beschreibung definiert, so daß der Prozessor beim Aufrufen der `description()`-Methode dieser Annotation den Standardwert „no description“ vorfindet.

21.2.1 Elemente von Annotationen

[17] Die im Beispiel *UseCase.java* auf Seite 821 definierte Annotation `@UseCase` hat das `int`-Element `id` und das `String`-Element `description`. Die folgenden Typen sind bei Elementen von Annotationen erlaubt:

- Alle primitiven Typen (`int`, `float`, `boolean` und so weiter),
- `String`,
- `Class`,
- Aufzählungstypen,
- Annotationen sowie
- Arrays der aufgezählten Typen.

Bei Verwendung eines anderen Typs gibt der Compiler eine Fehlermeldung aus. Beachten Sie, daß die Wrapperklassen der primitiven Typen nicht erlaubt sind, aber durch Autoboxing ist dieses Verbot keine Einschränkung. Elemente können wiederum Annotationen sein. Geschachtelte Annotation können sehr nützlich sein (siehe Unterabschnitt 21.2.3).

21.2.2 Einschränkungen bei Elementwerten

[18] Der Compiler ist bei Standardwerten für Elemente ziemlich pingelig. Kein Element darf einen nicht genau angegeben Wert haben. Ein Element muß somit entweder Standardwerte haben oder in der Klasse bewertet werden, in der die Annotation auftritt.

[19] Es gibt noch eine Einschränkung: Ein Element nicht-primitiven Typs darf nicht mit `null` bewertet werden, weder bei Verwendung der Annotation im Quelltext noch als Standardwert bei der Definition der Annotation. Dies erschwert das Schreiben eines Prozessors, der die An- beziehungsweise Abwesenheit eines Elementes feststellen soll, da effektiv jedes Element in der Definition einer Annotation stets anwesend ist. Eine Lösungsmöglichkeit ist das Prüfen auf bestimmte Werte, wie die leere Zeichenkette oder negative Werte:

```
//: annotations/SimulatingNull.java
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SimulatingNull {
    public int id() default -1;
    public String description() default '''';
} ///:~
```

Dieses Beispiel zeigt eine typische Annotationsdefinition.

21.2.3 Generieren externer Dateien

[20] Annotationen sind besonders bei der Arbeit mit Frameworks nützlich, die auf zusätzliche, Ihren Quelltext begleitende Informationen angewiesen sind. Technologien wie die Enterprise JavaBeans vor Version 3, benötigen Quelltextvorlagen in Form zahlreicher Interfaces und Deployment-Deskriptoren, die bei jeder JavaBean in derselben Weise definiert sind. Webservices, anwendungsspezifische Tagbibliotheken und Werkzeuge im Kontext objektrelationaler Abbildungen wie Toplink und Hibernate benötigen häufig Deskriptoren im XML-Format, die außerhalb des Quelltextes gepflegt werden. Nach der Entwicklung einer Java-Klasse muß der Programmierer Informationen wie den Klassennamen und die Packagezuordnung, Informationen also, die bereits in der Klasse selbst vorhanden sind, ein weiteres mal angegeben. Wenn Sie eine externe Deskriptordatei verwenden, haben Sie zwei Informationsquellen über eine Klasse zu pflegen, was in der Regel Synchronisierungsprobleme mit sich bringt. Außerdem müssen die am Projekt beteiligten Programmierer neben dem Entwicklen von Java-Programmen auch das Editieren des Deskriptors beherrschen.

[21] Angenommen, Sie möchten eine einfache objektrelationale Abbildungsfunktionalität implementieren, die das Anlegen einer Datenbanktabelle zum Speichern einer JavaBean automatisiert. Sie könnten eine Deskriptordatei im XML-Format wählen, um den Klassennamen, alle Komponenten sowie Informationen hinsichtlich der Abbildung in die Datenbank dort hinterlegen. Mit Annotationen können Sie dagegen sämtliche Informationen in den Quelltext der JavaBean eintragen. Zu diesem Zweck brauchen Sie Annotationen, um den Namen der mit der JavaBean verknüpften Datenbanktablle, die Tabellenspalten und die zum Speichern der Eigenschaften der JavaBean zu verwendenden SQL-Typen zu definieren.

[22] Die folgende Annotation für JavaBeans teilt dem Annotationsprozessor mit, daß er eine Datenbanktabelle anlegen soll:

```
//: annotations/database/DBTable.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.TYPE) // Applies to classes only
    @Retention(RetentionPolicy.RUNTIME)
    public @interface DBTable {
        public String name() default '';
    } ///:~
```

Der in der `@Target`-Annotation definierte Elementtyp (`ElementType.TYPE`) legt eine Einschränkung fest, die dem Compiler mitteilt, daß Annotationen dieses Typs nur bei dem festgelegten Komponententyp verwendet werden dürfen. Sie können eine einzelne Konstante des Aufzählungstyps `ElementType` wählen, oder eine kommaseparierte Liste mit einer beliebigen Kombination von Konstanten. Soll die Annotation bei jedem Komponententyp erlaubt werden, so können Sie die `@Target`-Annotation fortlassen (das Auslassen ist allerdings unüblich).

[23] Beachten Sie das Element `name` der `@DBTable`-Annotation, mit dessen Hilfe die Annotation den Namen der Datenbanktabelle angibt, die der Prozessor erzeugen soll.

[24] Die Annotationen für die Eigenschaften der JavaBean lauten:

```
//: annotations/database/Constraints.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Constraints {
    boolean primaryKey() default false;
    boolean allowNull() default true;
    boolean unique() default false;
} ///:~

//: annotations/database/SQLString.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLString {
    int value() default 0;
    String name() default "";
    Constraints constraints() default @Constraints;
} ///:~

//: annotations/database/SQLInteger.java
package annotations.database;
import java.lang.annotation.*;

@Target(ElementType.FIELD)
@Retention(RetentionPolicy.RUNTIME)
public @interface SQLInteger {
    String name() default "";
    Constraints constraints() default @Constraints;
} ///:~
```

Die `@Constraints`-Annotation beschreibt Metadaten der Datenbanktabelle. `@Constraints` repräsentiert zwar nur einen kleinen Teil der bei Datenbanken üblicherweise verfügbaren Einschränkungen, vermittelt aber das unterliegende Konzept. Die Elemente `primaryKey`, `allowNull` und `unique` haben sinnvolle Standardwerte, so daß Sie beim Anwenden der Annotation nicht zu viel schreiben müssen.

[25] Die beiden anderen Annotationen definieren SQL-Typen. Diese kleine Bibliothek wird umso nützlicher, wenn Sie für jeden SQL-Typ eine weitere Annotation definieren. Für unser Beispiel reichen die beiden obigen SQL-Typen aus.

[26] Sowohl `@SQLString` als auch `@SQLInteger` haben je ein `name`- und ein `constraints`-Element. Das letztere nutzt die Schachtelung von Annotationen, um die Einschränkungen für den Spaltentyp zu erfassen. Beachten Sie den Standardwert `@Constraints` des `constraints`-Elementes. Da die `@Constraints`-Annotion bei `default` keine eingeklammerte Elementliste hat, ist der Standardwert des `constraints`-Elementes eine `@Constraints`-Annotion mit ihren eigenen Standardwerten. Das folgende Beispiel zeigt, wie Sie das Element `unique` der geschachtelten `@Constraints`-Annotation mit `true` bewerten können:

```
//: annotations/database/Uniqueness.java
// Sample of nested annotations
```

```
package annotations.database;

public @interface Uniqueness {
    Constraints constraints() default @Constraints(unique=true);
} ///:~
```

Die folgende einfache JavaBean verwendet die oben definierten Annotationen:

```
//: annotations/database/Member.java
package annotations.database;

@DBTable(name = ''MEMBER'')
public class Member {
    @SQLString(30) String firstName;
    @SQLString(50) String lastName;
    @SQLInteger Integer age;
    @SQLString(value = 30, constraints = @Constraints(primaryKey = true))
    String handle;
    static int memberCount;
    public String getHandle() { return handle; }
    public String getFirstName() { return firstName; }
    public String getLastName() { return lastName; }
    public String toString() { return handle; }
    public Integer getAge() { return age; }
} ///:~
```

Der `name`-Elementwert „MEMBER“ der Klassenannotation `@DBTable` wird als Name der Datenbanktabelle verwendet. Die Eigenschaften `firstName` und `lastName` der JavaBean sind beide mit `@SQLString` und den Werten 30 beziehungsweise 50 annotiert. Die beiden `@SQLString`-Annotationen sind aus zwei Gründen interessant: Erstens verwenden sie die Standardwerte der geschachtelten Annotation `@Constraints` und zweitens demonstrieren sie eine abgekürzte Schreibweise. Definiert eine Annotation ein Element namens `value`, so brauchen Sie, solange Sie keine weiteren Elemente zugleich bewerten, keine Schlüssel/Wert-Syntax anzugeben, sondern die Übergabe des Wertes in Klammern genügt. Diese Regel gilt für jeden der zulässigen Elementtypen. Sie sind dadurch zwar an den Elementnamen `value` gebunden, andererseits gestattet diese Regel im obigen Beispiel eine semantisch aussagekräftige und leicht lesbare Schreibweise:

```
@SQLString(30)
```

Der Prozessor wird diesen Wert verwenden, um die Breite der entsprechenden Spalte in der Datenbanktabelle zu wählen. Trotz dieser ansprechenden Notation wird die Syntax schnell kompliziert. Betrachten Sie zum Beispiel die Annotation der Eigenschaft `handle`. Die Eigenschaft ist einerseits mit `@SQLString` annotiert, dient aber andererseits auch als Schlüsselfeld in der Datenbanktabelle, so daß das Element `primaryKey` der geschachtelten `@Constraints`-Annotation auf `true` gesetzt werden muß. Ab diesem Punkt wird die Syntax unübersichtlich. Sie sind gezwungen, bei der geschachtelten Annotation die weitschweifige Schlüssel/Wert-Syntax zu verwenden und sowohl den Elementnamen als auch den Namen der Annotation anzugeben. Da das Element mit dem speziellen Namen `value` nicht mehr das einzige bewertete Element ist, scheidet die abgekürzte Schreibweise aus. Wie Sie sehen, ist das Ergebnis nicht mehr attraktiv.

21.2.3.1 Alternative Lösungsansätze

[27] Es gibt noch andere Möglichkeiten, um Annotationen für diese Aufgabe zu definieren. Sie könnten beispielsweise eine einzelne Annotation `@TableColumn` definieren, die einen Aufzählungstyp mit Werten wie `STRING`, `INTEGER` oder `FLOAT` definiert. Dadurch umgehen Sie die Notwendigkeit

einer Annotation für jeden SQL-Typ, versperren aber die Möglichkeit, Ihre Typen mit zusätzlichen Elementen wie etwa `size` oder `precision` auszustatten, die unter Umständen nützlicher wären.

[28] Sie könnten den tatsächlichen SQL-Typ per `String`-Element beschreiben, zum Beispiel `VARCHAR(30)` oder `INTEGER`. Dieser Ansatz gestattet zwar die Angabe der SQL-Typen, legt aber auch die Abbildung der Java-Typen spezifisch auf SQL-Typen fest und ist kein gutes Design, da Sie beim Wechsel zu einer anderen Datenbank den Quelltext nicht neu übersetzen möchten. Eine elegantere Lösung wäre, dem Annotationsprozessor mitzuteilen, daß Sie einen anderen SQL-Dialekt verwenden wollen und dies beim Verarbeiten der Annotationen zu berücksichtigen.

[29] Eine dritte funktionstüchtige Lösung besteht darin, eine Kombination aus den beiden Annotationen `@Constraints` und dem erforderlichen SQL-Typ (etwa `@SQLInteger`) zu verwenden, um das gewünschte Feld zu annotieren. Das ist zwar ein wenig unübersichtlich, aber der Compiler gestattet beliebig viele Annotationen pro Komponente. Beachten Sie bei Verwendung mehrerer Annotationen, daß Sie keine Annotation bezüglich einer Komponente zweimal setzen können.

21.2.4 Annotationen unterstützen keine Vererbung

[30] Das Schlüsselwort `extends` ist bei der Definition einer Annotation (`@Interface`) leider nicht erlaubt, da somit eine elegante Lösung ausscheidet, nämlich eine `@TableColumn`-Annotation, wie zuvor beschrieben, mit einer geschachtelten Annotation `@SQLType`, wobei die Annotationen für alle SQL-Typen, etwa `@SQLInteger` und `@SQLString` von `@SQLType` abgeleitet werden würden. Ein solcher Ansatz würde die Schreibarbeit reduzieren und die Syntax verbessern. Es scheint keine Anregung in die Richtung zu geben, daß Annotationen in zukünftigen Sprachversionen Ableitung unterstützen, so daß die obigen Beispiele unter den gegebenen Umständen wohl die beste Lösung sind.

21.2.5 Implementierung des Prozessors

[31] Das folgende Programm ist ein Annotationsprozessor, der eine Klassendatei einliest, nach Datenbankannotationen sucht und das SQL-Kommando generiert, um die erforderliche Datenbanktabelle anzulegen:

```
//: annotations/database/TableCreator.java
// Reflection-based annotation processor.
// {Args: annotations.database.Member}
package annotations.database;
import java.lang.annotation.*;
import java.lang.reflect.*;
import java.util.*;

public class TableCreator {
    public static void main(String[] args) throws Exception {
        if(args.length < 1) {
            System.out.println("arguments: annotated classes");
            System.exit(0);
        }
        for(String className : args) {
            Class<?> cl = Class.forName(className);
            DBTable dbTable = cl.getAnnotation(DBTable.class);
            if(dbTable == null) {
                System.out.println("No DBTable annotations in class " + className);
                continue;
            }
            String tableName = dbTable.name();
```

```
            // If the name is empty, use the Class name:
            if(tableName.length() < 1)
                tableName = cl.getName().toUpperCase();
            List<String> columnDefs = new ArrayList<String>();
            for(Field field : cl.getDeclaredFields()) {
                String columnName = null;
                Annotation[] anns = field.getDeclaredAnnotations();
                if(anns.length < 1)
                    continue; // Not a db table column
                if(anns[0] instanceof SQLInteger) {
                    SQLInteger sInt = (SQLInteger) anns[0];
                    // Use field name if name not specified
                    if(sInt.name().length() < 1)
                        columnName = field.getName().toUpperCase();
                    else
                        columnName = sInt.name();
                    columnDefs.add(columnName + '' INT'' +
                                   getConstraints(sInt.constraints()));
                }
                if(anns[0] instanceof SQLString) {
                    SQLString sString = (SQLString) anns[0];
                    // Use field name if name not specified.
                    if(sString.name().length() < 1)
                        columnName = field.getName().toUpperCase();
                    else
                        columnName = sString.name();
                    columnDefs.add(columnName + '' VARCHAR('' +
                                   sString.value() + ''))'' +
                                   getConstraints(sString.constraints()));
                }
                StringBuilder createCommand =
                    new StringBuilder(''CREATE TABLE '' + tableName + ''('');
                for(String columnDef : columnDefs)
                    createCommand.append(''\n '' + columnDef + '','');
                // Remove trailing comma
                String tableCreate =
                    createCommand.substring(0, createCommand.length() - 1) + '');'';
                System.out.println(''Table Creation SQL for '' +
                                   className + '' is :\n'' + tableCreate);
            }
        }
    }
    private static String getConstraints(Constraints con) {
        String constraints = '''';
        if(!con.allowNull())
            constraints += '' NOT NULL'';
        if(con.primaryKey())
            constraints += '' PRIMARY KEY'';
        if(con.unique())
            constraints += '' UNIQUE'';
        return constraints;
    }
} /* Output:
      Table Creation SQL for annotations.database.Member is :
      CREATE TABLE MEMBER(
      FIRSTNAME VARCHAR(30));
      Table Creation SQL for annotations.database.Member is :
```

```
    CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50));
    Table Creation SQL for annotations.database.Member is :
    CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT);
    Table Creation SQL for annotations.database.Member is :
    CREATE TABLE MEMBER(
    FIRSTNAME VARCHAR(30),
    LASTNAME VARCHAR(50),
    AGE INT,
    HANDLE VARCHAR(30) PRIMARY KEY);
*///:~
```

Die `main()`-Methode verarbeitet der Reihe nach die auf der Kommandozeile übergebenen Klassennamen. Die `forName()`-Methode gibt das Klassenobjekt der jeweiligen Klasse zurück. Danach prüft die `main()`-Methode per `getAnnotation(DBTable.class)`, ob die untersuchte Klasse mit `@DBTable` annotiert ist. Ist die `@DBTable`-Annotation vorhanden, so wird der Tabellenname abgefragt und gespeichert. Anschließend werden alle Felder der Klasse mit Hilfe der Methode `getDeclaredAnnotations()` untersucht. Die Methode gibt ein Array der beim jeweiligen Feld deklarierten Annotationen zurück. Der `instanceof`-Operator wird verwendet, um zu prüfen, ob diese Annotationen vom Typ `@SQLInteger` beziehungsweise `@SQLString` sind, woraufhin der entsprechende Bestandteil der Spaltendefinition in der Datenbanktabelle erzeugt wird. Beachten Sie, daß sich polymorphes Verhalten, aufgrund der fehlenden Unterstützung des Ableitungsmechanismus bei Annotationen, nur per `getDeclaredAnnotations()` näherungsweise emulieren läßt.

[32] Die geschachtelte `@Constraints`-Annotation wird mit Hilfe der Methode `getConstraints()` ausgewertet, die ein `String`-Objekt mit den SQL-Einschränkungen für die jeweilige Spalte zurückgibt.

[33] Es soll darauf hingewiesen werden, daß der obige Ansatz ein etwas naives Verfahren zur Definition einer objektrelationalen Abbildung darstellt. Eine Annotation vom Typ `@DBTable`, die den Tabellennamen als Parameter erwartet, zwingt Sie, bei jeder Änderung des Tabellennamens den Quelltext neu zu übersetzen. Das ist nicht wünschenswert. Es gibt viele Frameworks, um Objekte auf relationale Datenbanken abzubilden und immer mehr dieser Frameworks machen von Annotationen Gebrauch.

Übungsaufgabe 1: (2) Implementieren Sie im obigen Datenbankbeispiel Annotationen für weitere SQL-Typen. ■

Projekt[3]**:** Ändern Sie das Datenbankbeispiel, so daß es sich per JDBC mit einer echten Datenbank verbinden läßt und mit dieser kommuniziert. ■

Projekt: Ändern Sie das Datenbankbeispiel, so daß es konforme XML-Dateien statt SQL-Anweisungen generiert. ■

[3] Projekte sind Vorschläge für Semester- oder Halbjahresarbeiten. Der *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können, enthält *keine* Lösungsvorschläge für Projekte.

21.3 Der Annotationsprozessor apt von Sun

[34] Das Hilfsprogramm `apt` („annotation processing tool“) ist die erste Version eines Werkzeuges zur Verarbeitung von Annotationen. Als erste Inkarnation ist das Programm noch immer ein wenig primitiv, bietet aber Eigenschaften und Fähigkeiten, die Ihnen das Leben erleichtern.

[35] Das Hilfsprogramm `apt` verarbeitet wie `javac` Quelltextdateien statt übersetzter Klassen. Im Standardverhalten übersetzt `apt` die Quelltextdateien nach der Verarbeitung. Dieses Verhalten ist nützlich, wenn Sie während des Übersetzungsvorgangs automatisch neue Quelltextdateien generieren. `apt` prüft vielmehr neu erzeugte Quelltextdateien auf Annotationen und übersetzt sie alle im selben Durchgang.

[36] Generiert Ihr Annotationsprozessor eine neue Quelltextdatei, so wird diese Datei in einer neuen Verarbeitungsrunde (im Sinne der Dokumentation) wiederum auf Annotationen geprüft. Das Programm setzt die Verarbeitung Runde um Runde fort, bis keine Quelltextdateien mehr generiert werden. Anschließend werden alle Quelltextdateien übersetzt.

[37] Sie brauchen einen Prozessor für jede Ihrer selbstgeschriebenen Annotationen. Das Hilfsprogramm `apt` ist aber in der Lage, verschiedene Annotationsprozessoren zusammenzufassen. Auf diese Weise können Sie viele Klassen zur Verarbeitung auswählen und haben eine erheblich einfachere Lösung, als alle Klassen mit Hilfe von `File`-Objekten selbst zu durchsuchen. Außerdem können Sie Ereignisbehandler registrieren, um benachrichtigt zu werden, wenn eine Verarbeitungsrunde beendet ist.

[38] Zum Zeitpunkt der Drucklegung der amerikanischen Originalausgabe der vierten Auflage dieses Buches, war `apt` noch nicht als Ant-Task verfügbar (siehe Anhang in *http://www.mindview.net/Books/BetterJava*), konnte aber in der Zwischenzeit von Ant als externer Prozeß aufgerufen werden.[4] Das Übersetzen der Annotationsprozessoren in diesem Beispiel setzt voraus, daß sich die Datei *tools.jar* in Ihrem Klassenpfad befindet. Diese Bibliothek enthält die Interfaces in den Packages `com.sun.mirror.*`.

[39] `apt` bedient sich eines Fabrikobjektes vom Typ *`com.sun.mirror.apt.AnnotationProcessorFactory`*, um für jede vorgefundene Annotation den richtigen Annotationsprozessor zu erzeugen. Sie geben beim Aufruf von `apt` entweder eine Fabrikklasse oder einen Klassenpfad an, in dem `apt` die benötigten Klassen findet. Andernfalls begibt sich `apt` auf eine geheimnisvolle Entdeckungsreise,

[4] Anmerkung des Übersetzers: Aus dem Ant-Handbuch (*http://ant.apache.org/manual/index.html*): „Runs the annotation processor tool (`apt`), and then optionally compiles the original code, and any generated source code. This task requires Version 5 der Java Standard Edition. It may work on later versions, but this cannot be confirmed until those versions ship. Be advised that the Apt tool does appear to be an unstable part of the JDK framework, so may change radically in future versions. In particular it is likely to be obsolete in JDK 6, which can run annotation processors as part of `javac`. If the `<apt>` task does break when upgrading JVM, please check to see if there is a more recent version of Ant that tracks any changes.

This task inherits from the `Javac` Task, and thus supports nearly all of the same attributes, and subelements. There is one special case, the `fork` attribute, which is present but which can only be set to `true`. That is, `apt` only works as a forked process.“

Übersetzt etwa: „Ruft den Annotationsprozessor `apt` („annotation processor tool“) auf und übersetzt optional den ursprünglichen Quelltext sowie alle generierten Quelltextdateien. Dieses Task erfordert Version 5 der Java Standard Edition und wird eventuell auch bei Folgeversionen funktionieren, wobei die Funktionstüchtigkeit erst bestätigt werden kann, wenn diese Versionen verfügbar sind. Beachten Sie, daß `apt` ein instabiler Bestandteil des JDK-Frameworks zu sein scheint, sich also bei zukünftigen Versionen erheblich verändern kann. Insbesondere ist `apt` in JDK 6 wahrscheinlich überholt, da Annotationsprozessoren in dieser Sprachversion per `javac` aufgerufen werden können. Sollte das Task `<apt>` nach dem Upgrade Ihrer Laufzeitumgebung nicht mehr funktionieren, so prüfen Sie bitte nach, ob es eine neue Ant-Version gibt, die alle Änderungen berücksichtigt.

Das Task `<apt>` erbt von `<javac>` und unterstützt somit fast alle Attribute und Unterelemente. Es gibt einen Spezialfall: Das Attribut `fork` ist zwar vorhanden, kann aber nur mit `true` bewertet werden, das heißt `apt` läuft ausschließlich als abgespalteter Kindprozeß.“

deren Einzelheiten Sie im Abschnitt *Developing an Annotation Processor* in der Dokumentation von Sun nachlesen können (*http://java.sun.com/j2se/1.5.0/docs/guide/apt/GettingStarted.html*).

[40] Wenn Sie einen Annotationsprozessor zur Verknüpfung mit `apt` schreiben, steht Ihnen der Reflexionsmechanismus von Java nicht zur Verfügung, da Sie Quelltext und keine übersetzten Klassen verarbeiten.[5] Die Programmierschnittstelle im Package `com.sun.mirror`[6] löst dieses Problem aber, indem sie Ihnen ermöglicht, Methoden, Felder und Typen in unübersetztem Quelltext auszuwerten.

[41] Die folgende Annotation kann verwendet werden, um die öffentlichen Methoden einer Klasse abzufragen und in einem Interface zusammenzufassen:

```
//: annotations/ExtractInterface.java
// APT-based annotation processing.
package annotations;
import java.lang.annotation.*;

@Target(ElementType.TYPE)
@Retention(RetentionPolicy.SOURCE)
public @interface ExtractInterface {
    public String value();
} ///:~
```

Die Metaannotation `@Retention` ist mit `RetentionPolicy.SOURCE` bewertet, da es keinen Grund gibt, diese Annotation in die Klassendatei zu übernehmen, nachdem die Schnittstelle der Klasse ausgewertet ist. Die folgende Klasse `Multiplier` stellt eine öffentliche Methode zur Verfügung, die Bestandteil eines nützlichen Interfaces sein könnte:

```
//: annotations/Multiplier.java
// APT-based annotation processing.
package annotations;

@ExtractInterface("IMultiplier")
public class Multiplier {
    public int multiply(int x, int y) {
        int total = 0;
        for(int i = 0; i < x; i++)
            total = add(total, y);
        return total;
    }
    private int add(int x, int y) { return x + y; }
    public static void main(String[] args) {
        Multiplier m = new Multiplier();
        System.out.println("11*16 = " + m.multiply(11, 16));
    }
} /* Output:
      11*16 = 176
   *///:~
```

Die Klasse `Multiplier` (funktioniert nur bei positiven ganzen Zahlen) hat eine `multiply()`-Methode, welche eine private `add()`-Methode mehrmals aufruft, um die Multiplikation zu bewerkstelligen. Die `add()`-Methode gehört als nicht-öffentliche Methode nicht zur Schnittstelle der Klasse. Der Wert „IMultiplier“ in der Annotation ist der Name des zu generierenden Interfaces: Der folgende Prozessor führt die Abfrage der öffentlichen Methoden durch:

[5] Die nicht standardisierte Option `-XclassesAsDecls` gestattet aber die Auswertung von Annotationen in übersetzten Klassen.

[6] Die Designer von Java weisen kokett darauf hin, daß Sie in einen Spiegel sehen sollten, wenn Sie nach Reflexion suchen.

```
//: annotations/InterfaceExtractorProcessor.java
// APT-based annotation processing.
// {Exec: apt -factory
// annotations.InterfaceExtractorProcessorFactory
// Multiplier.java -s ../annotations}
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.io.*;
import java.util.*;

public class InterfaceExtractorProcessor implements AnnotationProcessor {
    private final AnnotationProcessorEnvironment env;
    private ArrayList<MethodDeclaration> interfaceMethods =
        new ArrayList<MethodDeclaration>();
    public InterfaceExtractorProcessor(AnnotationProcessorEnvironment env) {
        this.env = env;
    }
    public void process() {
        for(TypeDeclaration typeDecl : env.getSpecifiedTypeDeclarations()) {
            ExtractInterface annot =
                typeDecl.getAnnotation(ExtractInterface.class);
            if(annot == null)
                break;
            for(MethodDeclaration m : typeDecl.getMethods())
                if(m.getModifiers().contains(Modifier.PUBLIC) &&
                   !(m.getModifiers().contains(Modifier.STATIC)))
                    interfaceMethods.add(m);
            if(interfaceMethods.size() > 0) {
                try {
                    PrintWriter writer =
                        env.getFiler().createSourceFile(annot.value());
                    writer.println("package " +
                                   typeDecl.getPackage().getQualifiedName() +";");
                    writer.println("public interface " + annot.value() + " {");
                    for(MethodDeclaration m : interfaceMethods) {
                        writer.print(" public ");
                        writer.print(m.getReturnType() + " ");
                        writer.print(m.getSimpleName() + " (");
                        int i = 0;
                        for(ParameterDeclaration parm : m.getParameters()) {
                            writer.print(parm.getType() + " " +
                                         parm.getSimpleName());
                            if(++i < m.getParameters().size()
                                writer.print(", ");
                        }
                        writer.println(");");
                    }
                    writer.println("}");
                    writer.close();
                } catch(IOException ioe) {
                    throw new RuntimeException(ioe);
                }
            }
        }
    }
} ///:~
```

Die gesamte Arbeit wird in der Methode `process()` verrichtet. Die öffentlichen Methode der verarbeiteten Klassen (nicht aber die statischen) werden mit Hilfe der Methode `getModifiers()` der Klasse *`MethodDeclaration`* identifiziert. Gefundene Methoden werden in einem `ArrayList`-Container gespeichert und als Methodendeklarationen in eine *.java* Datei eingesetzt, die ein neues Interface definiert.

[42] Beachten Sie, daß der Konstruktor der Klasse `InterfaceExtractorProcessor` ein Argument vom Typ *`AnnotationProcessorEnvironment`* erwartet. Sie können dieses Objekt zu jedem von `apt` verarbeiteten Typ (Klassendefinition) abfragen. Außerdem können Sie über das *`AnnotationProcessorEnvironment`*-Objekt auch Objekte der Typen *`Filer`* und *`Messager`* anfordern. *`Messager`*-Objekte gestatten die Benachrichtigung des Benutzers, zum Beispiel bei Fehlern während der Verarbeitung und der Position des Fehlers im Quelltext. *`Filer`*-Objekte sind eine Art `PrintWriter` über die Sie neue Dateien anlegen können. Der Hauptgrund für die Verwendung von *`Filer`*- anstelle von `PrintWriter`-Objekten besteht darin, daß `apt` im ersteren Fall die neu angelegten Dateien „im Auge behält“, um sie auf Annotationen überprüfen und gegebenenfalls übersetzen zu können.

[43] Die Methode `createSourceFile()` öffnet einen gewöhnlichen Ausgabestrom mit dem korrekten Namen der Java-Klasse beziehungsweise des Interfaces. Es gibt keine Unterstützung zum Erzeugen von Java-Anweisungen, so daß Sie den Java-Quelltext auf etwas primitive Weise mit Hilfe der Methoden `print()` und `println()` zusammensetzen müssen. Sie müssen also darauf achten, daß Ihren Klammern paarweise zusammenpassen und der Quelltext syntaktisch korrekt ist.

[44] Das Hilfsprogramm `apt` ruft die `process()`-Methode auf, benötigt also eine Fabrikklasse, die den passenden Annotationsprozessor liefert:

```
//: annotations/InterfaceExtractorProcessorFactory.java
// APT-based annotation processing.
package annotations;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import java.util.*;

public class InterfaceExtractorProcessorFactory
    implements AnnotationProcessorFactory {
    public AnnotationProcessor
        getProcessorFor(Set<AnnotationTypeDeclaration> atds,
                        AnnotationProcessorEnvironment env) {
        return new InterfaceExtractorProcessor(env);
    }
    public Collection<String> supportedAnnotationTypes() {
        return Collections.singleton(''annotations.ExtractInterface'');
    }
    public Collection<String> supportedOptions() {
        return Collections.emptySet();
    }
} ///:~
```

Das Interface *`AnnotationProcessorFactory`* deklariert lediglich drei Methoden. Die Methode `getProcessorFor()` erwartet einen *`Set`*-Container mit Typdeklarationen (*`AnnotationTypeDeclaration`*, den von `apt` verarbeiteten Java-Klassen) sowie das *`AnnotationProcessorEnvironment`*-Objekt, das anschließend dem Annotationsprozessor übergeben wird. Die beiden anderen Methoden, `supportedAnnotationTypes()` und `supportedOptions()`, gestatten Ihnen, zu prüfen, ob zu jeder von `apt` gefundenen Annotation ein Prozessor vorhanden ist beziehungsweise ob alle auf der Kommandozeile übergebenen Optionen unterstützt werden. Die Methode `supportedAnnotationTypes()` ist besonders wichtig, da `apt` die Verarbeitung mit der Warnung beendet, daß kein entsprechender Annotationsprozessor existiert, wenn der zurückgegebene *`Collection<String>`*-Container nicht die

vollqualifizierten Klassennamen Ihrer Annotationstypen enthält.

[45] Der Annotationsprozessor und die zugehörige Fabrikklasse liegen im Package `annotations`. Das `Exec`-Tag zu Beginn des Beispiels *InterfaceExtractorProcessor.java* auf Seite 833 dokumentiert den Aufruf von `apt`. Das Kommando weist `apt` an, die oben definierte Fabrikklasse zu verwenden und die Datei *Multiplier.java* auf Seite 832 zu verarbeiten. Die Option `-s` definiert, daß neu erzeugte Dateien im Verzeichnis *annotations* deponiert werden müssen. Der Inhalt der generierten Datei *IMultiplier.java* lautet, wie Sie anhand der `println()`-Anweisungen im obigen Prozessor vielleicht schon vermutet haben:

```
package annotations;
public interface IMultiplier {
    public int multiply (int x, int y);
}
```

Diese Datei wird von `apt` auch übersetzt, so daß Sie auch *IMultiplier.class* in demselben Verzeichnis vorfinden.

Übungsaufgabe 2: (3) ~~Add support for division to the interface extractor.~~ ■

21.4 Kombination des Entwurfsmusters Visitor mit apt

[46] Das Verarbeiten von Annotationen kann eine komplizierte Aufgabe werden. Das obige Beispiel ist ein vergleichsweise einfacher Annotationsprozessor und interpretiert lediglich eine Annotation, verlangt aber einen beträchtlichen Komplexitätsaufwand, um zu funktionieren. Die Programmierschnittstelle im Package `com.sun.mirror` stellt Klassen zur Verfügung, um das Entwurfsmuster *Visitor* zu unterstützen. *Visitor* ist eines klassischen Entwurfsmuster aus Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995).

[47] Ein *Visitor* traversiert eine Datenstruktur oder Kollektion von Objekten und führt auf jedem Element eine Operation aus. Die Datenstruktur braucht nicht geordnet zu sein und die auf einem Element ausgeführte Operation hängt vom Typ des Objektes ab. Dieser Ansatz hebt die Kopplung zwischen Operation und Objekt auf, das heißt Sie können neue Operationen definieren, ohne in den Klassen der traversierten Elemente neue Methoden anlegen zu müssen.

[48] Das Entwurfsmuster *Visitor* eignet sich somit zur Verarbeitung von Annotationen, da Sie sich eine Java-Klasse als Kollektion von Objekten der Typen *`TypeDeclaration`*, *`FieldDeclaration`*, *`MethodDeclaration`* und so weiter vorstellen können. Wenn Sie das Hilfsprogramm `apt` und das Entwurfsmuster *Visitor* kombinieren möchten, müssen Sie eine ~~Visitor~~-Klasse anlegen, die zu jedem „besuchten“ Deklarationstyp eine Behandlermethode definiert. Sie können also entsprechendes Verhalten für Annotationen bei Methoden, Klassen, Feldern und so weiter implementieren.

[49] Das folgende Programm zeigt nochmals das Generieren einer Datenbanktabelle, diesesmal stützen sich Fabrikklasse und Prozessor aber auf das Entwurfsmuster *Visitor*:

```
//: annotations/database/TableCreationProcessorFactory.java
// The database example using Visitor.
// {Exec: apt -factory
// annotations.database.TableCreationProcessorFactory
// database/Member.java -s database}
package annotations.database;
import com.sun.mirror.apt.*;
import com.sun.mirror.declaration.*;
import com.sun.mirror.util.*;
```

```
import java.util.*;
import static com.sun.mirror.util.DeclarationVisitors.*;

public class TableCreationProcessorFactory
    implements AnnotationProcessorFactory {
    public AnnotationProcessor
        getProcessorFor(Set<AnnotationTypeDeclaration> atds,
                        AnnotationProcessorEnvironment env) {
        return new TableCreationProcessor(env);
    }
    public Collection<String> supportedAnnotationTypes() {
        return Arrays.asList("annotations.database.DBTable",
                             "annotations.database.Constraints",
                             "annotations.database.SQLString",
                             "annotations.database.SQLInteger");
    }
    public Collection<String> supportedOptions() {
        return Collections.emptySet();
    }
    private static class TableCreationProcessor implements AnnotationProcessor {
        private final AnnotationProcessorEnvironment env;
        private String sql = "";
        public TableCreationProcessor(AnnotationProcessorEnvironment env) {
            this.env = env;
        }
        public void process() {
            for(TypeDeclaration typeDecl : env.getSpecifiedTypeDeclarations()) {
                typeDecl.accept(
                    getDeclarationScanner(new TableCreationVisitor(), NO_OP));
                sql = sql.substring(0, sql.length() - 1) + ");";
                System.out.println("creation SQL is :\n" + sql);
                sql = "";
            }
        }
        private class TableCreationVisitor extends SimpleDeclarationVisitor {
            public void visitClassDeclaration(ClassDeclaration d) {
                DBTable dbTable = d.getAnnotation(DBTable.class);
                if(dbTable != null) {
                    sql += "CREATE TABLE ";
                    sql += (dbTable.name().length() < 1)
                        ? d.getSimpleName().toUpperCase()
                        : dbTable.name();
                    sql += " (";
                }
            }
            public void visitFieldDeclaration(FieldDeclaration d) {
                String columnName = "";
                if(d.getAnnotation(SQLInteger.class) != null) {
                    SQLInteger sInt = d.getAnnotation(SQLInteger.class);
                    // Use field name if name not specified
                    if(sInt.name().length() < 1)
                        columnName = d.getSimpleName().toUpperCase();
                    else
                        columnName = sInt.name();
                        sql += "\n " + columnName + " INT" +
                        getConstraints(sInt.constraints()) + ",";
                }
                if(d.getAnnotation(SQLString.class) != null) {
```

```
                    SQLString sString = d.getAnnotation(SQLString.class);
                    // Use field name if name not specified.
                    if(sString.name().length() < 1)
                        columnName = d.getSimpleName().toUpperCase();
                    else
                        columnName = sString.name();
                        sql += ''\n '' + columnName + '' VARCHAR('' +
                        sString.value() + '')'' +
                        getConstraints(sString.constraints()) + '','';
                }
            }
            private String getConstraints(Constraints con) {
                String constraints = '''';
                if(!con.allowNull())
                    constraints += '' NOT NULL'';
                if(con.primaryKey())
                    constraints += '' PRIMARY KEY'';
                if(con.unique())
                    constraints += '' UNIQUE'';
                return constraints;
            }
        }
    }
} ///:~
```

Die Ausgabe ist identisch mit dem Beispiel *TableCreator.java* auf Seite 828f.

[50] Prozessor und Besucher sind in diesem Beispiel innere Klassen. Beachten Sie, daß die `process()`-Methode lediglich ein `TableCreationVisitor`-Objekt registriert und das SQL-Kommando mit der leeren Zeichenkette initialisiert.

[51] Beide Parameter der Methode `getDeclarationScanner()` sind vom Typ *`DeclarationVisitor`* („Besucher“). Der erste Parameter wird *vor* dem Besuch einer Deklaration angewendet, der zweite *danach*. Dieser Annotationsprozessor braucht nur den ersten Besucher, so daß die Konstante `DeclarationVisitors.NO_OP` als zweites Argument übergeben wird. `NO_OP` ist ein statisches Feld der Klasse `DeclarationVisitors` und repräsentiert ein *`DeclarationVisitor`*-Objekt ohne Funktionalität.

[52] Die Klasse `TableCreationVisitor` ist von der Klasse `SimpleDeclarationVisitor` abgeleitet und überschreibt die beiden Methoden `visitClassDeclaration()` und `visitFieldDeclaration()`. `SimpleDeclarationVisitor` ist eine Adapterklasse und implementiert alle im Interface *`DeclarationVisitor`* deklarierten Methoden, so daß Sie sich auf die Methoden konzentrieren können, die Sie brauchen. Die Methode `visitClassDeclaration()` untersucht das übergebene *`ClassDeclaration`*-Objekt (Klassendefinition) auf eine `@DBTable`-Annotation und verwendet, falls vorhanden, den Wert des `name`-Elementes, um das Anfangsstück des SQL-Kommandos zusammenzusetzen. Die Methode `visitFieldDeclaration()` untersucht das übergebene *`FieldDeclaration`*-Objekt (Felddefinition) auf eine `@SQLInteger`- oder `@SQLString`-Annotation und verwendet die ausgewertete Information auf dieselbe Weise wie im ursprünglichen Beispiel *TableCreator.java* auf Seite 828f.

[53] Dieser Ansatz wirkt zwar komplizierter, liefert aber eine ~~more scalable~~ Lösung. Mit zunehmender Komplexität Ihres Annotationsprozessors wird die Entwicklung eines eigenständigen Prozessors wie im Beispiel *TableCreator.java* rasch ziemlich kompliziert.

Übungsaufgabe 3: (2) Ergänzen Sie das Beispiel *TableCreationProcessorFactory.java* um weitere SQL-Typen. ■

21.5 Annotationsbasierte Modultests

[54] Ein Modultest besteht aus einem oder mehreren Tests für jede Methode einer Klasse, um deren Bestandteile regelmäßig auf korrektes Verhalten überprüfen zu können. Das beliebteste Hilfsmittel für Modultests bei Java ist JUnit. Zum Zeitpunkt der Drucklegung der amerikanischen Originalausgabe der vierten Auflage dieses Buches, wurde JUnit gerade zu Version 4 aktualisiert, um auch Annotationen einzugliedern.[7] Eines der Hauptprobleme der JUnit-Versionen ohne Unterstützung von Annotationen war der erforderliche Aufwand, um einen Test aufzusetzen und durchzuführen. Obwohl dieser Aufwand im Laufe der Zeit reduziert wurde, gestatten erst Annotationen eine Annäherung an das einfachstmögliche funktionstüchtige Modultestsystem.

[55] Bei den JUnit-Versionen ohne Unterstützung von Annotationen mußten Sie für Ihren Modultest eine separate Klassen anlegen. Mit Annotationen läßt sich der Modultest in die getestete Klasse einfügen und verringert Zeit und Aufwand zum Testen von Modulen auf ein Minimum. Dieser Ansatz hat den zusätzlichen Nutzen, daß sich private Methoden eben so leicht testen lassen wie öffentliche Methoden.

[56] Das Testframework in diesem Abschnitt ist annotationsbasiert und wird daher als „`@Unit`-Framework“ bezeichnet. Die einfachste Anwendungsform (und die Variante, die Sie wahrscheinlich in den meisten Fällen wählen werden) benötigt nur die Annotation `@Test` zur Kennzeichnung der Testmethoden. Ein Ansatz zur Implementierung der Testmethoden besteht darin, argumentlose Methoden mit Rückgabewert vom Typ `boolean` zu schreiben. Die Testmethoden können beliebig benannt werden. Darüber hinaus können Sie die Zugriffsmöglichkeiten der Testmethoden frei wählen, insbesondere können Testmethoden also auch private Methoden aufgerufen.

[57] Die Anwendung des `@Unit`-Frameworks setzt lediglich voraus, daß Sie das Package `net.mindview.atunit` importieren,[8] die Testmethoden und -felder mit dem `@Test`-Tag zu markieren (siehe Beispiele in diesem Abschnitt) und die annotierte Quelltextdatei anschließend mit einem Annotationsprozessor und dem `@Unit`-Framework zu verarbeiten. Ein einfaches Beispiel:

```
//: annotations/AtUnitExample1.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample1 {
    public String methodOne() {
        return ''This is methodOne'';
    }
    public int methodTwo() {
        System.out.println(''This is methodTwo'');
        return 2;
    }
    @Test boolean methodOneTest() {
        return methodOne().equals(''This is methodOne'');
    }
    @Test boolean m2() { return methodTwo() == 2; }
    @Test private boolean m3() { return true; }
    // Shows output for failure:
    @Test boolean failureTest() { return false; }
```

[7] Ich hatte ursprünglich vor, einen verbesserten JUnit-Test auf der Grundlage des in diesem Abschnitt gezeigten Ansatzes vorzustellen. JUnit4 scheint allerdings viele hier präsentierte Ideen zu umfassen, so daß einfacher ist sich that zu fügen.

[8] Die Bibliothek im Package `net.mindview.atunit` gehört zur Quelltextdistribution dieses Buches, die Sie von der Webadresse *http://www.mindview.net* herunterladen können.

```
    @Test boolean anotherDisappointment() { return false; }
    public static void main(String[] args) throws Exception {
        OSExecute.command("java net.mindview.atunit.AtUnit AtUnitExample1");
    }
} /* Output:
      annotations.AtUnitExample1
      . methodOneTest
      . m2 This is methodTwo

      . m3
      . failureTest (failed)
      . anotherDisappointment (failed)
      (5 tests)

      >>> 2 FAILURES <<<
      annotations.AtUnitExample1: failureTest
      annotations.AtUnitExample1: anotherDisappointment
   *///:~
```

Per `@Unit`-Framework getestete Klassen müssen einem Package zugeordnet sein.

[58] Die `@Test`-Annotation vor den Methoden `methodOneTest()`, `m2()`, `m3()`, `failureTest()` und `anotherDisappointment()` kennzeichnet diese Methoden als Testmethoden (allgemeiner „Modultests“). Die Klasse `net.mindview.atunit.AtUnit` des `@Unit`-Frameworks erzwingt außerdem, daß diese fünf Testmethoden keine Argumente haben und entweder `boolean` oder `void` zurückgeben. Sie müssen beim Schreiben einer Testmethode nur festlegen, ob der Test erfolgreich verlaufen oder gescheitert ist, die Methode also `true` oder `false` zurückgibt (für Methoden mit Rückgabewert vom Typ `boolean`).

2[59] Wenn Sie bereits mit JUnit vertraut sind, ist Ihnen sicher die informationshaltige Ausgabe aufgefallen: Der aktuell verarbeitete Test wird angezeigt, so the output from that test is more useful. Gescheiterte Klassen und Tests werden am Schluß angegeben.

[60] Sie sind nicht gezwungen, die Testmethoden in die getestete Klasse einzubetten, wenn diese Vorgehensweise mit Ihren Anforderungen nicht vereinbar ist. Der Ableitungsmechanismus ist die einfachste Möglichkeit, um die Einbettung zu umgehen:

```
//: annotations/AtUnitExternalTest.java
// Creating non-embedded tests.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExternalTest extends AtUnitExample1 {
    @Test boolean _methodOne() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean _methodTwo() { return methodTwo() == 2; }
    public static void main(String[] args) throws Exception {
        OSExecute.command("java net.mindview.atunit.AtUnit AtUnitExternalTest");
    }
} /* Output:
      annotations.AtUnitExternalTest
      . _methodOne
      . _methodTwo This is methodTwo

      OK (2 tests)
   *///:~
```

Dieses Beispiel zeigt nebenbei auch den Wert der flexiblen Wahl von Bezeichern (im Gegensatz dazu müssen bei JUnit alle Tests mit dem Präfix `test` beginnen). Hier erhält eine mit `@Test` annotierten Methode, die eine andere Methode unmittelbar testet, den Namen der getesteten Methode, kombiniert mit einem führenden Unterstrich (diese Notation empfiehlt keineswegs den idealen Stil, sondern eine von mehreren wählbaren Konventionen).

[61] Die Komposition liefert eine andere Möglichkeit, um die Einbettung der Testmethoden zu vermeiden:

```
//: annotations/AtUnitComposition.java
// Creating non-embedded tests.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitComposition {
    AtUnitExample1 testObject = new AtUnitExample1();
    @Test boolean _methodOne() {
        return testObject.methodOne().equals("This is methodOne");
    }
    @Test boolean _methodTwo() {
        return testObject.methodTwo() == 2;
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command("java net.mindview.atunit.AtUnit AtUnitComposition");
    }
} /* Output:
      annotations.AtUnitComposition
      . _methodOne
      . _methodTwo This is methodTwo

      OK (2 tests)
   *///:~
```

Da zu jedem Test ein neues `AtUnitComposition`-Objekt erzeugt wird, wird auch das von dem `testObject` referenziert `AtUnitExample1`-Objekt bei jedem Test neu erzeugt.

[62] Das `@Unit`-Framework enthält keine Zusicherungsmethoden wie JUnit, but the second form of the @Test method gestattet die Rückgabe von `void` (oder `boolean`, wenn Sie auch in diesem Fall noch `true` oder `false` zurückgeben wollen). Die Auswertung von Zusicherungen muß bei Java normalerweise mit dem Schalter `-ea` des `java`-Kommandos auf der Kommandozeile zugeschaltet werden. Das `@Unit`-Framework aktiviert Zusicherungen dagegen automatisch. Sie können sogar eine Ausnahme auswerfen, um das Scheitern eines Tests anzuzeigen. Eines der Designziele des `@Unit`-Frameworks besteht darin, so wenig zusätzliche Syntax wie möglich zu verlangen und Sie brauchen bei Java nicht mehr als Zusicherungen und Ausnahmen, um Fehler anzuzeigen. Eine nicht erfüllte Zusicherung oder eine von einer Testmethode hervorgerufene Ausnahme wird als Scheitern des Tests interpretiert, wobei das `@Unit`-Framework den Test in diesem Fall aber nicht abbricht, sondern fortsetzt, bis alle Teile des Tests verarbeitet sind. Ein Beispiel:

```
//: annotations/AtUnitExample2.java
// Assertions and exceptions can be used in @Tests.
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample2 {
    public String methodOne() {
```

```
        return ''This is methodOne'';
    }
    public int methodTwo() {
        System.out.println(''This is methodTwo'');
        return 2;
    }
    @Test void assertExample() {
        assert methodOne().equals(''This is methodOne'');
    }
    @Test void assertFailureExample() {
        assert 1 == 2: ''What a surprise!'';
    }
    @Test void exceptionExample() throws IOException {
        new FileInputStream(''nofile.txt''); // Throws
    }
    @Test boolean assertAndReturn() {
        // Assertion with message:
        assert methodTwo() == 2: ''methodTwo must equal 2'';
        return methodOne().equals(''This is methodOne'');
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(''java net.mindview.atunit.AtUnit AtUnitExample2'');
    }
} /* Output:
      annotations.AtUnitExample2
      . assertExample
      . assertFailureExample java.lang.AssertionError: \
        What a surprise! (failed)
      . exceptionExample java.io.FileNotFoundException: \
        nofile.txt (The system cannot find the file specified) (failed)
      . assertAndReturn This is methodTwo

      (4 tests)

      >>> 2 FAILURES <<<
      annotations.AtUnitExample2: assertFailureExample
      annotations.AtUnitExample2: exceptionExample
   *///:~
```

Das nächste Beispiel verwendet nicht-eingebettete Testmethoden mit Zusicherungen, um einige Methoden eines `HashSet`-Containers zu testen:

```
//: annotations/HashSetTest.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class HashSetTest {
    HashSet<String> testObject = new HashSet<String>();
    @Test void initialization() {
        assert testObject.isEmpty();
    }
    @Test void _contains() {
        testObject.add(''one'');
        assert testObject.contains(''one'');
    }
    @Test void _remove() {
```

```
        testObject.add(''one'');
        testObject.remove(''one'');
        assert testObject.isEmpty();
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command(''java net.mindview.atunit.AtUnit HashSetTest'');
    }
} /* Output:
      annotations.HashSetTest
      . initialization
      . _remove
      . _contains
      OK (3 tests)
   *///:~
```

Sofern keine weiteren Randbedingungen bestehen, wäre der Ableitungsansatz aus Beispiel *AtUnitExternalTest.java* auf Seite 839 hier einfacher.

Übungsaufgabe 4: (3) Verifizieren Sie bei den beiden Kompositionsbeispielen *AtUnitComposition.java* (Seite 840) und *HashSetTest.java* (Seite 841), daß dem Feld `testObject` bei jedem Test ein neu erzeugtes Objekt zugewiesen wird. ■

Übungsaufgabe 5: (1) Ändern Sie das Beispiel *HashSetTest.java* (Seite 841), so daß es den Ableitungsansatz implementiert. ■

Übungsaufgabe 6: (1) Testen Sie einen `LinkedList`-Container mit dem Ansatz von Beispiel *HashSetTest.java* (Seite 841). ■

Übungsaufgabe 7: (1) Ändern Sie Übungsaufgabe 6, so daß das Programm den Ableitungsansatz verwendet. ■

[63] Das `@Unit`-Framework erzeugt zu jedem Modultest mit Hilfe des Standardkonstruktors ein Objekt der zu testenden Klasse. Der Test wird mit diesem Objekt ausgeführt und das Objekt anschließend verworfen, um Seiteneffekte bei anderen Modultests zu vermeiden. Das `@Unit`-Framework verläßt sich also darauf, daß ein Standardkonstruktor existiert. Hat Ihre Klasse keinen Standardkonstruktor oder ist ein komplizierter Vorgang zur Objekterzeugung erforderlich, so können Sie eine statische Methode zur Objekterzeugung anlegen und mit der Annotation `@TestObjectCreate` kennzeichnen:

```
//: annotations/AtUnitExample3.java
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample3 {
    private int n;
    public AtUnitExample3(int n) { this.n = n; }
    public int getN() { return n; }
    public String methodOne() {
        return ''This is methodOne'';
    }
    public int methodTwo() {
        System.out.println(''This is methodTwo'');
        return 2;
    }
    @TestObjectCreate static AtUnitExample3 create() {
        return new AtUnitExample3(47);
    }
```

```
    @Test boolean initialization() { return n == 47; }
    @Test boolean methodOneTest() {
        return methodOne().equals("This is methodOne");
    }
    @Test boolean m2() { return methodTwo() == 2; }
    public static void main(String[] args) throws Exception {
        OSExecute.command("java net.mindview.atunit.AtUnit AtUnitExample3");
    }
} /* Output:
      annotations.AtUnitExample3
      . initialization
      . methodOneTest
      . m2 This is methodTwo

      OK (3 tests)
   *///:~
```

Das `@Unit`-Framework erzwingt, daß die mit `@TestObjectCreate` annotierte Methode statisch ist und ein Objekt der zu testenden Klasse zurückgibt.

[64] Gelegentlich brauchen Sie zusätzliche Felder zur Unterstützung eines Modultests. Die Annotation `@TestProperty` dient zur Markierung von Feldern, die ausschließlich für Modultests verwendet werden (also entfernt werden können, bevor Sie das Programm an den Kunden übergeben). Das folgende Beispiel trennt eine Zeichenkette mit Hilfe der `String`-Methode `split()` auf und verwendet die Einzelteile, um Testobjekte zu erzeugen:

```
//: annotations/AtUnitExample4.java
package annotations;
import java.util.*;
import net.mindview.atunit.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitExample4 {
    static String theory = "All brontosauruses " +
        "are thin at one end, much MUCH thicker in the " +
        "middle, and then thin again at the far end.";
    private String word;
    private Random rand = new Random(); // Time-based seed
    public AtUnitExample4(String word) { this.word = word; }
    public String getWord() { return word; }
    public String scrambleWord() {
        List<Character> chars = new ArrayList<Character>();
        for(Character c : word.toCharArray())
            chars.add(c);
        Collections.shuffle(chars, rand);
        StringBuilder result = new StringBuilder();
        for(char ch : chars)
            result.append(ch);
        return result.toString();
    }
    @TestProperty static List<String> input = Arrays.asList(theory.split(" "));
    @TestProperty static Iterator<String> words = input.iterator();
    @TestObjectCreate static AtUnitExample4 create() {
        if(words.hasNext())
            return new AtUnitExample4(words.next());
        else
            return null;
```

```
    }
    @Test boolean words() {
        print(''''' + getWord() + '''');
        return getWord().equals(''are'');
    }
    @Test boolean scramble1() {
        // Change to a specific seed to get verifiable results:
        rand = new Random(47);
        print(''''' + getWord() + '''');
        String scrambled = scrambleWord();
        print(scrambled);
        return scrambled.equals(''lAl'');
    }
    @Test boolean scramble2() {
        rand = new Random(74);
        print(''''' + getWord() + '''');
        String scrambled = scrambleWord();
        print(scrambled);
        return scrambled.equals(''tsaeborornussu'');
    }
    public static void main(String[] args) throws Exception {
        System.out.println(''starting'');
        OSExecute.command(''java net.mindview.atunit.AtUnit AtUnitExample4'');
    }
} /* Output:
      starting
      annotations.AtUnitExample4
      . scramble1 'All'
      lAl

      . scramble2 'brontosauruses'
      tsaeborornussu

      . words 'are'

      OK (3 tests)
   *///:~
```

Die `@TestProperty`-Annotation kann auch zur Markierung von Methoden verwendet werden, die im Rahmen eines Tests aufgerufen werden, aber selbst keine Testmethoden sind.

[65] Beachten Sie, daß sich dieses Programm auf die Ausführungsreihenfolge der Tests verläßt. Im allgemeinen ist dies keine gute Vorgehensweise.

[66] Umfaßt die Erzeugung Ihres Testobjektes Initialisierungsschritte, die zu einem späteren Zeitpunkt Aufräumarbeiten erforderlich machen, so können Sie optional eine entsprechende statische Methode anlegen, welche die nach dem Test notwendigen Schritte ausführt und diese Methode mit der Annotation `@TestObjectCleanup` markieren. Im folgenden Beispiel öffnet die mit `@TestObjectCreate` annotierte Methode für jedes Testobjekt eine Datei, die vor der Zerstörung des Testobjektes wieder geschlossen werden muß:

```
//: annotations/AtUnitExample5.java
package annotations;
import java.io.*;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class AtUnitExample5 {
```

```
    private String text;
    public AtUnitExample5(String text) { this.text = text; }
    public String toString() { return text; }
    @TestProperty static PrintWriter output;
    @TestProperty static int counter;
    @TestObjectCreate static AtUnitExample5 create() {
        String id = Integer.toString(counter++);
        try {
            output = new PrintWriter("Test" + id + ".txt");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        return new AtUnitExample5(id);
    }
    @TestObjectCleanup static void
        cleanup(AtUnitExample5 tobj) {
        System.out.println("Running cleanup");
        output.close();
    }
    @Test boolean test1() {
        output.print("test1");
        return true;
    }
    @Test boolean test2() {
        output.print("test2");
        return true;
    }
    @Test boolean test3() {
        output.print("test3");
        return true;
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command("java net.mindview.atunit.AtUnit AtUnitExample5");
    }
} /* Output:
      annotations.AtUnitExample5
      . test1
      Running cleanup
      . test2
      Running cleanup
      . test3
      Running cleanup
      OK (3 tests)
   *///:~
```

Die Ausgabe zeigt, daß die Aufräummethode `cleanup()` nach jedem Test automatisch aufgerufen wird.

21.5.1 Modultests mittels @Unit-Framework bei generischen Typen

[67] Generische Typen stellen ein besonderes Problem dar, da Sie nicht „generisch testen“ können. Sie müssen bezüglich eines spezifischen Parametertyps oder eine Auswahl von Parametertypen testen. Die Lösung ist einfach: Sie leiten eine Testklasse von einer Version der generischen Klasse mit fest gewähltem Parametertyp ab.

[68] Das folgende Beispiel zeigt eine einfache Stapelspeicherklasse:

```
//: annotations/StackL.java
// A stack built on a linkedList.
package annotations;
import java.util.*;

public class StackL<T> {
    private LinkedList<T> list = new LinkedList<T>();
    public void push(T v) { list.addFirst(v); }
    public T top() { return list.getFirst(); }
    public T pop() { return list.removeFirst(); }
} ///:~
```

Sie leiten nun eine Testklasse von `StackL<String>` ab, um eine `String`-Version von `StackL` zu testen:

```
//: annotations/StackLStringTest.java
// Applying @Unit to generics.
package annotations;
import net.mindview.atunit.*;
import net.mindview.util.*;

public class StackLStringTest extends StackL<String> {
    @Test void _push() {
        push("one");
        assert top().equals("one");
        push("two");
        assert top().equals("two");
    }
    @Test void _pop() {
        push("one");
        push("two");
        assert pop().equals("two");
        assert pop().equals("one");
    }
    @Test void _top() {
        push("A");
        push("B");
        assert top().equals("B");
        assert top().equals("B");
    }
    public static void main(String[] args) throws Exception {
        OSExecute.command("java net.mindview.atunit.AtUnit StackLStringTest");
    }
} /* Output:
      annotations.StackLStringTest
      . _push
      . _pop
      . _top
      OK (3 tests)
   *////:~
```

Der einzige potentielle Nachteil bei der Ableitung besteht darin, daß Sie die Zugriffsmöglichkeit auf private Methoden der getesteten Klasse verlieren. Ist diese Einschränkung ein Problem, so können Sie die fragliche Methode entweder als `protected` umdeklarieren oder eine mit `@TestProperty` annotierte nicht-private Methode anlegen, welche die private Methode aufruft (eine mit `@TestProperty` annotierte Methode kann mit Hilfe des Werkzeuges `AtUnitRemover` aus dem dem Quelltext entfernt werden, siehe Unterabschnitt 21.5.4).

Übungsaufgabe 8: (2) Schreiben Sie eine Klasse mit einer privaten Methode und einer nicht-

privaten mit `@TestProperty` annotierten Methode, wie oben beschrieben. Rufen Sie die annotierte Methode im Rahmen eines Tests auf. ■

Übungsaufgabe 9: (2) Schreiben Sie mit Hilfe des `@Unit`-Frameworks grundlegende Tests für den Containertyp `HashMap`. ■

Übungsaufgabe 10: (2) Wählen Sie ein beliebiges Beispiel aus dem Buch und erweitern Sie es um Modultests mit Hilfe des `@Unit`-Frameworks. ■

21.5.2 Das @Unit-Framework verlangt keine Testsuiten

[69] Einer der großen Vorteile des `@Unit`-Frameworks im Vergleich mit JUnit besteht darin, daß es keine „Testsuiten“ benötigt. Bei JUnit müssen Sie dem Werkzeug für Modultests mitteilen, was Sie testen wollen. Diese Eigenschaft bedingt die Einführung von „Testsuiten“, um Tests zu gruppieren, damit JUnit sie finden und ausführen kann.

[70] Das `@Unit`-Framework durchsucht Klassendateien nach entsprechenden Annotationen und ruft anschließend die mit `@Test` annotierten Methoden auf. Mein Ziel bei der Arbeit am `@Unit`-Framework bestand größtenteils darin, das Framework so transparent wie möglich zu machen, so daß die Benutzer mit der Verwendung beginnen können, in dem sie einfach Methoden mit `@Test` annotieren. Spezielle Anweisungen oder Kenntnisse, wie bei JUnit oder vielen anderen Frameworks für Modultests notwendig, sind beim `@Unit`-Framework erforderlich. Es ist schwierig genug, Tests zu entwickeln, ohne neue Hürden aufzubauen. Das `@Unit`-Framework macht diese Aufgabe trivial. Es ist daher wahrscheinlicher, daß Sie die Tests auch tatsächlich schreiben.

21.5.3 Implementierung des @Unit-Frameworks

[71] Zunächst müssen die Annotationen definiert werden, alle samt einfache Tags ohne Elemente. Die `@Test`-Annotation wurde in Unterabschnitt 21.1.1 zu Beginn dieses Kapitels definiert. Es folgen die Definitionen der übrigen Annotationen:

```
//: net/mindview/atunit/TestObjectCreate.java
  // The @Unit @TestObjectCreate tag.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCreate {} ///:~

//: net/mindview/atunit/TestObjectCleanup.java
// The @Unit @TestObjectCleanup tag.
package net.mindview.atunit;
import java.lang.annotation.*;

@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface TestObjectCleanup {} ///:~

//: net/mindview/atunit/TestProperty.java
// The @Unit @TestProperty tag.
package net.mindview.atunit;
import java.lang.annotation.*;

// Both fields and methods may be tagged as properties:
@Target({ElementType.FIELD, ElementType.METHOD})
```

```
@Retention(RetentionPolicy.RUNTIME)
public @interface TestProperty {} ///:~
```

Alle Annotation des `@Unit`-Frameworks bleiben bis zur Laufzeit erhalten (`RetentionPolicy.RUNTIME`), da das Framework die Testmethoden im Bytecode erkennen muß.

[72] Die Annotationen werden mit Hilfe des Reflexionsmechanismus' ausgewertet, um das `@Unit`-Framework zu implementieren, das die Tests ausführt. Das Programm verwendet diese Information zur Entscheidung, wie die Testobjekte erzeugt und die Tests auf diesen Objekten ausgeführt werden. Durch die Annotationen ist die Lösung überraschend kurz und einfach:

```
//: net/mindview/atunit/AtUnit.java
// An annotation-based unit-test framework.
// {RunByHand}
package net.mindview.atunit;
import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnit implements ProcessFiles.Strategy {
    static Class<?> testClass;
    static List<String> failedTests = new ArrayList<String>();
    static long testsRun = 0;
    static long failures = 0;
    public static void main(String[] args) throws Exception {
        ClassLoader.getSystemClassLoader()
            .setDefaultAssertionStatus(true); // Enable asserts
        new ProcessFiles(new AtUnit(), ''class'').start(args);
        if(failures == 0)
            print(''OK ('' + testsRun + '' tests)'');
        else {
            print(''('' + testsRun + '' tests)'');
            print(''\n>>> '' + failures + '' FAILURE'' +
                  (failures > 1 ? ''S'' : '''') + '' <<<'');
            for(String failed : failedTests)
                print('' '' + failed);
        }
    }
    public void process(File cFile) {
        try {
            String cName = ClassNameFinder.thisClass(BinaryFile.read(cFile));
            if(!cName.contains(''.''))
                return; // Ignore unpackaged classes
            testClass = Class.forName(cName);
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        TestMethods testMethods = new TestMethods();
        Method creator = null;
        Method cleanup = null;
        for(Method m : testClass.getDeclaredMethods()) {
            testMethods.addIfTestMethod(m);
            if(creator == null)
                creator = checkForCreatorMethod(m);
            if(cleanup == null)
                cleanup = checkForCleanupMethod(m);
```

```
        }
        if(testMethods.size() > 0) {
            if(creator == null)
                try {
                    if(!Modifier.isPublic(
                        testClass.getDeclaredConstructor().getModifiers())) {
                        print("Error: " + testClass +
                              " default constructor must be public");
                        System.exit(1);
                    }
                } catch(NoSuchMethodException e) {
                    // Synthesized default constructor; OK
                }
            print(testClass.getName());
        }
        for(Method m : testMethods) {
            printnb(" . " + m.getName() + " ");
            try {
                Object testObject = createTestObject(creator);
                boolean success = false;
                try {
                    if(m.getReturnType().equals(boolean.class))
                        success = (Boolean)m.invoke(testObject);
                    else {
                        m.invoke(testObject);
                        success = true; // If no assert fails
                    }
                } catch(InvocationTargetException e) {
                    // Actual exception is inside e:
                    print(e.getCause());
                }
                print(success ? "" : "(failed)");
                testsRun++;
                if(!success) {
                    failures++;
                    failedTests.add(testClass.getName() + ": " + m.getName());
                }
                if(cleanup != null)
                    cleanup.invoke(testObject, testObject);
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    static class TestMethods extends ArrayList<Method> {
        void addIfTestMethod(Method m) {
            if(m.getAnnotation(Test.class) == null)
                return;
            if(!(m.getReturnType().equals(boolean.class) ||
                 m.getReturnType().equals(void.class)))
                throw new RuntimeException(
                    "@Test method must return boolean or void");
            m.setAccessible(true); // In case it's private, etc.
            add(m);
        }
    }
    private static Method checkForCreatorMethod(Method m) {
```

```
        if(m.getAnnotation(TestObjectCreate.class) == null)
            return null;
        if(!m.getReturnType().equals(testClass))
            throw new RuntimeException(
                ''@TestObjectCreate must return instance of Class to be tested'');
        if((m.getModifiers() & java.lang.reflect.Modifier.STATIC) < 1)
            throw new RuntimeException(''@TestObjectCreate must be static.'');
        m.setAccessible(true);
        return m;
    }
    private static Method checkForCleanupMethod(Method m) {
        if(m.getAnnotation(TestObjectCleanup.class) == null)
            return null;
        if(!m.getReturnType().equals(void.class))
            throw new RuntimeException(''@TestObjectCleanup must return void'');
        if((m.getModifiers() & java.lang.reflect.Modifier.STATIC) < 1)
            throw new RuntimeException(''@TestObjectCleanup must be static.'');
        if(m.getParameterTypes().length == 0 ||
            m.getParameterTypes()[0] != testClass)
            throw new RuntimeException(
                ''@TestObjectCleanup must take an argument of the tested type.'');
        m.setAccessible(true);
        return m;
    }
    private static Object createTestObject(Method creator) {
        if(creator != null) {
            try {
                return creator.invoke(testClass);
            } catch(Exception e) {
                throw new RuntimeException(
                    ''Couldn't run @TestObject (creator) method.'');
            }
        } else { // Use the default constructor:
            try {
                return testClass.newInstance();
            } catch(Exception e) {
                throw new RuntimeException(
                    ''Couldn't create a test object.''
                    + ''Try using a @TestObject method.'');
            }
        }
    }
} ///:~
```

Das Programm *AtUnit.java* verwendet die Hilfsklasse `net.mindview.util.ProcessFiles`. Die Klasse `AtUnit` implementiert das Interface *`ProcessFiles.Strategy`*, welches die Methode `process()` deklariert. Somit kann dem Konstruktor von `ProcessFiles` ein `AtUnit`-Objekt übergeben werden. Das zweite Argument des Konstruktors weist `ProcessFiles` an, Dateien mit der Endung *.class* zu suchen.

[73] Wenn Sie das Programm ohne Kommandozeilenargumente aufrufen, traversiert es den Teilbaum unterhalb des aktuellen Verzeichnisses. Das Programm akzeptiert auch mehrere Argumente, die entweder Klassendateien (mit oder ohne *.class* Endung) oder Verzeichnisse sind. Da das `@Unit`-Framework die Testklassen und -methoden automatisch findet, ist kein Mechanismus für „Testsuiten“

erforderlich.[9]

[74] Das Programm *AtUnit.java* muß bei Klassendateien den qualifizierten Klassennamen (mit Packagezugehörigkeit) ermitteln, der sich nicht aus dem alleinigen Klassennamen ergibt. Das Abfragen dieser Informationen erfordert die Untersuchung der Klassendatei, eine zwar nicht triviale, aber auch nicht unmögliche Aufgabe.[10] Jede Klassendatei wird nach ihrer Entdeckung geöffnet, ihr binärer Inhalt eingelesen und der statischen `ClassNameFinder`-Methode `thisClass()` übergeben. Mit der Untersuchung des Inhaltes von Klassendateien, betreten wir das Gebiet des „Bytecode-Engineering“:

```
//: net/mindview/atunit/ClassNameFinder.java
package net.mindview.atunit;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class ClassNameFinder {
    public static String thisClass(byte[] classBytes) {
        Map<Integer,Integer> offsetTable = new HashMap<Integer,Integer>();
        Map<Integer,String> classNameTable = new HashMap<Integer,String>();
        try {
            DataInputStream data =
                new DataInputStream(new ByteArrayInputStream(classBytes));
            int magic = data.readInt(); // 0xcafebabe
            int minorVersion = data.readShort();
            int majorVersion = data.readShort();
            int constant_pool_count = data.readShort();
            int[] constant_pool = new int[constant_pool_count];
            for(int i = 1; i < constant_pool_count; i++) {
                int tag = data.read();
                int tableSize;
                switch(tag) {
                    case 1: // UTF
                        int length = data.readShort();
                        char[] bytes = new char[length];
                        for(int k = 0; k < bytes.length; k++)
                            bytes[k] = (char)data.read();
                        String className = new String(bytes);
                        classNameTable.put(i, className);
                        break;
                    case 5: // LONG
                    case 6: // DOUBLE
                        data.readLong(); // discard 8 bytes
                        i++; // Special skip necessary
                        break;
                    case 7: // CLASS
                        int offset = data.readShort();
                        offsetTable.put(i, offset);
                        break;
                    case 8: // STRING
                        data.readShort(); // discard 2 bytes
                        break;
                    case 3: // INTEGER
```

[9] Es ist nicht klar, warum der Standardkonstruktor der getesteten Klasse öffentlich sein muß. Der Aufruf der `newInstance()`-Methode „bleibt hängen“, wenn dieser Konstruktor nicht als `public` definiert ist.

[10] Jeremy Meyer und ich haben fast einen ganzen Tag investiert, um dies herauszufinden.

```
                    case 4: // FLOAT
                    case 9: // FIELD_REF
                    case 10: // METHOD_REF
                    case 11: // INTERFACE_METHOD_REF
                    case 12: // NAME_AND_TYPE
                        data.readInt(); // discard 4 bytes;
                        break;
                    default:
                        throw new RuntimeException("Bad tag " + tag);
                }
            }
            short access_flags = data.readShort();
            int this_class = data.readShort();
            int super_class = data.readShort();
            return
                classNameTable.get(offsetTable.get(this_class)).replace('/', '.');
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
    // Demonstration:
    public static void main(String[] args) throws Exception {
        if(args.length > 0) {
            for(String arg : args)
                print(thisClass(BinaryFile.read(new File(arg))));
        } else
            // Walk the entire tree:
            for(File klass : Directory.walk(".", ".*\\.class"))
                print(thisClass(BinaryFile.read(klass)));
    }
} ///:~
```

[75] An dieser Stelle können nicht alle Einzelheiten dargelegt werden. Jede Klassendatei hat aber ein festes Format und ich habe versucht, aussagekräftige Feldbezeichner für die einzelnen Datenstücke zu verwenden, die aus dem Eingabedatenstrom gelesen werden. Insbesondere können Sie die Größe jedes Datenstücks an der Länge der `read()`-Anweisung aus dem Eingabedatenstrom ablesen. Die ersten 32 Bit einer Klassendatei beschreiben stets die sogenannte „magische Zahl" (*magic number*) `0xcafebabe`,[11] die beiden folgenden `short`-Werte geben die Version des Formates der Klassendatei an. Der Konstantenpool enthält die Konstanten des Programms (darunter Zeichenketten, Namen von Klassen, Interfaces und Feldern) und hat somit keine feste Größe. Der nächste `short`-Wert gibt die Größe des Konstantenpools an, so daß ein Array passender Länge allokiert werden kann. Jedes Element des Konstantenpools kann einen Wert fester oder variabler Größe beinhalten, so daß wir den vor jedem Wert angegebenen Typ abfragen und in der `switch`-Anweisung auswerten müssen. Die Hilfsklasse `ClassNameFinder` unternimmt keine genaue Untersuchung sämtlicher Informationen in der Klassendatei, sondern beschränkt sich auf die für uns interessanten Daten und verwirft einen beträchtlichen Anteil des Dateiinhaltes. Informationen über Klassen werden in den von `classNameTable` und `offsetTable` referenzierten `HashMap`-Containern gespeichert. Nach dem Einlesen des Konstantenpools kann die Information `this_class` abgefragt werden. Es handelt sich dabei um einen Schlüssel in `offsetTable`, welcher einen Schlüssel in `classNameTable` liefert, der letztendlich zum Klassennamen führt.

[76] Zurück zu *AtUnit.java* auf Seite 848: Die `process()`-Methode hat nun den Klassennamen zur Verfügung und kann prüfen, ob dieser einen Punkt enthält, die Klasse also einem Package zugeordnet

[11] Verschiedende Legenden ranken sich um die Bedeutung dieser Zahl. Da Java aber von Nerds entwickelt wurde, ist die Vermutung begründet, daß Phantasien über eine Frau in einem Coffeeshop dabei eine Rolle spielen.

ist. Klassen, die keinem Package angehören, werden ignoriert. Packagegebundene Klassen werden beim Aufruf der statischen `Class`-Methode `forName()` über den Standardklassenlader geladen. Nun kann die Klasse auf die im `@Unit`-Framework definierten Annotationen hin untersucht werden.

[77] Lediglich drei Fälle sind zu beachten: Mit `@Test` annotierte Methoden (Testmethoden) werden in einem Objekt der von `ArrayList<Method>` abgeleiteten Klasse `TestMethods` gespeichert. Mit `@TestObjectCreate` oder `@TestObjectCleanup` annotierte Methoden are discovered through the associated method calls that you see in the code, which look for the annotations.

[78] Sind mit `@Test` annotierte Methoden vorhanden, so wird der Klassename ausgegeben, damit der Beobachter das Geschehen verfolgen kann und anschließend der Test ausgeführt. Dabei wird der Methodenname angezeigt und die `createTestObject()`-Methode aufgerufen, welche die mit `@TestObjectCreate` annotierte Methode aufruft (falls vorhanden) oder auf den Standardkonstruktor zurückgreift. Nach dem Erzeugen des Testobjektes wird der Test bezüglich dieses Objektes aufgerufen. Gibt der Test ein Ergebnis vom Typ `boolean` zurück, so wird dieser Wert erfaßt. Andernfalls wird die Abwesenheit von Ausnahmen als Erfolg interpretiert (Ausnahmen treten bei einer nicht erfüllten Zusicherung oder als Ausnahmen auf.) Ruft ein Test eine Ausnahme hervor, so wird die Ausnahmeinformation ausgegeben, um die Ursache anzugeben. Gescheiterte Tests bewirken, daß ein entsprechender Zähler inkrementiert und Klassen- sowie Methodenname in den von `failedTests` referenzierten `ArrayList<String>`-Container eingetragen werden, so daß sie vor dem Programmende berichtet werden können.

Übungsaufgabe 11: (5) Ergänzen Sie das `@Unit`-Framework um eine `@TestNote`-Annotation, deren Text einfach im Rahmen des Tests ausgegeben wird. ■

21.5.4 Entfernen von Testmethoden und -feldern

[79] Bei vielen Projekten macht es nichts aus, ob Sie Testklassen und -methoden in der ausgelieferten Version stehen lassen oder nicht (vor allem, wenn Sie alle Testmethoden als private Methoden anlegen). In einigen Fällen wollen Sie die Testanweisungen aber entfernen, um das Volumen des Produktes zu verringern oder um die Anweisungen vor dem Kunden zu verbergen.

[80] Dazu ist ein anspruchsvolleres Bytecode-Engineering notwendig, als die manuelle Lösung im Beispiel *ClassNameFinder.java* auf Seite 851f. Die quelloffene Javassist-Bibliothek[12] rückt Bytecode-Engineering aber in den Bereich des Möglichen. Das folgende Programm akzeptiert einen optionalen `-r`-Schalter als erstes Argument. Wenn Sie das Programm mit diesem Schalter aufrufen, entfernt es die `@Test`-Annotationen der verarbeiteten Klassendateien. Ohne Schalter zeigt es lediglich die Vorkommen dieser Annotationen an. Wiederum dient die Hilfsklasse `ProcessFiles` zum Traversieren der Dateien und Verzeichnisse Ihrer Wahl:

```
//: net/mindview/atunit/AtUnitRemover.java
// Displays @Unit annotations in compiled class files. If
// first argument is ''-r'', @Unit annotations are removed.
// {Args: ..}
// {Requires: javassist.bytecode.ClassFile;
// You must install the Javassist library from
// http://sourceforge.net/projects/jboss/ }
package net.mindview.atunit;
import javassist.*;
import javassist.expr.*;
import javassist.bytecode.*;
```

[12] Vielen Dank an Dr. Shigeru Chiba für die Entwicklung dieser Bibliothek und seine Unterstützung bei der Arbeit an *AtUnitRemover.java*.

```
import javassist.bytecode.annotation.*;
import java.io.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class AtUnitRemover implements ProcessFiles.Strategy {
    private static boolean remove = false;
    public static void main(String[] args) throws Exception {
        if(args.length > 0 && args[0].equals("-r")) {
            remove = true;
            String[] nargs = new String[args.length - 1];
            System.arraycopy(args, 1, nargs, 0, nargs.length);
            args = nargs;
        }
        new ProcessFiles(new AtUnitRemover(), "class").start(args);
    }
    public void process(File cFile) {
        boolean modified = false;
        try {
            String cName = ClassNameFinder.thisClass(BinaryFile.read(cFile));
            if(!cName.contains("."))
                return; // Ignore unpackaged classes
            ClassPool cPool = ClassPool.getDefault();
            CtClass ctClass = cPool.get(cName);
            for(CtMethod method : ctClass.getDeclaredMethods()) {
                MethodInfo mi = method.getMethodInfo();
                AnnotationsAttribute attr = (AnnotationsAttribute)
                    mi.getAttribute(AnnotationsAttribute.visibleTag);
                if(attr == null) continue;
                for(Annotation ann : attr.getAnnotations()) {
                    if(ann.getTypeName().startsWith("net.mindview.atunit")) {
                        print(ctClass.getName() + " Method: "
                              + mi.getName() + " " + ann);
                        if(remove) {
                            ctClass.removeMethod(method);
                            modified = true;
                        }
                    }
                }
            }
            // Fields are not removed in this version (see text).
            if(modified)
                ctClass.toBytecode(
                    new DataOutputStream(new FileOutputStream(cFile)));
            ctClass.detach();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
    }
} ///:~
```

Die Klasse `ClassPool` repräsentiert ein Abbild aller von Ihnen modifizierten Klassen des Systems und garantiert die Konsistenz aller modifizierten Klassen. Sie müssen jedes `CtClass`-Objekt bei diesem `ClassPool`-Objekt anfordern, in analoger Weise, wie der Klassenlader und die statische `Class`-Methode `forName()` Klassen in die Laufzeitumgebung laden.

[81–82] Die Klasse `CtClass` enthält den Bytecode eines Klassenobjektes und gestattet Ihnen, Infor-

mationen über diese Klasse abzufragen und den Bytecode dieser Klasse zu manipulieren. Wir rufen die `getDeclaredMethods()`-Methode (ähnlich wie beim Reflexionsmechanismus) auf, um über ein `CtMethod`-Objekt ein `MethodInfo`-Objekt anzufordern. Das letztere Objekt gestattet den Zugriff auf die Annotationen der Methode. Hat eine Methode eine Annotation aus dem Package `net.mindview.atunit`, so wird diese Annotation entfernt. Wurde die Klassendatei modifiziert, so wird die ursprüngliche Klassendatei mit der neuen Version überschrieben.

[83] Zum Zeitpunkt der Drucklegung der amerikanischen Originalausgabe der vierten Auflage dieses Buches, war die Javassist-Bibliothek gerade um die Löschfunktionalität ergänzt worden[13] und wir entdeckten, daß das Entfernen von Feldern mit der Annotation `@TestProperty` schwieriger war als das Entfernen von Methoden. Felder können nicht einfach gelöscht werden, da sich statische Initialisierungsoperationen eventuell auf diese Felder beziehen. Somit entfernt die obige Version nur Methoden im Rahmen des `@Unit`-Frameworks. Lesen Sie hin und wieder auf der Javassist-Webseite nach, ob es Aktualisierungen gibt. Das Entfernen von Feldern sollte letztendlich möglich sein. Beachten Sie in der Zwischenzeit, daß Sie durch die externen Testmethoden im Beispiel *AtUnitExternalTest.java* auf Seite 839 alle Tests einfach dadurch entfernen können, daß Sie die abgeleitete Testklasse löschen.

21.6 Zusammenfassung

[84] Annotationen sind eine willkommene Erweiterung für Java, ein strukturiertes typgeprüftes Hilfsmittel, um Metadaten in den Quelltext einzusetzen, ohne ihn unleserlich oder unübersichtlich zu machen. Annotationen können verwendet werden, um das lästige Pflegen von Deployment-Deskriptoren und anderen generierten Dateien zu vermeiden. Die Tatsache, daß das Javadoc-Tag `@deprecated` durch die Annotation `@Deprecated` verdrängt wurde, ist ein beispielhaftes Zeichen dafür, um wieviel sich Annotationen besser als Kommentare eignen, um Informationen über eine Klasse zu beschreiben.

[85] Die SE 5 definiert nur eine Handvoll von Annotationen. Wenn Sie keine Bibliothek finden, müssen Sie also Ihre benötigten Annotationen und deren Verarbeitung selbst definieren. Das Hilfsprogramm `apt` vereinfacht das Übersetzen von Quelltextdateien, indem es diese nach Verarbeitung der Annotation übersetzen kann, wobei die Programmierschnittstelle im Package `com.sun.mirror` gegenwärtig nur wenig mehr enthält, als etwas grundlegende Funktionalität, die Sie beim Identifizieren der Komponenten einer Java-Klasse unterstützt. Bytecode-Engineering ist einerseits mit Hilfe der Javassist-Bibliothek möglich, andererseits können Sie, in einfachen Fällen, eine eigene Klasse zur Manipulation von Bytecode schreiben.

[86] Die Situation wird sich sicher verbessern. Die Anbieter von Programmierschnittstellen und Frameworks werden Annotationen als Bestandteile ihrer Toolkits liefern. Wie Sie sich im Hinblick auf das `@Unit`-Framework vorstellen können, werden Annotationen unsere Programmiererfahrung mit Java sehr wahrscheinlich deutlich verändern.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

[13] Dr. Shigeru Chiba hat die `CtClass`-Methode `removeMethod()` auf unsere Bitte hin ergänzt.

Kapitel 22

Threadprogrammierung

Inhaltsübersicht

[0] Bis jetzt haben Sie die sequentielle Programmierung kennengelernt. Ein sequentielles Programm wird Schritt für Schritt verarbeitet.

[1] Viele Probleme können mittels sequentieller Programmierung gelöst werden. Bei manchen Problemen ist des dagegen bequem, wenn nicht sogar wesentlich, Programmteile parallel zu verarbeiten, so daß sie entweder scheinbar oder bei Vorhandensein mehrerer Prozessoren tatsächlich gleichzeitig verarbeitet werden.

[2] Durch parallele Programmierung (Threadprogrammierung) kann die Ausführungsgeschwindigkeit von Programmen erheblich verbessert oder/und das Designmodell bestimmter Programmtypen vereinfacht werden. Ein erfahrener Entwickler auf dem Gebiet der Threadprogrammierung zu werden, geht einen Schritt über die Dinge hinaus, die Sie bis zu dieser Stellen in diesem Buch gelernt haben und ist eine Aufgabe von mittlerem bis fortgeschrittenem Schwierigkeitsgrad. Dieses Kapitel kann nicht mehr als eine Einführung sein und Sie sollten sich auch nach sorgfältigem Durcharbeiten dieses Kapitels keinesfalls als versierter Entwickler auf dem Gebiet der Threadprogrammierung betrachten.

[3] Wie Sie lernen werden, treten die wirklichen Probleme in der Threadprogrammierung dann auf, wenn zeitgleich verarbeitete Aufgaben beginnen, sich gegenseitig störend zu beinflussen. Dies kann auf derart subtile Weise und so selten geschehen, daß die Charakterisierung wohl gerechtfertigt ist, ein paralleles Programm solle sich zwar deterministisch verhalten, verhalte sich in der Realität aber gerade entgegengesetzt. That is, you can make an argument to conclude that es möglich

ist, durch Achtsamkeit und sorgfältiges Studium des Quelltextes ein threadbasiertes Programm zu schreiben, welches korrekt funktioniert. Es ist in der Praxis aber viel leichter, ein solches Programm zu schreiben, das nur scheinbar funktioniert und unter den „richtigen“ Bedingungen scheitert. Diese Voraussetzungen treten unter Umständen nie oder nur so selten ein, daß Sie sie beim Testen nicht bemerken. Es ist sogar möglich, daß Sie keinen Test implementieren können, der eine bestimmte Fehlersituation für Ihr threadbasiertes Programm herbeiführt. Die resultierenden Probleme treten häufig nur gelegentlich auf und äußern Sie in Form von Kundenbeschwerden. Die Threadprogrammierung aber nicht zu beachten rächt sich eines Tages. Dies ist eines der besten Argumente, um sich mit dem Gebiet der Threadprogrammierung auseinanderzusetzen.

[4] Das Gebiet der Threadprogrammierung scheint also voller Tücken zu sein und es ist mit Sicherheit angebracht, dabei etwas Respekt zu empfinden. Obwohl Sun Microsystems mit Version 5 der Java Standard Edition (SE 5) die Möglichkeiten zur Threadprogrammierung erheblich verbessert hat, gibt es noch immer kein Sicherungsnetz, beispielsweise Prüfungen zur Übersetzungszeit oder geprüfte Ausnahmen, um Sie auf Fehltritte hinzuweisen. Auf dem Gebiet der Threadprogrammierung sind Sie auf sich alleine gestellt und es wird Ihnen nur durch Argwohn gepaart mit Draufgängertum gelingen, funktionstüchtige threadbasierte Java-Programme zu schreiben.

[5] Einige Kollegen sind der Ansicht, Threadprogrammierung sei ein zu weit fortgeschrittenes Thema für ein Buch, das in eine Programmiersprache einführt. Sie argumentieren, daß Threadprogrammierung ein separates Gebiet sei und unabhängig dargestellt werden könne, während die wenigen Vorkommen in der alltäglichen Programmierarbeit (etwa bei graphischen Benutzeroberflächen) mit Hilfe entsprechender Umschreibungen der Fachbegriffe diskutiert werden könnten. Warum ein so kompliziertes Thema einführen, wenn es sich verhindern läßt?

[6] Ach! Wenn es doch nur so einfach wäre. Sie haben leider keine Wahl, wann Threads in Ihren Programmen in Erscheinung treten. Daß Sie noch keinen Thread selbst gestartet haben bedeutet nicht, daß Sie es auf Dauer vermeiden können, in Ihren Programmen Threads zu benutzen. Beispielsweise gehören Webapplikationen zu den häufigsten Java-Anwendungen und Servlets unterliegen naturgemäß dem Zugriff durch mehr als einen Thread. Diese Eigenschaft ist wesentlich, da Webserver häufig mehrere Prozessoren haben und parallele Threadprogrammierung eine ideale Möglichkeit ist, um diese Prozessoren zu nutzen. Unabhängig davon, wie einfach die Funktionalität eines Servlets ist, müssen Sie die Funktionsweise threadbasierter Programme verstehen, um Servlets bestimmungsgemäß einsetzen zu können. Dasselbe gilt für die Entwicklung graphischer Benutzeroberflächen (siehe Kapitel 23). Die Swing- und die SWT-Bibliothek verfügen zwar beide über Mechanismus für Threadsicherheit, aber Sie müssen sie verstehen, um sie richtig verwenden zu können.

[7] Java ist eine multithreadfähige Sprache und die Belange (*issues*) der Threadprogrammierung sind unabhängig davon vorhanden, ob Sie sich ihrer bewußt sind oder nicht. Infolge dessen gibt es viele Java-Programme, die entweder nur zufällig funktionieren oder zumeist korrekt arbeiten, aber hin und wieder aufgrund nicht entdeckter Programmierfehler auf geheimnisvolle Weise ausfallen. Ein solcher Ausfall kann gutartig verlaufen, es können aber auch wertvolle Daten verloren gehen und Sie könnten die Ursache außerhalb Ihrer Software vermuten, wenn Sie kein Bewußtsein für die Probleme der Threadprogrammierung haben. Solche Fehler werden unter Umständen auch erst beim Betrieb des Programmes auf einem Mehrprozessorsystem sichtbar oder treten dort verstärkt auf. Kurz, Verständnis für Threadprogrammierung schärft Ihre analytischen Fähigkeiten, wenn sich ein scheinbar fehlerfreies Programm merkwürdig verhält.

[8] Die Auseinandersetzung mit der Threadprogrammierung ist wie der Eintritt in eine neue Welt und das Erlernen einer neuen Sprache oder wenigstens einer Auswahl von neuen Sprachkonzepten. Die Threadprogrammierung zu verstehen ist etwa so kompliziert, wie die Objektorientierte Programmierung zu verstehen. Mit etwas Lerneifer können Sie die Grundzüge ausloten. Echtes Durchdringen

setzt aber in der Regel ein vertieftes Studium des Gebietes voraus. Das Ziel dieses Kapitels besteht darin, Ihnen die Grundlagen der Threadprogrammierung zu vermitteln, damit Sie die Konzepte verstehen und vernünftige threadbasierte Programme schreiben können. Hüten Sie sich aber vor Vermessenheit. Wenn Sie ein komplexeres threadbasiertes Programm schreiben müssen, brauchen Sie entsprechende Literatur.

22.1 Die Vielseitigkeit der Threadprogrammierung

[9] Einer der primären Gründe, warum die Threadprogrammierung so unübersichtlich wirkt besteht darin, that there is more than one problem to solve using concurrency, and more than one approach to implementing concurrency, and no clean mapping between the two issues (and often a blurring of the lines all around). Sie sind daher gezwungen, alle Belange und Spezialfälle zu verstehen, um effektiv mit der Threadprogrammierung umgehen zu können.

[10] Die Probleme, die Sie mittels Threadprogrammierung lösen, lassen Sie grob nach „Geschwindigkeit“ und „Handlichkeit des Designs“ klassifizieren.

22.1.1 Erhöhte Verarbeitungsgeschwindigkeit

[11] Der Geschwindigkeitsaspekt wirkt auf den ersten Blick einleuchtend: Wenn Sie die Verarbeitung eines Programms beschleunigen möchten, teilen Sie es in mehrere Teile auf und verarbeiten jeden mit Hilfe eines separaten Prozessors. Threadprogrammierung ist ein fundamentaler Anwendungsfall für die Programmierung mit Ziel Mehrprozessorbetrieb. Das Mooresche Gesetz scheint seine Gültigkeit zu verlieren (zumindest für herkömmliche Chips). Steigerungen der Verarbeitungsgeschwindigkeit ergeben sich heute aus Mehrkernprozessoren, statt aus schnelleren Chips. Wenn Sie die Verarbeitung Ihrer Programme beschleunigen wollen, müssen Sie den Vorteil durch mehr als einen Prozessor zu nutzen lernen, einen Gewinn den Sie aus der Threadprogrammierung ziehen können.

[12] Wenn Sie einen Rechner mit mehr als einem Prozessor zur Verfügung haben, können Sie mehrere Aufgaben an diese Prozessoren verteilen und dadurch unter Umständen den Durchsatz drastisch erhöhen. Dieser Effekt tritt häufig bei leistungsfähigen Webservern mit mehreren Prozessoren ein, die viele Benutzeranfragen an die Prozessoren verteilen, wobei pro Anfrage ein Thread allokiert wird.

[13] Es ist allerdings auch möglich, die Verarbeitungsgeschwindigkeit eines Programms, welches von nur einem Prozessor verarbeitet wird, mittels Threadprogrammierung zu verbessern.

[14] Das ist auf den ersten Blick nicht einleuchtend. Verglichen mit der sequentiellen Verarbeitung aller Programmteile, müßte ein threadbasiertes Programm im Einprozessorbetrieb durch das erforderliche Umschalten zwischen den einzelnen Aufgaben (*context switch*) doch eigentlich zusätzliche Unkosten verursachen. Oberflächlich betrachtet wäre es günstiger, alle Programmteile in Form einer einzigen Aufgabe zu verarbeiten und somit die Unkosten durch das Umschalten einzusparen.

[15] Das sogenannte *Blockieren von Threads* kann hier den Ausschlag geben. Kann die Verarbeitung einer Teilaufgabe Ihres Programms, bedingt durch eine außerhalb der Kontrolle des Programms liegende Ursache, nicht ausgeführt werden, so sagen wir, der entsprechende Thread sei *blockiert* oder befinde sich im *blockierten Zustand*. Ein sequentielles Programm „steht“ nun solange, bis sich die externe Bedingung ändert. Bei einem threadbasierten Programm kann unter Umständen die Verarbeitung einiger der übrigen Teilaufgaben fortgesetzt werden, während die Ausführung eines Programmteils blockiert ist, so daß das Programm weiterläuft. Hinsichtlich der Verarbeitungsgeschwindigkeit ist die Ausführung eines threadbasierten Programms auf einem Rechner mit nur

einem Prozessor nur dann sinnvoll, wenn es eine Teilaufgabe gibt, deren Verarbeitung eventuell blockiert wird.

[16] Die ereignisgesteuerte Programmierung ist ein sehr häufig zitiertes Beispiel für erhöhte Verarbeitungsgeschwindigkeit im Einprozessorbetrieb. Tatsächlich sind reaktionsfähige Benutzerschnittstellen einer der am stärksten zwingenden Anlässe für Threadprogrammierung. Stellen Sie sich ein Programm mit einer langwierigen Operation vor, während deren Ausführung Benutzereingaben nicht beachtet werden und das Programm nicht reagiert. Selbst wenn es eine „Quit"-Schaltfläche gibt, möchten Sie deren Status nicht überall im Programm abfragen müssen. Daraus würde sich ein schwer verständlicher Quelltext ergeben, ohne Garantie, daß ein späterer Wartungsprogrammierer nicht vergißt, die Prüfung durchzuführen. Ohne Threadprogrammierung besteht die einzige Möglichkeit, eine reaktionsfähige Benutzerschnittstelle zu schreiben darin, daß alle Teilaufgaben periodisch auf Benutzereingaben achten. Durch einen separaten Ausführungsfaden (Thread) der auf Benutzereingaben wartet, erhält das Programm garantiert einen gewissen Grad an Reaktionsfähigkeit, auch wenn dieser Thread die meiste Zeit über blockiert ist.

[17] Das Programm muß einerseits seine Verarbeitung fortsetzen, andererseits aber die Kontrolle an die Benutzerschnittstelle zurückgeben, um die Kommunikation mit dem Benutzer zu ermöglichen. Eine normale Methode ist aber nicht in der Lage, gleichzeitig ihre eigenen Operationen weiter zu verarbeiten und die Kontrolle an das restliche Programm zurückzugeben. Eigentlich unmöglich, da der Prozessor zwei Stellen zugleich verarbeiten müßte. Bei der Ausführung eines threadbasierten Programms entsteht aber genau dieser Eindruck (bei einem Mehrprozessorsystem ist es mehr als nur ein Eindruck).

[18] Die Verwendung von Prozessen auf Betriebssystemebene ist ein geradliniger Weg, um Threadunterstützung zu implementieren. Ein Prozeß ist ein selbständiges Programm mit eigenem Adressenraum. Ein multitaskingfähiges Betriebssystem kann mehr als einen Prozeß (Programm) zugleich verarbeiten, in dem der Prozessor periodisch von einem Prozeß zum nächsten weitergeschaltet wird, während scheinbar jeder Prozeß einzeln ausgeführt wird. Prozesse sind sehr attraktiv, weil das Betriebssystem in der Regel die einzelnen Prozesse voneinander isoliert, so daß sie sich nicht gegenseitig beinflussen können. Diese Eigenschaft macht das Programmieren mit Prozessen relativ einfach. Im Gegensatz dazu teilen sich die Threads bei der Threadunterstützung von Java Resourcen wie Arbeitsspeicher und das Ein-/Ausgabesystem. Das fundamentale Problem beim Schreiben eines threadbasierten Java-Programms besteht darin, den Zugriff der Threads auf diese Resourcen zu koordinieren, so daß stets höchstens ein Thread Zugriff erhält.

[19] Ein einfaches Beispiel für die Nutzung von Betriebssystemprozessen: Während ich an einem Buch arbeite, lege ich regelmäßig mehrere redundante Sicherungskopien des aktuellen Standes an. Eine Kopie in ein lokales Verzeichnis, eine auf einen Memorystick, eine auf eine Zip-Disk und noch eine auf ein entferntes FTP-Konto. Um diesen Vorgang zu automatisieren, habe ich ein kleines Programm geschrieben (in Python, aber die Konzepte sind identisch), welches die Dateien in einer *.zip* Datei archiviert, deren Name eine Versionsbezeichnung enthält und anschließend die verschiedenen Kopien verteilt. Anfangs habe ich alle Kopien sequentiell angelegt und stets auf das Ende eines Kopiervorgangs gewartet, bis ich mit den nächsten begann. Dann fiel mir auf, daß jeder Kopiervorgang abhängig von der Geschwindigkeit des Ein-/Ausgabesystems bezüglich des Zielmediums unterschiedlich viel Zeit brauchte. Da ich ein multitaskingfähiges Betriebssystem zur Verfügung hatte, konnte ich jede Kopieroperation als separaten Prozeß starten und die vier Operationen parallel laufen lassen, wodurch das gesamte Programm schneller wurde. Während ein Prozeß blockiert ist, kann ein anderer weiter verarbeitet werden.

[20] Dies ist ein ideales Beispiel für parallele Verarbeitung. Jede Teilaufgabe wird als Prozeß mit eigenem Adressraum ausgeführt, so daß keine Möglichkeit zur gegenseitigen Beeinflussung der Teilaufgaben besteht. Wichtiger noch, es besteht für die Teilaufgaben auch keine Notwendigkeit, um

miteinander zu kommunizieren, da jede von den übrigen völlig unabhängig ist. Das Betriebsystem kümmert sich um alle Einzelheiten, um das korrekte Kopieren der Dateien zu gewährleisten. Insgesamt betrachtet, bekommen Sie kostenlos ein schnelleres Programm, ohne ein Risiko in Kauf nehmen zu müssen.

[21] Manche Kollegen vertreten sogar die Auffassung, Prozesse seien der einzige vernünftige Ansatz zur Threadunterstützung.[1] Bedauerlicherweise gibt es generelle Einschränkungen hinsichtlich der Anzahl von Prozessen und der durch sie verursachten Unkosten, welche die Abbildung des gesamten Spektrums der Threadprogrammierung auf Prozesse verhindern.

[22] Einige Programmiersprachen sind so entworfen, daß sie gleichzeitig verarbeitete Programmteile voneinander trennen. Bei diesen sogenannten *funktionalen Sprachen* haben Funktionsaufrufe keine Seiteneffekte (können andere Funktionen also nicht beeinflussen) und können als unabhängige Teilaufgaben verarbeitet werden. Ein Beispiel für eine funktionale Programmiersprache ist Erlang. Erlang beinhaltet sichere Mechanismen, über die Teilaufgaben miteinander kommunizieren können. Wenn ein Teil Ihres Programms erheblich auf paralleler Programmierung aufbaut und Sie beim Entwickeln dieses Programmteils auf ausufernde Schwierigkeiten stoßen, sollten Sie in Betracht ziehen, diesen Teil in einer für diesen Zweck bestimmten Sprache wie Erlang zu schreiben.

[23] Für Java wurde der eher traditionelle Ansatz gewählt, die Threadunterstützung auf die sequentielle Sprache aufzusetzen.[2] Statt in einem multitaskingfähigen Betriebssystem einen externen Prozeß zu erzeugen, verwendet die Threadunterstützung von Java „Unterprozesse“ des Prozesses, der das Programm verarbeitet. Ein Vorteil dieses Ansatzes besteht in der Transparenz hinsichtlich des unterliegenden Betriebssystems, einem wichtigen Ziel beim Entwurf von Java. Die älteren Versionen des Macintosh Betriebssystems vor OSX (einer einigermaßen wichtigen Zielplattform für die ersten Java-Versionen) unterstützten Multitasking beispielsweise nicht. Würde Java nicht von sich aus threadbasierte Programme unterstützen, so könnte kein solches Java-Programm auf Macintosh oder eine ähnliche Plattform portiert werden und das Credo „write once/run everywhere“ wäre verletzt.[3]

22.1.2 Handlicheres Design

[24] Ein auf einem Einprozessorsystem ausgeführtes Programm, welches aus mehreren parallel zu verarbeitenen Teilen besteht, verarbeitet stets höchstens *einen* Programmteil. Es sollte daher theoretisch möglich sein, dasselbe Programm so umzuschreiben, daß es nicht aus parallel zu verarbeitenden Teilen besteht. Die Threadprogrammierung bringt aber auch einen wesentlichen organisatorischen Vorteil mit sich: Das Design Ihres Programms kann erheblich vereinfacht werden. Manche Probleme, etwa bei Simulationen, sind ohne Threadunterstützung schwierig zu lösen.

[25] Die meisten Menschen kennen wenigstens eine Form von Simulation, entweder als Computerspiel oder als Computeranimation in Filmen. Simulationen beinhalten in der Regel viele miteinander interagierende Elemente, von denen die meisten „einen eigenen Kopf haben“. Obwohl Sie beim Beoachten der Simulation auf einem Einprozessorsystem wissen, daß jedes Element von diesem einen Prozessor gesteuert wird, ist es aus der Perspektive des Programmierers leichter, sich jedes Element der Simulation als unabhängigen Programmteil vorzustellen, der von einem eigenen Prozessor verarbeitet wird.

[1] Beispielsweise führt Eric S. Raymond in seinem Buch *The Art of UNIX Programming*, Addison-Wesley (2004) einige sehr überzeugende Argument an.

[2] Gegner argumentieren, daß das Aufsetzen der Threadunterstützung auf eine sequentielle Programmiersprache zum Scheitern verurteilt ist, aber Sie müssen selbst zu einem Urteil kommen.

[3] Dieses Ziel wurde nie tatsächlich erreicht und wird mittlerweile von Sun Microsystems weniger vehement angepriesen. Ironischerweise könnte der Grund dafür mit Problemen in der Threadunterstützung zusammenhängen. Eventuell wurden diese Probleme aber auch mit der SE 5 gelöst.

[26] Eine umfangreiche Simulation kann eine sehr große Anzahl von Teilaufgaben beinhalten, da jedes Element unabhängig von den anderen agieren kann (dies gilt nicht nur für Elfen und Zauberer, sondern auch für Türen und Felsen). Umgebungen mit Threadunterstützung begrenzen häufig die Anzahl verfügbarer Threads auf die Größenordnungen Zehn oder Hundert. Diese Höchstanzahl kann außerhalb des Einflußbereiches des Programms variieren und von der Plattform oder bei Java von der Version der Laufzeitumgebung abhängen. Unter Java können Sie im allgemeinen davon ausgehen, daß nicht genügend Threads für alle Elemente einer großen Simulation zur Verfügung stehen.

[27] Ein typischer Lösungsansatz für dieses Problem ist die sogenannte *kooperative Threadverarbeitung*. Die Threadverarbeitung bei Java ist dagegen *präemptiv*, das heißt es gibt einen Threadscheduler, der die Threads basierend auf Zeitscheiben verarbeitet. Die Verarbeitung des aktuellen Threads wird periodisch ausgesetzt und ein anderer Thread zur Verarbeitung ausgewählt, so daß jeder Thread ein sinnvolles Kontingent an Rechenzeit erhält, um seine Aufgabe zu verarbeiten. Bei der kooperativen Threadverarbeitung tritt der gegenwärtig verarbeitete Thread „freiwillig“ von seiner Ausführung zurück, das heißt der Programmierer ist angehalten, bewußt entsprechende Anweisungen in die einzelnen Programmteile einzusetzen. Die kooperative Threadverarbeitung hat zwei Vorteile: Die Umschaltung von einem Thread auf einen anderen ist erheblich günstiger als beim präemptiven Ansatz und die Anzahl der gleichzeitig verarbeiteten unabhängigen Aufgaben ist theoretisch unbegrenzt. Bei vielen Simulationselementen kann koopertive Threadverarbeitung die Ideallösung sein. Beachten Sie allerdings, daß es Systeme mit kooperativer Threadverarbeitung gibt, die entwurfsbedingt nicht in der Lage sind, Aufgaben auf mehrere Prozessoren zu verteilen (eventuell eine erhebliche Einschränkung).

[28] Parallele Verarbeitung ist ein sehr wertvolles Modell für den Fall, daß Sie mit einem modernen messaging system arbeiten, das aus vielen voneinander unabhängigen, über ein Netzwerk verteilten Rechnern besteht. In diesem Fall laufen sämtliche Prozesse völlig unabhängig voneinander ab und es besteht nicht einmal die Gelegenheit, Resourcen gemeinsam zu nutzen. Die Informationsübertragung zwischen den Prozessen muß dennoch synchronisiert („zeitlich aufeinander abgestimmt“) werden, damit das messaging system keine Informationen verliert oder Informationen zu einem falschen Zeitpunkt einbaut. Auch wenn Sie nicht beabsichtigen, sich in absehbarer Zukunft mit Threadprogrammierung zu beschäftigen, ist es sinnvoll, die Grundzüge zu verstehen, damit Sie die Architekturen von messaging systems erfassen können, die zum vorherrschenden Verfahren beim Aufbau verteilter Systeme geworden sind.

[29] Threadprogrammierung verursacht Unkosten, insbesondere durch erhöhte Komplexität, die aber in der Regel durch das verbesserte Anwendungsdesign, bessere Resourcenauslastung (*resource balancing*) und bequemere Bedienbarkeit für die Benutzer ausgeglichen werden. Generell gestatten Threads, ein weniger stark gekoppeltes Design. Andernfalls wären Teile Ihres Programms gezwungen, explizit auf die Verarbeitung von Aufgaben zu achten, die üblicherweise von Threads ausgeführt werden.

22.2 Grundlagen der Threadprogrammierung

[30] Die Threadprogrammierung gestattet Ihnen, ein Programm in separate, unabhängig voneinander verarbeitungsfähige Aufgaben zu untergliedern. Bei einem threadbasierten Programm wird jede dieser eigenständigen Aufgaben (*tasks*) von ein einem Ausführungsfaden (Thread) verarbeitet. Ein Thread ist ein ein einzelner sequentieller flow of control innerhalb eines Prozesses. Ein einzelner Prozeß kann also mehrere parallel zu verarbeitende Aufgaben enthalten, wobei Sie aber programmieren, als ob jede Aufgabe den Prozessor für sich alleine zur Verfügung hat. Es gibt einen Mechanismus, der die Rechenzeit des Prozessors einteilt. In der Regel brauchen Sie sich nicht darum zu kümmern.

[31] Das Threadmodell erleichtert das Programmieren, wenn es darum geht, mehrere parallel zu verarbeitende Operationen in einem einzigen Programm unter einen Hut zu bringen. Der Prozessor wird weitergeschaltet, so daß jede Aufgabe einen Teil der Rechenzeit abbekommt.[4] Jeder Thread „glaubt“, den Prozessor für sich alleine zu haben, obwohl die Rechenzeit eigentlich unter allen Threads aufgeteilt wird (ausgenommen dann, wenn das Programm auf einem Mehrprozessorsystem betrieben wird). Eine großartige Eigenschaft der Threadprogrammierung besteht darin, daß Sie durch Abstraktion von dieser Ebene getrennt sind, sich also bei der Entwicklung Ihres Quelltextes keine Gedanken darüber machen müssen, ob Ihr Programm auf einem oder mehreren Prozessoren laufen wird. Die Verwendung von Threads ist somit eine Möglichkeit, ~~transparently scalable~~ Programme zu schreiben: Läuft ein Programm zu langsam, so läßt sich die Verarbeitungsgeschwindigkeit mühelos steigern, indem Sie weitere Prozessoren hinzunehmen. Multitaskingfähige Betriebssysteme und Threadprogrammierung ~~tend to be~~ die vernünftigste Möglichkeit, um Mehrprozessorsysteme zu nutzen.

22.2.1 Definieren von Aufgaben

[32] Ein Thread verarbeitet eine Aufgabe (*task*). Sie brauchen also eine Möglichkeit, um eine solche Aufgabe zu beschreiben. Eine solche Möglichkeit ist das Interface `java.lang.Runnable`. Sie definieren eine Aufgabe, indem Sie das Interface `Runnable` implementieren, das heißt die dort deklarierte Methode `run()` auszuprogrammieren, um die Aufgabe zu formulieren. Das folgende Beispiel zeigt den Countdown vor dem Start an:

```
//: concurrency/LiftOff.java
// Demonstration of the Runnable interface.

public class LiftOff implements Runnable {
    protected int countDown = 10; // Default
    private static int taskCount = 0;
    private final int id = taskCount++;
    public LiftOff() {}
    public LiftOff(int countDown) {
        this.countDown = countDown;
    }
    public String status() {
        return "#" + id + "(" +
            (countDown > 0 ? countDown : "Liftoff!") + "), ";
    }
    public void run() {
        while(countDown-- > 0) {
            System.out.print(status());
            Thread.yield();
        }
    }
} ///:~
```

Das `id`-Feld dient zur Unterscheidung der einzelnen `LiftOff`-Objekte (Aufgaben). Das Feld ist als `final` deklariert, da nach seiner Initialisierung nicht mehr mit einer Änderung zu rechnen ist.

[33] Die `run()`-Methode einer Aufgabe enthält üblicherweise eine Schleife, deren Körper sooft durchlaufen wird, bis die Aufgabe nicht länger benötigt wird, das heißt, daß Sie eine Abbruchbedingung

[4] Gilt für Betriebssysteme die mit Zeitscheiben arbeiten, zum Beispiel Windows. Solaris verwendet ein FIFO-Modell zur Threadverarbeitung: Solange kein Thread mit höherer Priorität „erwacht“, wird der aktuelle Thread entweder bis zum Ende oder zu seiner Blockierung verarbeitet. Daraus ergibt sich, daß die Verarbeitung anderer Thread derselben Prioritätsstufe solange unterbleibt, bis der aktuelle Thread den Prozessor „hergibt“.

formulieren müssen, um die Schleife verlassen zu können (zum Beispiel per `return`-Anweisung). Häufig enthält die `run()`-Methode eine unendliche Schleife, das heißt eine Aufgabe deren Verarbeitung niemals abgeschlossen wird, sofern die `run()`-Methode nicht durch irgendeinen Einfluß beendet wird. (Sie lernen in Abschnitt 22.4 wie die Verarbeitung von Aufgaben sicher beendet werden kann.)

[34] Der Aufruf der statischen `java.lang.Thread`-Methode `yield()` im Körper der `run()`-Methode schlägt dem Threadscheduler (dem Mechanismus in der Threadverarbeitung unter Java, welcher den Threads Rechenzeit zuweist) vor, einen anderen Thread zur Verarbeitung auszuwählen. Das Aufrufen der `yield()`-Methoden ist an dieser Stelle völlig unverbindlich, gestaltet aber die Ausgabe des Beispiels interessanter. Mit `yield()` ist es wahrscheinlicher, daß Sie das Umschalten des Threadschedulers beobachten können.

[35] Im folgenden Beispiel wird die `run()`-Methode der `LiftOff`-Aufgabe nicht von einem separaten Thread gestartet, sondern in der `main()`-Methode direkt aufgerufen (genau genommen ist auch hier ein Thread im Spiel, nämlich der `main`-Thread, welcher die `main()`-Methode verarbeitet):

```
//: concurrency/MainThread.java
public class MainThread {
    public static void main(String[] args) {
        LiftOff launch = new LiftOff();
        launch.run();
    }
} /* Output:
      #0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
   *///:~
```

Jede Klasse, die das Interface *`Runnable`* implementiert, muß eine `run()`-Methode besitzen. Diese Methode ist nichts besonderes und implementiert keine angeborenen Threadfähigkeiten. Threadverhalten entsteht dann, wenn Sie explizit eine Aufgabe mit einem Thread verknüpfen.

22.2.2 Die Klasse Thread

[36] Der traditionelle Weg, um ein *`Runnable`*-Objekt in eine ausführbare Aufgabe zu verwandeln, besteht darin, es einem Konstruktor der Klasse `Thread` zu übergeben. Das nächste Beispiel zeigt die Verarbeitung eines Objektes der Klasse `LiftOff` per `Thread`-Objekt:

```
//: concurrency/BasicThreads.java
// The most basic use of the Thread class.

public class BasicThreads {
    public static void main(String[] args) {
        Thread t = new Thread(new LiftOff());
        t.start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (90% match)
      Waiting for LiftOff
      #0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
   *///:~
```

[37] Es genügt, dem `Thread`-Konstruktor ein *`Runnable`*-Objekt zu übergeben. Durch Aufrufen der `start()`-Methode des `Thread`-Objektes wird die erforderliche Initialisierung bewerkstelligt und die `run()`-Methode des *`Runnable`*-Objektes aufgerufen, um die Verarbeitung der Aufgabe mittels eines neuen Threads auszulösen. Obwohl `start()` eine Methode mit langer Laufzeit ist, wird die Meldung „Waiting for LiftOff“ ausgegeben, bevor der Countdown abgelaufen ist (siehe Ausgabe), das heißt das Programm kehrt schnell aus dem Aufruf der `start()`-Methode zurück. Eigentlich haben Sie die

`run()`-Methode des `LiftOff`-Objektes aufgerufen, deren Verarbeitung noch nicht beendet ist. Da die `run()`-Methode aber von einem separaten Thread verarbeitet wird, kann der `main`-Thread weitere Anweisungen verarbeiten. (Die Fähigkeit, selbst Threads zu erzeugen, ist nicht auf den `main`-Thread beschränkt. Jeder Thread kann weitere Threads erzeugen.) Das Programm führt also zwei Methoden auf einmal aus, nämlich `main()` und `run()`. Die `run()`-Methode enthält jene Anweisungen, die „zugleich“ mit den übrigen Threads eines Programmes verarbeitet werden.

[38] Sie können mühelos weitere Threads erzeugen, um weitere Aufgaben verarbeiten zu können. Das folgende Beispiel veranschaulicht die gemeinsame Verarbeitung mehrerer Threads[5]:

```
//: concurrency/MoreBasicThreads.java
// Adding more threads.

public class MoreBasicThreads {
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new Thread(new LiftOff()).start();
        System.out.println("Waiting for LiftOff");
    }
} /* Output: (Sample)
      Waiting for LiftOff
      #0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7),
#1(7), #2(7), #3(7), #4(7), #0(6), #1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5),
#3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3), #2(3), #3(3), #4(3),
#0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
   *///:~
```

Die Ausgabe dokumentiert, daß die verschiedenen Aufgaben infolge der Aus- und Abwahl ihrer Threads „durcheinander“ verarbeitet werden. Die Auswahl eines Threads zur Verarbeitung (und damit die Abwahl des aktuellen Threads) wird durch den Threadscheduler vorgenommen. Verfügt Ihr Rechner über mehr als einen Prozessor, so verteilt der Threadscheduler die Threads stillschweigend auf die vorhandenen Prozessoren.[6]

[39] Die Ausgabe des Programms fällt bei jedem Programmaufruf anders aus, da der Threadscheduler nicht deterministisch arbeitet. Die Ausgaben können sich sogar zwischen zwei Versionen des Java Development Kits drastisch unterscheiden. Beispielsweise verwendete eine frühere JDK-Version so grobe Zeitscheiben, daß erst Thread 1 vollständig verarbeitet wurde, dann Thread 2 und so weiter. Diese Funktionalität entspricht praktisch dem Aufrufen einer Methode, die alle Schleifen nacheinander ausführt (wobei das Starten der Threads zusätzliche Unkosten bewirkt). Die Threadverarbeitung mittels Zeitscheiben wurde bei den aktuelleren JDK-Versionen verbessert, so daß nun alle Threads regelmäßiger an die Reihe kommen. Solche Änderungen am Verhalten des Java Development Kits werden von Sun Microsystems in der Regel nicht erwähnt, so daß Sie sich nicht auf ein konsistentes Verhalten verlassen können. Am besten gehen Sie in der Threadprogrammierung konverativ vor.

[40] In der `main()`-Methode werden keine Referenzen auf die `Thread`-Objekte gespeichert. Ein gewöhnliches Objekt wäre damit eine leicht Beute für die automatische Speicherbereinigung, nicht aber ein `Thread`-Objekt. Ein `Thread`-Objekt „registriert“ sich selbst, so daß eine Referenz auf dieses Objekt existiert und die automatische Speicherbereinigung keinen Zugriff erhält, bevor die `run()`-Methode der verarbeiteten Aufgabe vollständig abgelaufen und der Thread „gestorben“ ist. Sie sehen an der Ausgabe, daß die einzelnen Aufgaben bis zum Ende verarbeitet werden, das heißt *so a thread creates a separate thread of execution that persists after the call to start() completes.*

[5] In diesem Beispiel erzeugt ein einzelner Thread (der `main`-Thread) alle `LiftOff`-Threads. Existieren dagegen mehrere Threads, die `LiftOff`-Threads erzeugen, so können zwei oder mehr `LiftOff`-Threads identische `id`-Felder haben. Der Grund hierfür wird in Abschnitt 22.3 erläutert.

[6] Nicht bei einigen der frühesten Java-Versionen.

Übungsaufgabe 1: (2) Implementieren Sie das Interface *`Runnable`*. Geben Sie in der `run()`-Methode eine Meldung aus und rufen Sie anschließend die `yield()`-Methode auf. Wiederholen Sie diese Kombination dreimal, bevor Sie aus der `run()`-Methode zurückkehren. Legen Sie im Konstruktor eine Startmeldung und an geeigneter Stelle eine zweite Meldung an, die nach beendeter Verarbeitung der Aufgabe ausgegeben wird (das Anlegen der zweiten Meldung greift auf die Unterabschnitte 22.4.3 und 22.4.4 vor). Erzeugen Sie einige dieser Aufgaben und verarbeiteten Sie sie mit Hilfe von Threads. ■

Übungsaufgabe 2: (2) Definieren Sie eine Aufgabe, die die ersten n Fibonacci-Zahlen berechnet, wobei die Schranke n dem Konstruktor übergeben wird (orientieren Sie sich am Beispiel *generics/Fibonacci.java*). Erzeugen Sie einige dieser Aufgaben und verarbeiteten Sie sie mit Hilfe von Threads. ■

22.2.3 Exekutoren

[41] Die seit der SE 5 verfügbaren Exekutoren („Ausführer“) aus dem Package `java.util.concurrent` erleichtern die Threadprogrammierung, indem sie `Thread`-Objekte beaufsichtigen. Die Exekutoren bilden eine Indirektionsschicht zwischen dem Client und der Verarbeitung einer Aufgabe, das heißt der Client führt dabei Aufgaben nicht unmittelbar aus, sondern beauftragt ein intermediäres Hilfsobjekt damit (den Exekutor). Exekutoren überwachen die Verarbeitung asynchroner Aufgaben, ohne daß Sie selbst explizit in den Lebenszyklus der Threads eingreifen müssen. Exekutoren sind in Version 5/6 der Standard Edition die bevorzugte Vorgehensweise, um Threads zu starten.

[42] Im folgenden Beispiel verwenden wir einen Exekutor anstelle eines `Thread`-Objektes (vergleichen Sie dies mit dem Beispiel *MoreBasicThreads.java*). Ein `LiftOff`-Objekt „weiß“ wie es seine Aufgabe zu erfüllen hat und bietet, wie beim *Command*-Entwurfsmuster, eine einzelne Methode zum Aufruf an. Ein Objekt vom Typ *`java.util.concurrent.ExecutorService`* (ein Objekt vom Typ *`Executor`*, welches den Lebenszyklus von `Thread`-Objekten unterstützt, zum Beispiel durch eine `shutdown()`-Methode) „weiß“ wie der passende Kontext zur Verarbeitung eines *`Runnable`*-Objektes erzeugt wird. Die Klasse `CachedThreadPool` im folgenden Beispiel erzeugt einen Thread pro Aufgabe. Beachten Sie, daß das *`ExecutorService`*-Objekt mit Hilfe einer statischen Methode der Klasse `java.util.concurrent.Executors` erzeugt wird, die den Typ des Exekutors festlegt:

```
//: concurrency/CachedThreadPool.java
import java.util.concurrent.*;

public class CachedThreadPool {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output: (Sample)
      #0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8),
#0(6), #1(7), #2(7), #3(7), #4(7), #0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5),
#2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2), #1(3), #2(3), #3(3),
#4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
   *///:~
```

Häufig genügt ein einzelner Exekutor, um alle Aufgaben innerhalb eines Programms zu verarbeiten.

[43] Das Aufrufen der `shutdown()`-Methode bewirkt, daß dem Exekutor keine neuen Aufgaben mehr übergeben werden können. Der aktuelle Thread (in diesem Beispiel der Thread, welcher die

`main()`-Methode ausführt) fährt fort, alle vor dem Aufruf der `shutdown()`-Methode an den Exekutor übergebenen Aufgaben zu verarbeiten. Das Programm wird unmittelbar nach der Verarbeitung aller beim Exekutor „registrierten“ Aufgaben beendet.

[44] Sie können den „cached Threadpool“ (`newCachedThreadPool()`) im vorigen Beispiel ohne weiteres durch einen anderen Exekutortyp ersetzen. Ein „fixed Threadpool“ (`newFixedThreadPool()`) verwendet einen beschränkten Vorrat von Threads, um die übergebenen Aufgaben zu verarbeiten:

```
//: concurrency/FixedThreadPool.java
import java.util.concurrent.*;

public class FixedThreadPool {
    public static void main(String[] args) {
        // Constructor argument is number of threads:
        ExecutorService exec = Executors.newFixedThreadPool(5);
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output: (Sample)
      #0(9), #0(8), #1(9), #2(9), #3(9), #4(9), #0(7), #1(8), #2(8), #3(8), #4(8),
#0(6), #1(7), #2(7), #3(7), #4(7), #0(5), #1(6), #2(6), #3(6), #4(6), #0(4), #1(5),
#2(5), #3(5), #4(5), #0(3), #1(4), #2(4), #3(4), #4(4), #0(2), #1(3), #2(3), #3(3),
#4(3), #0(1), #1(2), #2(2), #3(2), #4(2), #0(Liftoff!), #1(1), #2(1), #3(1), #4(1),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
   *///:~
```

[45] Ein „fixed Threadpool“ erledigt das aufwändige Allokieren der Threads einmalig zu Beginn und Sie legen dabei die Anzahl von Threads fest. Das spart Zeit, da nicht bei jedem erzeugten Thread (bei jeder Aufgabe) konstante Unkosten anfallen. Threadgestützte Ereignisbehandler bei ereignisgesteuerten Programmen können schnell verarbeitet werden, in dem Sie einen Thread aus dem Pool verwenden. Dabei können die verfügbaren Resourcen nicht überschritten werden, da ein „fixed Threadpool“ nur eine begrenzte Anzahl von Threads zur Verfügung stellt.

[46] Beachten Sie, daß bereits vorhandene Threads unabhängig vom Typ des Threadpools nach Möglichkeit wiederverwendet werden.

[47] Ziehen Sie bei Anwendungen im produktiven Betrieb einen „fixed Threadpool“ anstelle eines „cached Threadpools“ in Betracht, auch wenn im Buch die letzteren verwendet werden. Ein „cached Threadpool“ erzeugt im allgemeinen soviele Threads als während der Verarbeitung des Programms erforderlich sind und stellt anschließend das Erzeugen neuer Threads zugunsten der Wiederverwendung bereits vorhandener Threads ein. Somit ist ein „cached Threadpool“ eine sinnvoll erste Wahl. Nur wenn diese Wahl Schwierigkeiten verursacht, müssen Sie auf einen „fixed Threadpool“ umsteigen.

[48] Ein „single Threadexekutor“ (`newSingleThreadExecutor()`) verhält sich wie ein „fixed Threadpool“ mit nur einem Thread[7] und eignet sich für Aufgaben mit langer Laufzeit, die mittels eines eigenen Threads verarbeitet werden sollen, etwa das Warten auf Socketverbindungsanfragen. Ein „single Threadexekutor“ eignet sich aber auch für kurze Aufgaben, wie das Aktualisieren einer lokalen oder entfernten Protokolldatei oder für einen Ereignisbehandlerthread.

[49] Werden einem „single Threadexekutor“ mehrere Aufgaben übergeben, so gelangen sie in eine Warteschlange. Eine Aufgabe wird vollständig verarbeitet, bevor die nächste an die Reihe kommt

[7] Ein „single Threadexekutor“ bietet außerdem eine Garantie, welche die übrigen Typen nicht haben: Keine zwei Aufgaben werden jemals zugleich verarbeitet. Dadurch ändern sich die Anforderungen an den Sperrmechanismus für Aufgaben. (Sperrmechanismen werden in Unterabschnitt 22.3.2 behandelt.)

und alle Aufgaben werden von ein und demselben Thread verarbeitet. Das folgende Beispiel zeigt, daß die Aufgaben in der Reihenfolge ihrer Übergabe verarbeitet werden und jede Aufgabe vollständig verarbeitet wird, ehe die Verarbeitung der nächsten Aufgabe beginnt. Ein „single Threadexekutor“ verarbeitet seine Aufgaben also in einer definierten Reihenfolge und verwaltet sie in einer eigenen nach außen hin verborgenen Warteschlange:

```
//: concurrency/SingleThreadExecutor.java
import java.util.concurrent.*;

public class SingleThreadExecutor {
    public static void main(String[] args) {
        ExecutorService exec =
            Executors.newSingleThreadExecutor();
        for(int i = 0; i < 5; i++)
            exec.execute(new LiftOff());
        exec.shutdown();
    }
} /* Output:
      #0(9), #0(8), #0(7), #0(6), #0(5), #0(4), #0(3), #0(2), #0(1), #0(Liftoff!),
#1(9), #1(8), #1(7), #1(6), #1(5), #1(4), #1(3), #1(2), #1(1), #1(Liftoff!), #2(9),
#2(8), #2(7), #2(6), #2(5), #2(4), #2(3), #2(2), #2(1), #2(Liftoff!), #3(9), #3(8),
#3(7), #3(6), #3(5), #3(4), #3(3), #3(2), #3(1), #3(Liftoff!), #4(9), #4(8), #4(7),
#4(6), #4(5), #4(4), #4(3), #4(2), #4(1), #4(Liftoff!),
   *///:~
```

[50] Stellen Sie sich, als weiteres Beispiel, einige Threads vor, die auf das Dateisystem zugreifen. Wenn Sie diese Aufgaben mit Hilfe eines „single Threadexekutor“ verarbeiten, ist garantiert, daß stets höchstens ein Thread aktiv ist. Auf diese Weise sparen Sie sich die Synchronisierung bezüglich der gemeinsamen Resource. Gelegentlich ist die Synchronisierung dennoch die bessere Lösung (siehe später im diesem Kapitel). Ein „single Threadexekutor“ gestattet Ihnen bei einem Prototyp die Koordination beiseite zu lassen. Durch die Anordnung der Aufgaben in einer Reihenfolge, fällt die Notwendigkeit weg, eine Reihenfolge beim Zugriff auf die Resource zu erzwingen.

Übungsaufgabe 3: (1) Wiederholen Sie Übungsaufgabe 1 mit den verschiedenen in diesem Unterabschnitt gezeigten Typen von Exekutoren. ■

Übungsaufgabe 4: (1) Wiederholen Sie Übungsaufgabe 2 mit den verschiedenen in diesem Unterabschnitt gezeigten Typen von Exekutoren. ■

22.2.4 Aufgaben mit Rückgabewert

[51] Ein *Runnable*-Objekt repräsentiert eine separate Aufgabe nach deren Verarbeitung aber kein Wert zurückgegeben wird. Soll nach der Verarbeitung einer Aufgabe ein Wert zurückgegeben werden, so können Sie statt *Runnable* das Interface *java.util.concurrent.Callable* implementieren. Das seit der SE 5 vorhandene *Callable*-Interface ist generisch, hat also einen Typparameter, welcher den Rückgabetyp der `call()`-Methode (anstelle der `run()`-Methode) darstellt. Die `call()`-Methode wird über die *ExecutorService*-Methode `submit()` aufgerufen. Ein einfaches Beispiel:

```
//: concurrency/CallableDemo.java
import java.util.concurrent.*;
import java.util.*;

class TaskWithResult implements Callable<String> {
    private int id;
    public TaskWithResult(int id) {
        this.id = id;
```

```
    }
    public String call() {
        return "result of TaskWithResult " + id;
    }
}

public class CallableDemo {
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        ArrayList<Future<String>> results =
            new ArrayList<Future<String>>();
        for(int i = 0; i < 10; i++)
            results.add(exec.submit(new TaskWithResult(i)));
        for(Future<String> fs : results)
            try {
                // get() blocks until completion:
                System.out.println(fs.get());
            } catch(InterruptedException e) {
                System.out.println(e);
                return;
            } catch(ExecutionException e) {
                System.out.println(e);
            } finally {
                exec.shutdown();
            }
    }
} /* Output:
      result of TaskWithResult 0
      result of TaskWithResult 1
      result of TaskWithResult 2
      result of TaskWithResult 3
      result of TaskWithResult 4
      result of TaskWithResult 5
      result of TaskWithResult 6
      result of TaskWithResult 7
      result of TaskWithResult 8
      result of TaskWithResult 9
   *///:~
```

[52] Die `submit()`-Methode erzeugt ein *`java.util.concurrent.Future`*-Objekt, welches mit dem von dem *`Callable`*-Objekt zurückgegebenen Typ parametrisiert ist. Sie können per `isDone()` abfragen, ob die Erzeugung und Bewertung des *`Future`*-Objekts beendet ist. Ist die Verarbeitung abgeschlossen und der Rückgabewert erzeugt, so können Sie das Ergebnis per `get()` abfragen. Sie können die `get()`-Methode auch ohne vorheriges `isDone()` aufrufen, wobei `get()` solange blockiert, bis der Rückgabewert zur Verfügung steht. Sie können `get()` mit einer Zeitüberschreitung aufrufen oder aber zuerst `isDone()` und anschließend `get()` (ohne Zeitüberschreitung), um das Ergebnis abzuholen.

[53] Zwei Versionen der überladenen `Executors`-Methode `callable()` erwarten ein *`Runnable`*-Objekt und wandeln es in ein *`Callable`*-Objekt um. Das Interface *`ExecutorService`* deklariert mehrere `invokeXXX()`-Methoden, um Kollektionen von *`Callable`*-Objekten zu verarbeiten.

Übungsaufgabe 5: (2) Formulieren Sie die Aufgabe in Übungsaufgabe 2 als *`Callable`*-Objekt, welches die Werte der ersten n Fibonacci-Zahlen addiert. Erzeugen Sie mehrere Objekte und zeigen Sie die Ergebnisse an. ■

22.2.5 Die TimeUnit-Methode sleep()

[54] Es gibt eine einfache Möglichkeit, um das Verhalten eines Threads zu beeinflussen: Wenn Sie die die `sleep()`-Methode aufrufen, wird die Verarbeitung des zugehörigen Threads für eine gegebene Zeitdauer eingestellt. Wenn Sie in der Klasse `LiftOff` die Methode `yield()` durch `sleep()` ersetzen, erhalten Sie das folgende Beispiel:

```
//: concurrency/SleepingTask.java
// Calling sleep() to pause for a while.
import java.util.concurrent.*;

public class SleepingTask extends LiftOff {
    public void run() {
        try {
            while(countDown-- > 0) {
                System.out.print(status());
                // Old-style:
                // Thread.sleep(100);
                // Java SE5/6-style:
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch(InterruptedException e) {
            System.err.println("Interrupted");
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new SleepingTask());
        exec.shutdown();
    }
} /* Output:
      #0(9), #1(9), #2(9), #3(9), #4(9), #0(8), #1(8), #2(8), #3(8), #4(8), #0(7),
#1(7), #2(7), #3(7), #4(7), #0(6), #1(6), #2(6), #3(6), #4(6), #0(5), #1(5), #2(5),
#3(5), #4(5), #0(4), #1(4), #2(4), #3(4), #4(4), #0(3), #1(3), #2(3), #3(3), #4(3),
#0(2), #1(2), #2(2), #3(2), #4(2), #0(1), #1(1), #2(1), #3(1), #4(1), #0(Liftoff!),
#1(Liftoff!), #2(Liftoff!), #3(Liftoff!), #4(Liftoff!),
   *///:~
```

Die `sleep()`-Methode kann eine Ausnahme vom Typ `InterruptedException` auswerfen, die hier innerhalb der `run()`-Methode abgefangen wird. Da Threads Ausnahmen nicht an die `main()`-Methode weitergeben, müssen Ausnahmen bei der Verarbeitung von Aufgaben lokal behandelt werden. Seit der SE 5 verfügt auch der Aufzählungstyp `java.util.concurrent.TimeUnit` über eine `sleep()`-Methode. Sie verbessert die Lesbarkeit, indem sich die Einheit der durch `sleep()` bewirkten Verzögerung angeben läßt. Der Aufzählungstyp `TimeUnit` dient darüber hinaus für Umwandlungen zwischen Zeiteinheiten, etwa von Minuten in Sekunden (siehe beispielsweise auch Seite 956).

[55] Je nachdem welche Plattform Sie verwenden, können Sie eventuell beobachten, daß die Aufgaben „perfekt verteilt“ verarbeitet werden, das heißt in der Reihenfolge $0, 1, \ldots, 4, 0, 1, \ldots, 4, 0, \ldots$. Dieses Verhalten kommt dadurch zustande, daß jeder Thread nach seiner `print()`-Anweisung für eine Zehntelsekunde „schläft“ (blockiert wird), wodurch der Threadscheduler einen anderen Thread zur Verarbeitung auswählt. Das sequentielle Verhalten hängt aber von der Threadverarbeitung des unterliegenden Betriebssystems ab, Sie können sich also nicht auf ein bestimmtes Verhalten verlassen. Wenn Sie die Verarbeitungsreihenfolge Ihrer Aufgaben steuern müssen, verwenden Sie am besten die neuen Synchronisierungsklassen aus dem Package `java.util.concurrent` (siehe Unterabschnitt 22.7) oder Sie verzichten ganz auf Threads und legen eigene Methoden an, die sich gegenseitig in

einer bestimmten Reihenfolge aufrufen.

Übungsaufgabe 6: (2) Legen Sie einen Thread an, der für eine zufällig bestimmte Zeitdauer zwischen einer und zehn Sekunden „schläft“ und sich anschließend beendet. Erzeugen Sie eine (per Kommandozeile übergebene) Anzahl dieser Threads und starten Sie sie. ■

22.2.6 Threadprioritäten

[56] Die Priorität eines Threads gibt dem Threadscheduler gegenüber an, wie wichtig ein Thread ist. Obwohl die Reihenfolge in der der Prozessor eine Anzahl von Threads verarbeitet nicht deterministisch ist, neigt der Threadscheduler dazu, den wartenden Thread mit der höchsten Priorität zuerst auszuwählen. Letzteres bedeutet nicht, daß Threads mit geringerer Priorität nicht verarbeitet werden (Threadprioritäten verursachen keine Verklemmungen), sondern lediglich, daß sie weniger oft zur Verarbeitung ausgewählt werden.

[57] Threads sollten weitestgehend mit der Standardpriorität verarbeitet werden. Eingriffe in die Threadpriorität stellen sich in der Regel als Fehler heraus.

[58] Das folgende Beispiel führt die Wirkung von Threadprioritäten vor. Sie können die Priorität eines vorhandenen Threads per `getPriority()` abfragen und jederzeit per `setPriority()` ändern:

```
//: concurrency/SimplePriorities.java
// Shows the use of thread priorities.
import java.util.concurrent.*;

public class SimplePriorities implements Runnable {
    private int countDown = 5;
    private volatile double d; // No optimization
    private int priority;
    public SimplePriorities(int priority) {
        this.priority = priority;
    }
    public String toString() {
        return Thread.currentThread() + ": " + countDown;
    }
    public void run() {
        Thread.currentThread().setPriority(priority);
        while(true) {
            // An expensive, interruptable operation:
            for(int i = 1; i < 100000; i++) {
                d += (Math.PI + Math.E) / (double)i;
                if(i % 1000 == 0)
                    Thread.yield();
            }
            System.out.println(this);
            if(--countDown == 0) return;
        }
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new SimplePriorities(Thread.MIN_PRIORITY));
        exec.execute(new SimplePriorities(Thread.MAX_PRIORITY));
        exec.shutdown();
    }
} /* Output: (70% match)
```

```
    Thread[pool-1-thread-6,10,main]: 5
    Thread[pool-1-thread-6,10,main]: 4
    Thread[pool-1-thread-6,10,main]: 3
    Thread[pool-1-thread-6,10,main]: 2
    Thread[pool-1-thread-6,10,main]: 1
    Thread[pool-1-thread-3,1,main]: 5
    Thread[pool-1-thread-2,1,main]: 5
    Thread[pool-1-thread-1,1,main]: 5
    Thread[pool-1-thread-5,1,main]: 5
    Thread[pool-1-thread-4,1,main]: 5
    ...
*///:~
```

[59] Die `toString()`-Methode ist überschrieben, um die `Thread`-Version der `toString()`-Methode nutzen zu können, welche den Namen des Threads, seine Prioritätsstufe und seine Gruppenzugehörigkeit (siehe Unterabschnitt 22.2.13) angibt. Sie können den Namen eines Threads per Konstruktor festlegen. In diesem Beispiel werden die automatisch vergebenen Namen `pool-1-thread-1`, `pool-2-thread-2` und so weiter verwendet. Die überschriebene `toString()`-Methode zeigt außerdem den Countdown-Zählerstand der von diesem Thread verarbeiteten Aufgabe an. Beachten Sie, daß die statische `Thread`-Methode `currentThread()` eine Referenz auf das `Thread`-Objekt liefert, dessen Aufgabe gerade verarbeitet wird.

[60] Der zuletzt erzeugte Thread hat maximale Priorität, die übrigen fünf Threads haben minimale Priorität. Beachten Sie, daß die Priorität zu Beginn der `run()`-Methode festgelegt wird. Die Definition der Priorität im Konstruktor wäre nicht sinnvoll, da der Exekutor noch nicht mit der Verarbeitung der Aufgabe begonnen hat (die Threadpriorität richtet sich nach der Aufgabe).

[61] In der `run()`-Methode werden 100000 Wiederholungen einer vergleichsweise teuren Fließkommaberechnung ausgeführt, die aus zwei Additionen und einer Division besteht. Das `d`-Feld ist als volatiles Feld deklariert (siehe Unterabschnitt 22.3.3), um Optimierungen durch den Compiler zu unterdrücken. Ohne die Berechnung wäre die Wirkung der Prioritäten nicht erkennbar. (Versuchen Sie es und kommentieren Sie `for`-Schleife mit der Berechnung aus.) Mit der Berechnung ist erkennbar, daß dem Thread mit maximaler Priorität vom Threadscheduler Vorrang vor den übrigen Threads eingeräumt wird. (Ich habe dieses Verhalten zumindest unter Windows XP beobachtet.) Die Konsolenausgabe ist zwar teuer, taugt aber nicht zur Beobachtung der Threadprioritäten, da sie nicht vom Threadscheduler unterbrochen wird (die Ausgabe könnte andernfalls durch das Umschalten zwischen den Threads entstellt werden). Die Berechnung kann dagegen unterbrochen werden und nimmt genügend Zeit in Anspruch, damit sich der Threadscheduler zwischenschaltet, eine andere Aufgabe zur Verarbeitung auswählt und dabei auf die Priorität achtet, so daß Threads mit hoher Priorität Vorrang haben. Um sicher zu sein, daß der Threadscheduler umschaltet, wird regelmäßig die `yield()`-Methode aufgerufen.

[62] Die zehn Prioritätsstufen des Java Development Kit lassen sich nicht immer gut auf das unterliegende Betriebssystem abbilden. Windows hat zum Beispiel nur sieben Prioritätsstufen, ~~that are not fixed~~, so daß die Abbildung unbestimmt ist, Solaris hat dagegen 2^{31} Prioritätsstufen. Der einzige portable Ansatz besteht darin, die Stufen `Thread.MAX_PRIORITY`, `Thread.NORM_PRIORITY` und `Thread.MIN_PRIORITY` zu wählen, wenn Sie Threadprioritäten vergeben wollen.

22.2.7 Die yield()-Methode

[63] Wenn Sie nach einem Schleifendurchlauf in Ihrer `run()`-Methode einen sinnvollen Zwischenstand erreicht haben, können Sie dem Threadscheduler ein Zeichen geben, daß Sie einen Schritt vollzogen haben und nun ein anderer Thread ausgewählt werden kann. Ein Aufruf der `yield()`-Methode

unterbreitet dem Threadscheduler diesen Vorschlag (es ist tatsächlich nur ein Vorschlag; es gibt keine Garantie dafür, daß der Threadscheduler diesen Vorschlag befolgt). Ein Aufruf der `yield()`-Methode regt an, daß der Threadscheduler einen anderen Thread zur Verarbeitung auswählt.

[64] Im Beispiel *LiftOff.java* dient `yield()` dazu, die Rechenzeit gut unter allen Aufgaben aufzuteilen. Kommentieren Sie versuchsweise einmal die `yield()`-Methode aus, um den Unterschied zu sehen. Im allgemeinen aber ist `yield()` kein verläßlicher Mechanismus zur Steuerung oder Abstimmung eines Programms. In der Tat wird `yield()` sogar häufig falsch verwendet.

22.2.8 Hintergrundthreads

[65] Ein **Hintergrundthread** (*daemon thread*) hat die Aufgabe, während der Laufzeit eines Programms im Hintergrund einen Dienst verfügbar zu machen, ist aber selbst kein wesentlicher Bestandteil des Programms. Daher wird das Programm beendet, wenn die Verarbeitung aller **Vordergrundthreads** (*non-daemon threads*) abgeschlossen ist, das heißt, aller Threads, die keine Hintergrundthreads sind. Umgekehrt läuft ein Programm, solange noch Vordergrundthreads verarbeitet werden müssen. Die `main()`-Methode wird beispielsweise von einem Vordergrundthread verarbeitet.

```
//: concurrency/SimpleDaemons.java
// Daemon threads don't prevent the program from ending.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

public class SimpleDaemons implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + " " + this);
            }
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public static void main(String[] args) throws Exception {
        for(int i = 0; i < 10; i++) {
            Thread daemon = new Thread(new SimpleDaemons());
            daemon.setDaemon(true); // Must call before start()
            daemon.start();
        }
        print("All daemons started");
        TimeUnit.MILLISECONDS.sleep(175);
    }
} /* Output: (Sample)
      All daemons started
      Thread[Thread-0,5,main] SimpleDaemons@530daa
      Thread[Thread-1,5,main] SimpleDaemons@a62fc3
      Thread[Thread-2,5,main] SimpleDaemons@89ae9e
      Thread[Thread-3,5,main] SimpleDaemons@1270b73
      Thread[Thread-4,5,main] SimpleDaemons@60aeb0
      Thread[Thread-5,5,main] SimpleDaemons@16caf43
      Thread[Thread-6,5,main] SimpleDaemons@66848c
      Thread[Thread-7,5,main] SimpleDaemons@8813f2
      Thread[Thread-8,5,main] SimpleDaemons@1d58aae
      Thread[Thread-9,5,main] SimpleDaemons@83cc67
```

```
    ...
*///:~
```

Ein Hintergrundthread muß vor seinem Start per `setDaemon()` als solcher deklariert werden.

[66] Da nach dem Ende der Verarbeitung der `main()`-Methode nur noch Hintergrundthreads verarbeitet werden können, hindert nichts das Programm daran, sich zu beenden. Der `main`-Thread wird für kurze Zeit „schlafen gelegt“, damit Sie den Start der Hintergrundthreads beobachten können. Andernfalls wird nur ein Teil der Meldungen des `SimpleDaemons`-Threads ausgegeben. (Probieren Sie die `sleep()`-Methode mit verschiedenen Zeitintervallen aus.)

[67] Das obige Beispiel (*SimpleDaemons.java*) erzeugt explizit `Thread`-Objekte, um sie als Hintergrundthreads kennzeichnen zu können. Alternativ können Sie die Attribute „Threadtyp“, „Priorität“ und „Name“ der von einem Exekutor erzeugten Threads konfigurieren, indem Sie eine eigene Threadfabrik schreiben (das heißt das Interface *`java.util.concurrent.ThreadFactory`* implementieren):

```
//: net/mindview/util/DaemonThreadFactory.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        Thread t = new Thread(r);
        t.setDaemon(true);
        return t;
    }
} ///:~
```

[68] Der einzige Unterschied zu einer gewöhnlichen Threadfabrik besteht darin, daß die Variante `net.mindview.util.DaemonThreadFactory` jeden erzeugten Thread als Hintergrundthread kennzeichnet. Nur übergeben Sie der `Executors`-Methode `newCachedThreadPool()` ein Objekt der neuen Threadfabrikklasse:

```
//: concurrency/DaemonFromFactory.java
// Using a Thread Factory to create daemons.
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.Print.*;

public class DaemonFromFactory implements Runnable {
    public void run() {
        try {
            while(true) {
                TimeUnit.MILLISECONDS.sleep(100);
                print(Thread.currentThread() + '' '' + this);
            }
        } catch(InterruptedException e) {
            print(''Interrupted'');
        }
    }
    public static void main(String[] args) throws Exception {
        ExecutorService exec =
            Executors.newCachedThreadPool(new DaemonThreadFactory());
        for(int i = 0; i < 10; i++)
            exec.execute(new DaemonFromFactory());
        print(''All daemons started'');
        TimeUnit.MILLISECONDS.sleep(500); // Run for a while
    }
} /* (Execute to see output) *///:~
```

Alle statischen *ExecutorService*-Methoden zum Erzeugen von Threadpools sind überladen und haben eine Variante, die ein *ThreadFactory*-Objekt erwartet, mit dessen Hilfe neue Threads erzeugt werden.

[69] Daraus läßt sich eine Hilfsklasse entwickeln:

```
//: net/mindview/util/DaemonThreadPoolExecutor.java
package net.mindview.util;
import java.util.concurrent.*;

public class DaemonThreadPoolExecutor extends ThreadPoolExecutor {
    public DaemonThreadPoolExecutor() {
        super(0, Integer.MAX_VALUE, 60L, TimeUnit.SECONDS,
              new SynchronousQueue<Runnable>(),
              new DaemonThreadFactory());
    }
} ///:~
```

Die Argumente beim Aufruf des Basisklassenkonstruktors stammen aus dem Quelltext der Klasse `Executors` (*Executors.java*).

[70] Sie können per `isDaemon()` ermitteln, ob ein Thread ein Hintergrundthread ist. Ist ein Thread Hintergrundthread, so sind auch alle von ihm erzeugten Threads Hintergrundthreads, wie das folgende Beispiel zeigt:

```
//: concurrency/Daemons.java
// Daemon threads spawn other daemon threads.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Daemon implements Runnable {
    private Thread[] t = new Thread[10];
    public void run() {
        for(int i = 0; i < t.length; i++) {
            t[i] = new Thread(new DaemonSpawn());
            t[i].start();
            printnb("DaemonSpawn " + i + " started, ");
        }
        for(int i = 0; i < t.length; i++)
            printnb("t[" + i + "].isDaemon() = " + t[i].isDaemon() + ", ");
        while(true)
            Thread.yield();
    }
}

class DaemonSpawn implements Runnable {
    public void run() {
        while(true)
            Thread.yield();
    }
}

public class Daemons {
    public static void main(String[] args) throws Exception {
        Thread d = new Thread(new Daemon());
        d.setDaemon(true);
        d.start();
        printnb("d.isDaemon() = " + d.isDaemon() + ", ");
        // Allow the daemon threads to
        // finish their startup processes:
```

```
        TimeUnit.SECONDS.sleep(1);
    }
} /* Output: (Sample)
      d.isDaemon() = true, DaemonSpawn 0 started, DaemonSpawn 1 started,
      DaemonSpawn 2 started, DaemonSpawn 3 started, DaemonSpawn 4 started,
      DaemonSpawn 5 started, DaemonSpawn 6 started, DaemonSpawn 7 started,
      DaemonSpawn 8 started, DaemonSpawn 9 started, t[0].isDaemon() = true,
      t[1].isDaemon() = true, t[2].isDaemon() = true, t[3].isDaemon() = true,
      t[4].isDaemon() = true, t[5].isDaemon() = true, t[6].isDaemon() = true,
      t[7].isDaemon() = true, t[8].isDaemon() = true, t[9].isDaemon() = true,
   *///:~
```

Der `Daemon`-Thread ist ein Hintergrundthread. Die von ihm erzeugten Threads sind ebenfalls Hintergrundthreads, obwohl sie nicht ausdrücklich als solche markiert sind. Anschließend tritt der `Daemon`-Thread in eine unendliche Schleife ein, in der die `yield()`-Methode aufgerufen wird, damit auch die anderen Threads zur Verarbeitung ausgewählt werden können.

[71] Seien Sie sich der Tatsache bewußt, daß Hintergrundthreads ihre `run()`-Methoden einfach abbrechen, ohne eine eventuelle `finally`-Klausel zu beachten:

```
//: concurrency/DaemonsDontRunFinally.java
// Daemon threads don't run the finally clause
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class ADaemon implements Runnable {
    public void run() {
        try {
            print("Starting ADaemon");
            TimeUnit.SECONDS.sleep(1);
        } catch(InterruptedException e) {
            print("Exiting via InterruptedException");
        } finally {
            print("This should always run?");
        }
    }
}

public class DaemonsDontRunFinally {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new ADaemon());
        t.setDaemon(true);
        t.start();
    }
} /* Output:
      Starting ADaemon
   *///:~
```

Wenn Sie dieses Programm ausführen, werden Sie feststellen, daß die `finally`-Klausel nicht verarbeitet wird. Kommentieren Sie aber die `setDaemon()`-Methode aus, so wird die `finally`-Klausel berücksichtigt.

[72] Dieses Verhalten ist korrekt, obwohl Sie es angesichts der früheren Versprechen über die `finally`-Klausel nicht erwartet haben werden. Hintergrundthreads werden abrupt beendet, sobald die Verarbeitung des letzten Vordergrundthreads beendet ist. Sobald die Verarbeitung der `main()`-Methode beendet ist, bricht die Laufzeitumgebung sämtliche Hintergrundthreads ohne irgendeine der Formalitäten ab, mit denen Sie möglicherweise gerechnet haben. Da sich Hintergrundthreads nicht „sauber“ beenden lassen, sind sie nur selten eine gute Wahl. Exekutoren mit Vordergrund-

threads sind in der Regel ein besserer Ansatz, da alle von einem Exekutor kontrollierten Aufgaben auf einmal beendet werden können. Sie werden später in diesem Kapitel lernen, daß das Herunterfahren hier in geordneter Weise stattfindet.

Übungsaufgabe 7: (2) Probieren Sie im Beispiel *Daemons.java* bei `sleep()` unterschiedliche Zeitintervalle aus und beobachten Sie die Wirkung. ■

Übungsaufgabe 8: (1) Ändern Sie das Beispiel *MoreBasicThreads.java* so, daß alle Threads Hintergrundthreads sind und verifizieren Sie, daß das Programm beendet wird, sobald die Verarbeitung der `main()`-Methode abgeschlossen ist. ■

Übungsaufgabe 9: (3) Ändern Sie das Beispiel *SimplePriorities.java* so, daß die Threadpriorität über eine eigene Threadfabrik (eine Klasse, die das Interface *`ThreadFactory`* implementiert) definiert wird. ■

22.2.9 Implementierungsvarianten

[73] Abgesehen von Unterabschnitt 22.2.4 wurde bis jetzt bei allen Aufgaben das Interface *`Runnable`* implementiert. In sehr einfachen Fällen genügt eventuell auch eine direkt von `Thread` abgeleitete Klasse:

```
//: concurrency/SimpleThread.java
// Inheriting directly from the Thread class.

public class SimpleThread extends Thread {
    private int countDown = 5;
    private static int threadCount = 0;
    public SimpleThread() {
        // Store the thread name:
        super(Integer.toString(++threadCount));
        start();
    }
    public String toString() {
        return "#" + getName() + "(" + countDown + "), ";
    }
    public void run() {
        while(true) {
            System.out.print(this);
            if(--countDown == 0)
                return;
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new SimpleThread();
    }
} /* Output:
      #1(5), #1(4), #1(3), #1(2), #1(1), #2(5), #2(4), #2(3), #2(2), #2(1), #3(5),
      #3(4), #3(3), #3(2), #3(1), #4(5), #4(4), #4(3), #4(2), #4(1), #5(5), #5(4),
      #5(3), #5(2), #5(1),
   *///:~
```

Ein `Thread`-Objekt erhält durch Aufrufen des entsprechenden Konstruktors einen Namen. Die `toString()`-Methode fragt den Namen des Threads per `getName()` ab.

[74] Eine weitere Variante ist die „selbst startende“ *`Runnable`*-Implementierung:

```
//: concurrency/SelfManaged.java
// A Runnable containing its own driver Thread.

public class SelfManaged implements Runnable {
    private int countDown = 5;
    private Thread t = new Thread(this);
    public SelfManaged() { t.start(); }
    public String toString() {
        return Thread.currentThread().getName() +
            ''('' + countDown + ''), '';
    }
    public void run() {
        while(true) {
            System.out.print(this);
            if(--countDown == 0)
                return;
        }
    }
    public static void main(String[] args) {
        for(int i = 0; i < 5; i++)
            new SelfManaged();
    }
} /* Output:
      Thread-0(5), Thread-0(4), Thread-0(3), Thread-0(2), Thread-0(1), Thread-1(5),
      Thread-1(4), Thread-1(3), Thread-1(2), Thread-1(1), Thread-2(5), Thread-2(4),
      Thread-2(3), Thread-2(2), Thread-2(1), Thread-3(5), Thread-3(4), Thread-3(3),
      Thread-3(2), Thread-3(1), Thread-4(5), Thread-4(4), Thread-4(3), Thread-4(2),
      Thread-4(1),
   *///:~
```

Diese Variante unterscheidet sich eigentlich nicht von der vorigen (Ableitung einer neuen Klasse von `Thread`), ist aber syntaktisch etwas komplizierter. Andererseits ermöglicht die Implementierung eines Interfaces, die Klasse zugleich von einer anderen Klasse abzuleiten; im Gegensatz zum vorigen Beispiel.

[75] Beachten Sie, daß die `start()`-Methode im Konstruktor aufgerufen wird. Dieses Beispiel ist sehr einfach und daher wahrscheinlich sicher. Seien Sie sich aber der Tatsache bewußt, daß das Starten von Threads per Konstruktor problematisch sein kann, da möglicherweise die Verarbeitung eines anderen Threads bereits beginnt, bevor der Konstruktor beendet ist, die Aufgabe das Objekt also in einem instabilen Zustand vorfinden kann. Dies ist ein weiteres Argument dafür, Exekutoren zu verwenden, statt `Thread`-Objekte selbst zu erzeugen.

[76] Es ist gelegentlich sinnvoll, die Implementierung Ihres Threads mittels einer inneren Klasse zu verbergen:

```
//: concurrency/ThreadVariations.java
// Creating threads with inner classes.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

// Using a named inner class:
class InnerThread1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner extends Thread {
        Inner(String name) {
            super(name);
            start();
```

```
        }
        public void run() {
            try {
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    sleep(10);
                }
            } catch(InterruptedException e) {
                print(''interrupted'');
            }
        }
        public String toString() {
            return getName() + '': '' + countDown;
        }
    }
    public InnerThread1(String name) {
        inner = new Inner(name);
    }
}

// Using an anonymous inner class:
class InnerThread2 {
    private int countDown = 5;
    private Thread t;
    public InnerThread2(String name) {
        t = new Thread(name) {
                public void run() {
                    try {
                        while(true) {
                            print(this);
                            if(--countDown == 0) return;
                            sleep(10);
                        }
                    } catch(InterruptedException e) {
                        print(''sleep() interrupted'');
                    }
                }
                public String toString() {
                    return getName() + '': '' + countDown;
                }
            };
        t.start();
    }
}

// Using a named Runnable implementation:
class InnerRunnable1 {
    private int countDown = 5;
    private Inner inner;
    private class Inner implements Runnable {
        Thread t;
        Inner(String name) {
            t = new Thread(this, name);
            t.start();
        }
        public void run() {
            try {
```

```
                while(true) {
                    print(this);
                    if(--countDown == 0) return;
                    TimeUnit.MILLISECONDS.sleep(10);
                }
            } catch(InterruptedException e) {
                print(''sleep() interrupted'');
            }
        }
        public String toString() {
            return t.getName() + '': '' + countDown;
        }
    }
    public InnerRunnable1(String name) {
        inner = new Inner(name);
    }
}

// Using an anonymous Runnable implementation:
class InnerRunnable2 {
    private int countDown = 5;
    private Thread t;
    public InnerRunnable2(String name) {
        t = new Thread(new Runnable() {
                public void run() {
                    try {
                        while(true) {
                            print(this);
                            if(--countDown == 0) return;
                            TimeUnit.MILLISECONDS.sleep(10);
                        }
                    } catch(InterruptedException e) {
                        print(''sleep() interrupted'');
                    }
                }
                public String toString() {
                    return Thread.currentThread().getName() +
                        '': '' + countDown;
                }
            }, name);
        t.start();
    }
}

// A separate method to run some code as a task:
class ThreadMethod {
    private int countDown = 5;
    private Thread t;
    private String name;
    public ThreadMethod(String name) { this.name = name; }
    public void runTask() {
        if(t == null) {
            t = new Thread(name) {
                    public void run() {
                        try {
                            while(true) {
                                print(this);
                                if(--countDown == 0) return;
```

```
                                sleep(10);
                            }
                        } catch(InterruptedException e) {
                            print("sleep() interrupted");
                        }
                    }
                    public String toString() {
                        return getName() + ": " + countDown;
                    }
                };
            t.start();
        }
    }
}

public class ThreadVariations {
    public static void main(String[] args) {
        new InnerThread1("InnerThread1");
        new InnerThread2("InnerThread2");
        new InnerRunnable1("InnerRunnable1");
        new InnerRunnable2("InnerRunnable2");
        new ThreadMethod("ThreadMethod").runTask();
    }
} /* (Execute to see output) *////:~
```

[77] Die Klasse `InnerThread1` enthält eine benannte innere Klasse (`Inner`), die von `Thread` abgeleitet ist. Dies ist sinnvoll, wenn die innere Klasse spezielle Fähigkeiten (Methoden) hat, die Sie in Methoden der äußeren Klasse benötigen. Im allgemeinen wird aber nur dann eine Klasse von `Thread` abgeleitet, wenn die `Thread`-Funktionalität erforderlich ist, das heißt es ist nicht notwendig, eine *benannte* innere Klasse zu wählen. Die Klasse `InnerThread2` dokumentiert diese Alternative: Der Konstruktor definiert eine anonyme von `Thread` abgeleitete innere Klasse. Eine Referenz auf ein Objekt dieser inneren Klassen wird in den Basistyp `Thread` konvertiert und im Feld `t` gespeichert. Die übrigen Methoden der äußeren Klasse können über die Schnittstelle der Klasse `Thread` mit dem von `t` referenzierten Objekt kommunizieren und müssen nicht „wissen", welchem exakten Typ dieses Objekt angehört.

[78] Die Klassen `InnerRunnable1` und `InnerRunnable2` wiederholen die beiden ersten Beispiele, wobei sie das Interface *`Runnable`* implementieren, statt von der Klasse `Thread` abgeleitet zu sein.

[79] Die Klasse `ThreadMethod` führt das Erzeugen und Starten eines Threads mittels einer Methode vor. Die Methode wird aufgerufen, wenn Sie bereit sind den Thread zu starten und kehrt nach dem Start des Threads zurück. Wenn ein Thread nur eine Hilfsoperation ausführt und keine für die Klasse fundamentale Funktion erfüllt, ist dieser Ansatz eventuell nützlicher und passender, als den Thread im Konstruktor seiner eigenen Klasse zu starten.

Übungsaufgabe 10: (4) Ändern Sie Übungsaufgabe 5 dem Beispiel der Klasse `ThreadMethod` entsprechend. Die `runTask()`-Methode erwartet die Anzahl der zu summierenden Fibonaccizahlen und gibt das beim Aufruf der `submit()`-Methode erzeugte *`Future`*-Objekt zurück. ■

22.2.10 Terminologie

[80] Wie der vorige Unterabschnitt zeigt, haben Sie beim Schreiben threadbasierter Java-Programme Wahlmöglichkeiten. Diese Entscheidungsfreiheit kann Verwirrung stiften. Häufig ergeben sich die Schwierigkeiten bereits aus der Bezeichnungsweise, die zur Beschreibung der Gegenstände und Zusammenhänge auf dem Gebiet der parallelen Programmierung verwendet wird, insbesondere in der

Threadprogrammierung.

[81] An dieser Stelle des Kapitels sollten Sie den Unterschied zwischen der Aufgabe und dem Thread, welcher die Aufgabe verarbeitet, verinnerlicht haben. Dieser Unterschied äußert sich in den Java-Bibliotheken besonders deutlich, indem Sie keinerlei Kontrolle über die Klasse `Thread` ausüben können. (Die Trennung wird im Hinblick auf die Exekutoren noch klarer, die sich an Ihrer Stelle um das Erzeugen und den Betrieb Ihrer Threads kümmern.) Sie beschränken sich darauf, Aufgaben anzulegen und irgendwie mit einem Thread zu verbinden, welcher sich anschließend um die Verarbeitung Ihrer Aufgabe kümmert.

[82] Die Funktion der Java-Klasse `Thread` besteht lediglich darin, die ihr übergebene Aufgabe zu verarbeiten. Dennoch durchzieht der Sprachgebrauch „der Thread führt diese oder jene Aktion aus“ die Fachliteratur. Dadurch entsteht der Eindruck, der Thread selbst sei die Aufgabe. Als ich zum erstenmal auf Java-Threads stieß, war ich so stark von diesem Eindruck geprägt, daß ich von einer „ist ein“-Beziehung zwischen Thread und Aufgabe überzeugt war und glaubte, die Aufgabenklasse von `Thread` ableiten zu müssen. Dazu kommt die unglückliche Bezeichnung des ***`Runnable`*** Interfaces, das meiner Ansicht nach treffender mit „***`Task`***“ bezeichnet worden wäre. Falls ein Interface lediglich eine generische Kapselung seiner Methoden ist, so ist das Namensschema „kann-dieses-oder-jenes-tun-able“ passend. Drückt das Interface aber ein höheres Konzept aus (wie eine zu verarbeitende Aufgabe), so ist es sinnvoller, den Namen des Konzeptes zu verwenden.

[83] Das Problem besteht darin, daß die Abstraktionsebenen nicht sauber getrennt werden. Vom konzeptionellen Standpunkt aus betrachtet, wollen wir eine Aufgabe definieren, die von anderen Aufgaben unabhängig verarbeitet werden kann. Kurz, wir wollen eine Aufgabe beschreiben, die Verarbeitung starten und uns nicht um die Einzelheiten kümmern müssen. Aus der physikalischen Perspektive betrachtet, sind Threads teuer zu erzeugen, sollten also konserviert und verwaltet werden. Hinsichtlich der Implementierung ist es also sinnvoll, Aufgaben und Threads voneinander zu trennen. Darüber hinaus baut die Threadunterstützung von Java auf den systemnahen *pthreads* von C auf, der voraussetzt, daß Sie sich vertieft mit dem Thema auseinandersetzen und die praktischen Grundzüge sorgfältig durchdenken. Ein Teil dieses systemnahen Wesens ist in die Threadunterstützung von Java durchgesickert, so daß Sie beim Programmieren Disziplin walten lassen müssen, um auf einer höheren Abstraktionsebene zu bleiben. (Ich werde versuchen, mich in diesem Kapitel an diese Disziplin zu halten.)

[84] Ich werde versuchen, stets den Begriff „Aufgabe“ (*task*) zu verwenden, wenn ich von der zu verrichtenden Arbeit spreche und den Begriff „Thread“ nur für den Mechanismus zu gebrauchen, welcher die Aufgabe verarbeitet. Wenn Sie aus der konzeptionellen Sicht ein Programm oder eine Anwendung diskutieren, können Sie daher von „Aufgaben“ sprechen, ohne den verarbeitenden Mechanismus überhaupt zu erwähnen.

22.2.11 Die join()-Methode

[85] Ein Thread kann die `join()`-Methode eines anderen Threads aufrufen, um mit seiner eigenen Verarbeitung erst dann fortzufahren, wenn die Verarbeitung des anderen Threads beendet ist. Verarbeitet ein Thread die Anweisung `t.join()`, wobei `t` wiederum ein Thread ist, so wird die Verarbeitung des aufrufenden Threads solange eingestellt, bis die Verarbeitung von `t` abgeschlossen ist (das heißt, wenn `t.isAlive()` `false` liefert).

[86] Alternativ können Sie die `join()`-Methode mit einer Zeitdauer als Argument (entweder in Millisekunden oder in Millisekunden plus Nanosekunden) aufrufen. Der Aufruf der `join()`-Methode wird in diesem Fall beendet, wenn die Verarbeitung des Zielthreads nicht in der gegebenen Zeit beendet wird.

[87] Ein Aufruf der `join()`-Methode kann durch Aufrufen der `interrupt()`-Methode auf dem aufrufenden Thread unterbrochen werden, so daß eine `try`/`catch`-Kombination erforderlich ist.

[88] Alle diese Operationen kommen im folgenden Beispiel vor:

```
//: concurrency/Joining.java
// Understanding join().
import static net.mindview.util.Print.*;

class Sleeper extends Thread {
    private int duration;
    public Sleeper(String name, int sleepTime) {
        super(name);
        duration = sleepTime;
        start();
    }
    public void run() {
        try {
            sleep(duration);
        } catch(InterruptedException e) {
            print(getName() + '' was interrupted. '' +
                  ''isInterrupted(): '' + isInterrupted());
            return;
        }
        print(getName() + '' has awakened'');
    }
}

class Joiner extends Thread {
    private Sleeper sleeper;
    public Joiner(String name, Sleeper sleeper) {
        super(name);
        this.sleeper = sleeper;
        start();
    }
    public void run() {
        try {
            sleeper.join();
        } catch(InterruptedException e) {
            print(''Interrupted'');
        }
        print(getName() + '' join completed'');
    }
}

public class Joining {
    public static void main(String[] args) {
        Sleeper
            sleepy = new Sleeper(''Sleepy'', 1500),
            grumpy = new Sleeper(''Grumpy'', 1500);
        Joiner
            dopey = new Joiner(''Dopey'', sleepy),
            doc = new Joiner(''Doc'', grumpy);
        grumpy.interrupt();
    }
} /* Output:
      Grumpy was interrupted. isInterrupted(): false
      Doc join completed
      Sleepy has awakened
```

```
    Dopey join completed
*///:~
```

Die Klasse `Sleeper` implementiert einen Thread, der für die per Konstruktor übergebene Zeitdauer „schläft“. Die im Körper der `run()`-Methode aufgerufene `sleep()`-Methode kann nach Ablauf der Wartezeit regulär beendet, aber auch unterbrochen werden. Eine Unterbrechung wird durch eine Textmeldung in der `catch`-Klausel dokumentiert, wobei auch der Rückgabewert der `isInterrupted()`-Methode ausgegeben wird. Ruft ein anderer Thread die `interrupt()`-Methode dieses Threads auf, so wird ein Flag gesetzt, um anzuzeigen, daß der Thread unterbrochen wurde. Dieses Flag wird aber beim Abfangen der Ausnahme zurückgesetzt, so daß die `isInterrupted()`-Methode in einer `catch`-Klausel stets `false` zurückgibt. Das Flag wird in Situationen ausgewertet, in denen ein Thread außerhalb des Ausnahmekontextes seinen Unterbrechungszustand auswerten muß.

Die Klasse `Joiner` implementiert eine Aufgabe die nach dem Aufrufen der `join()`-Methode eines `Sleeper`-Threads auf dessen „Erwachen“ wartet. In der `main()`-Methode wird jeder `Joiner`-Thread mit einem `Sleeper`-Thread verknüpft. Die Ausgabe zeigt, daß der `Joiner`-Thread stets zusammen mit dem `Sleeper`-Thread beendet wird, gleichgültig ob der `Sleeper`-Thread unterbrochen oder regulär beendet wird.

[89] Die SE 5 enthält im Package `java.util.concurrent` Hilfsklassen wie `CyclicBarrier` (siehe Abschnitt 22.7), die sich eventuell besser eignen, als die `join()`-Methode, die zur ursprünglichen Thread-Bibliothek gehört.

22.2.12 Reaktionsfähige Benutzerschnittstellen

[90] Das Entwickeln reaktionsfähiger (*responsive*) Benutzerschnittstellen ist eine Motivation, um sich mit dem Thema Threadprogrammierung auseinanderzusetzen. Da wir *graphische* Benutzerschnittstellen erst in Kapitel 23 besprechen, ist das folgende Beispiel nur eine konsolengesteuerte Schnittstellenattrappe. Das Beispiel besteht aus zwei Versionen: `UnresponsiveUI` bleibt in einer Berechnung stecken und ist daher nicht in der Lage, Eingaben von der Konsole zu lesen. `ResponsiveUI` lagert die Berechnung in eine separate Aufgabe aus und kann daher sowohl die Berechnung ausführen, als auch Eingaben von der Konsole entgegen nehmen:

```
//: concurrency/ResponsiveUI.java
// User interface responsiveness.
// {RunByHand}

class UnresponsiveUI {
    private volatile double d = 1;
    public UnresponsiveUI() throws Exception {
        while(d > 0)
            d = d + (Math.PI + Math.E) / d;
        System.in.read(); // Never gets here
    }
}

public class ResponsiveUI extends Thread {
    private static volatile double d = 1;
    public ResponsiveUI() {
        setDaemon(true);
        start();
    }
    public void run() {
        while(true) {
```

```
                d = d + (Math.PI + Math.E) / d;
            }
        }
        public static void main(String[] args) throws Exception {
            //! new UnresponsiveUI(); // Must kill this process
            new ResponsiveUI();
            System.in.read();
            System.out.println(d); // Shows progress
        }
    } ///:~
```

[91] Die Klasse `UnresponsiveUI` führt in einer unendlichen Schleife eine Berechnung aus, kann also die Zeile mit der Konsoleneingabeaufforderung nicht erreichen (der Compiler „glaubt“, daß die Eingabeaufforderung erreichbar ist, weil die `while`-Schleife eine nichttriviale Bedingung hat). Wenn Sie die auskommentierte Zeile, die ein `UnresponsiveUI`-Objekt erzeugt wieder aktivieren, müssen Sie den Prozeß abbrechen, um das Programm zu beenden.

[92] Wenn das Programm reagieren können soll, müssen Sie die Berechnung in eine `run()`-Methode setzen (also als separate Aufgabe formulieren), damit es präemptiv (mit Zeitscheiben) verarbeitet wird. Wenn Sie die Eingabetaste drücken, sehen Sie, daß die Berechnung tatsächlich im Hintergrund fortgesetzt wurde, während das Programm auf Benutzereingaben gewartet hat.

22.2.13 Threadgruppen

[93] Eine Threadgruppe (*thread group*) ist eine Kollektion von Threads. Die Bedeutung von Threadgruppen läßt sich in einer Aussage von Joshua Bloch[8] zusammenfassen, dem Softwarearchitekten, der während seiner Arbeit bei Sun Microsystems für Fehlerkorrekturen, wesentliche Verbesserungen an und Beiträge zu Version 1.2 des Java Development Kits verantwortlich zeichnet:

> „Thread groups are best viewed as an unsuccessful experiment, and you may simply ignore their existence.“
>
> Übersetzt etwa: „Threadgruppen sollten am besten als gescheitertes Experiment angesehen und ihr Vorhandensein einfach nicht beachtet werden.“

Wenn Sie (wie ich) Zeit und Energie investiert haben, um herauszufinden, wozu Threadgruppen eigentlich gut sind, werden Sie sich vielleicht gefragt haben, warum Sun Microsystems nicht deutlicher auf diese Fähigkeit hingewiesen hat. Dieselbe Frage betrifft noch eine Reihe anderer Eigenschaften und Fähigkeiten von Java, die im Laufe der Zeit in den Sprachumfang aufgenommen wurden. Der Nobelpreisträger und Wirtschaftswissenschaftler Joseph Stiglitz vertritt eine Weltanschauung, die zu diesem Kontext paßt.[9] Sie wird als die „Theorie des eskalierenden Commitments“[10] bezeichnet:

[8] Joshua Block: *Effective Java Language Programming Guide*, Addison-Wesley (2001), Seite 211.

[9] Die Theorie des eskalierenden Commitments paßt auch in einer Reihe anderer Stellen zu meiner Erfahrung mit Java. Well, why stop here? Ich ware an nicht wenigen Projekten in beratender Funktion beteiligt, in denen diese Beobachtung zutraf.

[10] Anmerkung des Übersetzers: Mit dem Begriff „eskalierendes Commitment“ beschreibt man ein Phänomen, wonach sich Menschen oft von einem einmal eingeschlagenen Kurs (das heißt einer konkreten Verhaltensweise, einer Verhaltensstrategie, einem Handlungsmuster) nicht abbringen lassen und zwar selbst dann nicht, wenn sich immer deutlicher abzeichnet, dass der eingeschlagene Kurs in die Irre führt. Unter „Commitment“ versteht man die Festlegung auf ein bestimmtes Verhalten. Sich auf ein Verhalten festzulegen ist unumgänglich. Wenn es denn überhaupt möglich wäre, sich auf gar nichts festzulegen (also keinerlei Commitment aufzubringen), könnte man nicht ein einziges Ziel erreichen, ohne Commitment gibt es kein Handeln und damit auch kein Überleben. Commitment ist also ein „normales“ Phänomen. Problematisch wird Commitment allerdings dann, wenn man es nicht auch wieder aufgeben kann, wenn es also gewissermaßen eingefroren ist und sich gegen bessere Einsichten verschließt — oder sich vielleicht sogar verstärkt, also eskaliert, gerade dann, wenn es eigentlich geboten ist, sich von seinem Commitment wieder zu lösen. (Quelle: *http://perso.uni-lueneburg.de/index.php?id=81*)

„The cost of continuing mistakes is borne by others, while the cost of admitting mistakes is borne by yourself.“

Übersetzt etwa: „Den Preis für die Existenz von Fehlern tragen andere. Den Preis, einen Fehler eingestehen zu müssen, tragen Sie alleine.“

22.2.14 Abfangen von Ausnahmen

[94] Eine einem Thread „entwischte“ Ausnahme läßt sich, bedingt durch die Natur von Threads, nicht mehr abfangen. Hat eine Ausnahme den Geltungsbereich der `run()`-Methode einer Aufgabe verlassen, so wird sie auf der Konsole ausgegeben, falls Sie keine Schritte einleiten, um derart fehlgeleitete Ausnahmen abzufangen. Vor der SE 5 wurden solche Ausnahmen mit Hilfe von Threadgruppen abgefangen. Ab Version 5 können Sie das Problem per Exekutor lösen und müssen sich nicht mehr mit Threadgruppen auseinandersetzen (außer, um ältere Quelltexte zu verstehen; siehe „Thinking in Java“, 2nd ed., auf *http://www.mindview.net* zu Einzelheiten über Threadgruppen).

[95] Die folgende Aufgabe wirft stets eine Ausnahme aus, die den Geltungsbereich der `run()`-Methode verläßt. Die `main()`-Methode dient dazu, die Aufgabe per Thread zu verarbeiten und das Ergebnis zu dokumentieren:

```
//: concurrency/ExceptionThread.java
// {ThrowsException}
import java.util.concurrent.*;

public class ExceptionThread implements Runnable {
    public void run() {
        throw new RuntimeException();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} ///:~
```

Ausgabe:

```
Exception in thread ''pool-1-thread-1'' java.lang.RuntimeException
        at ExceptionThread.run(ExceptionThread.java:7)
        at ThreadPoolExecutor$Worker.runTask(ThreadPoolExecutor.java:885)
        at ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:907)
        at java.lang.Thread.run(Thread.java:619)
```

Das Einschließen der Anweisungen im Körper der `main()`-Methode in eine einfache `try`/`catch`-Kombination hilft nicht:

```
//: concurrency/NaiveExceptionHandling.java
// {ThrowsException}
import java.util.concurrent.*;

public class NaiveExceptionHandling {
    public static void main(String[] args) {
        try {
            ExecutorService exec = Executors.newCachedThreadPool();
            exec.execute(new ExceptionThread());
        } catch(RuntimeException ue) {
            // This statement will NOT execute!
            System.out.println(''Exception has been handled!'');
        }
```

```
    }
} ///:~
```

Das Ergebnis ist wie im vorigen Falle eine nicht abgefangene Ausnahme.

[96] Die Lösung des Problems besteht darin, die Art und Weise der Threaderzeugung durch den Exekutor zu beeinflussen. Das statische Interface *`java.lang.Thread.UncaughtExceptionHandler`* ist seit der SE 5 vorhanden und gestattet, ein `Thread`-Objekt mit einem Ereigsnisbehandler zu verknüpfen. Die in diesem Interface deklarierte Methode `uncaughtException()` wird automatisch aufgerufen, wenn ein Thread aufgrund einer nicht abgefangenen Ausnahme zu „sterben“ droht. Die Anwendung dieses Mechanismus erfordert das Schreiben einer neuen Klasse, die das Interface *`ThreadFactory`* implementiert und die jedem neuen `Thread`-Objekt einen individuellen Ausnahmebehandler vom Typ *`Thread.UncaughtExceptionHandler`* zuweist. Wir übergeben der `Executors`-Methode, die das neue *`ExecutorService`*-Objekt erzeugt, eine Referenz auf die Threadfabrik:

```
//: concurrency/CaptureUncaughtException.java
import java.util.concurrent.*;

class ExceptionThread2 implements Runnable {
    public void run() {
        Thread t = Thread.currentThread();
        System.out.println("run() by " + t);
        System.out.println("eh = " + t.getUncaughtExceptionHandler());
        throw new RuntimeException();
    }
}

class MyUncaughtExceptionHandler
    implements Thread.UncaughtExceptionHandler {
    public void uncaughtException(Thread t, Throwable e) {
        System.out.println("caught " + e);
    }
}

class HandlerThreadFactory implements ThreadFactory {
    public Thread newThread(Runnable r) {
        System.out.println(this + " creating new Thread");
        Thread t = new Thread(r);
        System.out.println("created " + t);
        t.setUncaughtExceptionHandler(new MyUncaughtExceptionHandler());
        System.out.println("eh = " + t.getUncaughtExceptionHandler());
        return t;
    }
}

public class CaptureUncaughtException {
    public static void main(String[] args) {
        ExecutorService exec =
            Executors.newCachedThreadPool(new HandlerThreadFactory());
        exec.execute(new ExceptionThread2());
    }
} /* Output: (90% match)
      HandlerThreadFactory@de6ced creating new Thread
      created Thread[Thread-0,5,main]
      eh = MyUncaughtExceptionHandler@1fb8ee3
      run() by Thread[Thread-0,5,main]
      eh = MyUncaughtExceptionHandler@1fb8ee3
      caught java.lang.RuntimeException
   *///:~
```

Einige zusätzliche Meldungen dokumentieren, daß die Threads von der Threadfabrik erzeugt und mit einem Ausnahmebehandler vom Typ *`Thread.UncaughtExceptionHandler`* verknüpft werden. Sie sehen, daß die nicht abgefangenen Ausnahmen nun mit Hilfe der Methode `uncaughtException()` erfaßt werden.

[97] Das obige Beispiel weist jedem Thread einen individuellen Ausnahmebehandler zu. Wenn Sie stets denselben Ausnahmebehandler verwenden, können sie sich die Arbeit erleichtern und einen Ausnahmebehandler für nicht abgefangene Ausnahmen voreinstellen, wodurch ein statische Feld der Klasse `Thread` bewertet wird:

```
//: concurrency/SettingDefaultHandler.java
import java.util.concurrent.*;

public class SettingDefaultHandler {
    public static void main(String[] args) {
        Thread.setDefaultUncaughtExceptionHandler(
            new MyUncaughtExceptionHandler());
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new ExceptionThread());
    }
} /* Output:
      caught java.lang.RuntimeException
   *///:~
```

Dieser Ausnahmebehandler wird nur aufgerufen, wenn kein threadspezifischer Ausnahmebehandler für nicht abgefangene Ausnahmen definiert ist. Das System prüft zunächst, ob ein threadspezifischer Ausnahmebehandler definiert ist und sucht im Negativfall nach einer für die Threadgruppe definierten `uncaughtException()`-Methode. Falls auch diese Methode nicht definiert ist, ruft das System den voreingestellten Ausnahmebehandler für nicht abgefangene Ausnahmen auf.

22.3 Gemeinsam verwendete Resourcen

[98] Sie können sich ein Programm, an dessen Ausführung nur ein einzelner Thread beteiligt ist, als einsame Entität im Raum Ihrer Probleme vorstellen, die stets höchstens einen Arbeitsschritt verarbeitet. Da es nur eine Entität gibt, sind Sie niemals mit dem Problem zweier Entitäten konfrontiert, die gleichzeitig dieselbe Resource benötigen, zum Beispiel zwei Autofahrer, die denselben Parkplatz beanspruchen, zwei Fußgänger, die gleichzeitig durch dieselbe Tür treten oder zwei Menschen, die zur selben Zeit sprechen wollen.

[99] In der Threadprogrammierung gibt es keine separaten Entitäten mehr, sondern es ist möglich, daß sich zwei oder mehr Aufgaben während ihrer Verarbeitung störend beeinflussen. Wenn Sie solche Kollisionen nicht vermeiden, ändern unter Umständen zwei Threads gleichzeitig dasselbe Bankkonto, verwenden denselben Drucker oder bedienen dasselbe Ventil.

22.3.1 Mißbräuchlicher Resourcenzugriff

[100] Im folgenden Beispiel generiert eine Aufgabe gerade Zahlen, die von den anderen Aufgaben „konsumiert“ werden. Die einzige Tätigkeit der konsumierenden Aufgaben besteht darin, die Gültigkeit der geraden Zahlen zu überprüfen.

[101] Wir definieren zunächst die konsumierende Aufgabe `EvenChecker`, die auch in den folgenden Beispielen vorkommt. Wir entkoppeln `EvenChecker` von den verschiedenen Generatortypen mit den wir experimentieren werden, indem wir das Minimum an benötigten Generatormethoden (`next()`

und `cancel()`) in die abstrakte Klasse `IntGenerator` auslagern. `IntGenerator` implementiert das Interface *`Generator<T>` nicht*, da die Klasse `int`-Werte erzeugen muß und generische Typen keine Parameter primitiven Typs unterstützen:

```
//: concurrency/IntGenerator.java
public abstract class IntGenerator {
    private volatile boolean canceled = false;
    public abstract int next();
    // Allow this to be canceled:
    public void cancel() { canceled = true; }
    public boolean isCanceled() { return canceled; }
} ///:~
```

[102] Die Klasse `IntGenerator` besitzt eine `cancel()`-Methode, um das `boolean`-Feld `cancelled` mit `true` bewerten zu können, sowie eine `isCanceled()`-Methode, um den Betriebszustand des Objektes abfragen zu können. Da das `cancelled`-Feld vom Typ `boolean` ist, handelt es sich um ein *atomares Feld*, das heißt einfache Operationen wie die Wertzuweisung oder das Abfragen des Feldwertes können nicht unterbrochen werden. Das Feld befindet sich also niemals während einer solchen Operation in einem undefinierten Zustand. Das `cancelled`-Feld ist außerdem als volatiles Feld (`volatile`) deklariert, um seine Sichtbarkeit zu gewährleisten. Die Begriffe Atomizität und Volatilität werden im Unterabschnitt 22.3.3 erklärt.

[103] Jede konkrete, von `IntGenerator` abgeleitete Klasse kann mit der folgenden `EvenChecker`-Klasse getestet werden:

```
//: concurrency/EvenChecker.java
import java.util.concurrent.*;

public class EvenChecker implements Runnable {
    private IntGenerator generator;
    private final int id;
    public EvenChecker(IntGenerator g, int ident) {
        generator = g;
        id = ident;
    }
    public void run() {
        while(!generator.isCanceled()) {
            int val = generator.next();
            if(val % 2 != 0) {
                System.out.println(val + " not even!");
                generator.cancel(); // Cancels all EvenCheckers
            }
        }
    }
    // Test any type of IntGenerator:
    public static void test(IntGenerator gp, int count) {
        System.out.println("Press Control-C to exit");
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < count; i++)
            exec.execute(new EvenChecker(gp, i));
        exec.shutdown();
    }
    // Default value for count:
    public static void test(IntGenerator gp) {
        test(gp, 10);
    }
} ///:~
```

Beachten Sie, daß die Generatorklasse keine Implementierung des *`Runnable`*-Interfaces ist. Statt dessen fragen alle von einem `IntGenerator`-Objekt abhängigen `EvenChecker`-Aufgaben in ihren `run()`-Methoden den Betriebszustand ihres Generators ab. Auf diese Weise überwachen die Aufgaben, welche sich eine gemeinsame Resource (den Generator) teilen, ihre Resource und achten auf das Signal, um sich zu beenden. Dadurch wird die sogenannte **Wettlaufsituation** (*race condition*) ausgeschaltet, in der zwei oder mehr Aufgaben darum wetteifern, auf eine bestimmte Bedingung zu reagieren und daher kollidieren oder auf anderem Wege inkonsistente Ergebnisse erzeugen. Sie müssen sorgfältig über alle Möglichkeiten nachdenken, auf die ein threadbasiertes Programm scheitern kann und sich davor schützen. Beispielsweise darf eine Aufgabe nicht (ohne weiteres) von einer anderen Aufgabe abhängen, da die Reihenfolge, in der die Aufgabe beendet werden nicht garantiert werden kann.

[104] Die `test()`-Methode initialisiert und verarbeitet einen Test für einen beliebigen von `IntGenerator` abgeleiteten Generatortyp, indem Sie eine Reihe von `EvenChecker`-Aufgaben startet, die denselben Generator verwenden. Erzeugt der Generator einen Fehler (das heißt eine ungerade Zahl), so gibt `test()` eine entsprechende Fehlermeldung aus und kehrt zurück. Andernfalls müssen Sie das Programm über die Tastenkombination Ctrl+C beenden.

[105] Die `EvenChecker`-Aufgaben lesen und testen die von ihrem Generator gelieferten Werte ohne Pause. Gibt die Methode `generator.isCanceled()` `true` zurück, so beendet sich die `run()`-Methode und dem Exekutor in der `test()`-Methode wird mitgeteilt, daß die Verarbeitung der Aufgabe abgeschlossen ist. Jede `EvenChecker`-Aufgabe kann die `cancel()`-Methode ihres Generators aufrufen und dadurch veranlassen, daß alle anderen `EvenChecker`-Aufgaben, die denselben Generator verwenden, in geordneter Weise beendet werden. Sie lernen in Abschnitt 22.4, daß Java allgemeinere Mechanismen zum Beenden von Threads besitzt.

[106] Die `next()`-Methode der ersten von `IntGenerator` abgeleiteten Klasse erzeugt eine Folge ganzer Zahlen:

```
//: concurrency/EvenGenerator.java
// When threads collide.

public class EvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public int next() {
        ++currentEvenValue; // Danger point here!
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new EvenGenerator());
    }
} /* Output: (Sample)
      Press Control-C to exit
      89476993 not even!
      89476993 not even!
   *///:~
```

[107] Es ist möglich, daß eine Aufgabe die `next()`-Methode aufruft, während eine andere Aufgabe das Feld `currentEvenValue` zum ersten, aber noch nicht zum zweiten Mal inkrementiert hat, nämlich an der mit „Danger point here!“ bezeichneten Stelle. Dadurch gerät der generierte Wert in einen „unzulässigen“ Zustand. Die `test()`-Methode erzeugt eine Anzahl von `EvenChecker`-Objekten, welche die vom Generator erzeugten Zahlen liest und prüft, ob es gerade Zahlen sind, um zu beweisen, daß unzulässige Zustände tatsächlich möglich sind. Tritt ein solcher Zustand ein, wird der Fehler gemeldet und das Programm beendet.

[108] Das obige Programm wird letztendlich abgebrochen, weil die `EvenChecker`-Aufgaben die vom Generator erzeugte Information abfragen können, während sie sich in einem unzulässigen Zustand befindet. Das Problem kann sich durchaus erst nach vielen Zyklen zeigen, je nach den Eigenschaften Ihres Betriebssystems und andere Implementierungsdetails. Wenn Sie den Abbruch früher herbeiführen wollen, können Sie es mit einem `yield()`-Aufruf zwischen den beiden Inkrementierungsanweisungen versuchen. Ist die Wahrscheinlichkeit des Scheiterns sehr gering, so kann ein threadbasiertes Programm trotz vorhandener Fehler korrekt erscheinen. Dies ist ein Teil des Problems in der Threadprogrammierung.

[109] Es ist wichtig, darauf hinzuweisen, daß die Inkrementierungsanweisung selbst aus mehreren Operationen besteht und die Verarbeitung einer Aufgabe vom Threadscheduler mitten im Inkrementierungsvorgang ausgesetzt werden kann. Die Inkrementierungsanweisung ist bei Java also keine atomare Operation. Selbst das einfache Inkrementieren eines Wertes ist also ohne entsprechenden Schutz nicht sicher.

22.3.2 Resolving Shared Resource Contention

[110] Das vorige Beispiel zeigt ein fundamentales Problem der Threadprogrammierung: Sie wissen nicht, wann ein bestimmter Thread verarbeitet wird. Stellen Sie sich vor, Sie sitzen mit einer Gabel am Tisch und sind gerade dabei, den letzten Bissen von der Servierplatte aufzuspießen. Während Sie Ihre Gabel auf die Platte zubewegen, verschwindet der Bissen plötzlich, weil die Verarbeitung Ihres Threads ausgesetzt wurde und ein anderer Gast den Rest von der Platte gegessen hat. Damit ein threadbasiertes Programm richtig funktioniert, müssen Sie verhindern, daß zwei Aufgaben auf ein und dieselbe Resource zugreifen, wenigstens in kritischen Situationen.

[111] Derartige Kollisionen lassen sich einfach dadurch vermeiden, daß die Resource während des Zugriffs durch einen Thread gesperrt wird. Die erste Aufgabe, die Zugriff auf die Resource erhält, muß die Resource sperren, so daß kann keine der übrigen Aufgaben auf diese Resource zugreifen kann, bis die Sperre wieder aufgehoben wird, woraufhin ein anderer Thread die Resource sperrt und verwendet und so weiter. Stellen Sie sich den Beifahrersitz im Auto als knappe Resource vor. Das Kind das zuerst „Schrotflinte“ ruft, besetzt den Beifahrersitz (für die Dauer dieser Fahrt).

[112] Das Problem der Threadkollision wird bei fast allen Threadmodellen durch sequentialisierten Zugriff auf die gemeinsam genutzte Resource gelöst, das heißt, daß stets höchstens eine Aufgabe Zugriff auf diese Resource erhält. Dies wird gewöhnlich dadurch bewerkstelligt, daß ein Block von Anweisungen eingeklammert und so gekennzeichnet wird, daß die darin enthaltenen Anweisungen stets von höchstens einem Thread verarbeitet werden können. Da die Einklammerung und Markierung bewirken, daß sich die Threads gegenseitig vom Zugriff auf die Resource ausschließen (*mutual exclusion*), wird diese Vorgehensweise häufig als Mutex-Mechanismus bezeichnet.

[113] Nehmen Sie Ihr Badezimmer als Beispiel: Es gibt mehrere Haushaltsmitglieder (von Threads verarbeitet Aufgaben), die das Badezimmer (gemeinsam verwendete Resource) alleine benutzen möchten. Vor dem Betreten des Badezimmers klopft ein Haushaltsmitglied an die Tür, um zu sehen, ob der Raum frei ist. Ist niemand im Bad, so tritt das Haushaltsmitglied ein und verriegelt (sperrt) die Tür. Jedes andere Haushaltsmitglied, das ebenfalls ins Bad möchte, wird „blockiert“ und wartet vor der Tür, bis der Raum wieder frei wird.

[114] Der Vergleich hinkt an der Stelle, an der das Badezimmer frei wird und das nächste Haushaltsmitglied eintritt. Die Kandidaten stellen sich nicht in einer Reihe auf und es läßt sich nicht voraussagen, wer als nächster an die Reihe kommt, da der Threadscheduler nicht deterministisch arbeitet. Stellen Sie sich eine Anzahl „blockierter Haushaltsmitglieder“ vor, die vor der Badezimmertür auf- und abgehen und wenn derjenige der das Bad gesperrt hat, die Tür entriegelt und

herauskommt, betritt derjenige das Bad, der der Tür am nächsten ist. Sie können dem Threadschedulder per `yield()` oder `setPriority()` Vorschläge machen, deren Berücksichtigung aber vom Betriebssystem und Ihrer Laufzeitumgebung abhängen.

[115] Das Schlüsselwort `synchronized` dient dazu, Kollisionen bei Resourcen zu vermeiden. Steht eine Aufgabe vor der Verarbeitung einiger durch `synchronized` geschützter Zeilen, so prüft die Aufgabe den Sperrstatus, akquiriert die Sperre, führt die Anweisungen aus und hebt die Sperre wieder auf.

[116] Die gemeinsame Resource ist typischerweise ein Speicherbereich in Gestalt eines Objektes, kann aber ebenso eine Datei, ein Port oder ein Drucker sein. Um den Zugriff auf eine gemeinsame Resource kontrollieren zu können, muß sie zuerst in einem Objekt „verpackt“ werden. Anschließend kann jede Methode, die auf die Resource zugreift, als `synchronized` deklariert werden (**synchronisierte Methode**). Hat eine Aufgabe eine synchronisierte Methode aufgerufen, so werden alle anderen Aufgaben beim Versuch, eine synchronisierte Methode aufzurufen blockiert, bis die erste Aufgabe von ihrem Methodenaufruf zurückkehrt.

[117] Sie wissen bereits, daß im produktiven Betrieb die Felder einer Klasse als `private` deklariert werden und der Zugriff auf diese Speicherbereiche nur über Methoden stattfinden soll. Kollisionen lassen sich verhindern, indem Sie diese Methoden synchronisieren:

```
synchronized void f() { /* ... */ }
synchronized void g() { /* ... */ }
```

Jedes Objekt hat eine individuelle Sperre (auch als „Monitor“ bezeichnet). Wenn Sie eine synchronisierte Methode aufrufen, wird das Objekt gesperrt und keine andere synchronisierte Methode dieses Objektes kann aufgerufen werden, bevor die Verarbeitung der ersten Methode beendet und die Sperre aufgehoben ist. Für die beiden obigen Methoden `f()` und `g()` bedeutet das: Ruft eine Aufgabe die Methode `f()` eines Objektes auf, so kann keine andere Aufgabe die Methoden `f()` und `g()` desselben Objektes aufrufen, ehe `f()` beendet ist und die Sperre des Objektes aufgehoben wurde. Es gibt also eine Sperre, die von allen synchronisierten Methoden eines Objektes verwendet wird und diese Sperre kann genutzt werden, um das gleichzeitige Schreiben in die Felder durch mehrere Aufgaben zu verhindern.

[118] Beachten Sie, daß die `private`-Deklaration von Feldern in der Threadprogrammierung besonders wichtig ist. Andernfalls kann auch das Schlüsselwort `synchronized` andere Aufgaben nicht am direkten Zugriff auf ein Feld und am Verursachen von Kollisionen hindern.

[119] Ein Aufgabe kann ein Objekt mehrfach sperren. Dieser Effekt tritt ein, wenn eine sychronisierte Methode eine zweite synchronisierte Methode desselben Objektes aufruft, welche wiederum eine dritte synchronisierte Methode desselben Objektes aufruft und so weiter. Die Laufzeitumgebung „merkt“ sich, wie oft ein Objekt gesperrt wird. Im ungesperrten Zustand des Objektes steht dieser Zähler auf 0. Sperrt eine Aufgabe das Objekt zum erstenmal, so wird der Zähler auf den Wert 1 erhöht. Jedesmal, wenn diesselbe Aufgabe das Objekt ein weiteres mal sperrt, wird der Zähler inkrementiert. Selbstverständlich ist die Mehrfachsperrung nur der Aufgabe erlaubt, welche bereits die erste Sperrung verursacht hat. Jedesmal, wenn die Aufgabe eine synchronisierte Methode verläßt, wird der Zähler dekrementiert, bis der Zählerstand 0 erreicht und die Sperrung des Objektes zugunsten anderer Aufgaben aufgehoben wird.

[120] Auch das Klassenobjekt jeder Klasse besitzt eine Sperre, um gleichzeitigen Zugriff auf statische Felder durch synchronisierte statische Methoden verhindern zu können.

[121] Wann ist Synchronisierung erforderlich? Wenden Sie **Brian Goetz' Synchronisierungsregel**[11] an:

[11] Nach Brian Goetz, einem der Autoren von *Java Concurrency in Practice*, Addison-Wesley (2006). Autoren: Brian

> „If you are writing a variable that might next be read by another thread, or reading a variable that might have last been written by another thread, you must use synchronization, and further, both the reader and the writer must synchronize using the same monitor lock.“
>
> Übersetzt etwa: „Synchronisierung ist erforderlich, wenn Sie ein Feld bewerten, das möglicherweise unmittelbar danach von einem Thread abgefragt wird oder ein Feld abfragen, das möglicherweise unmittelbar zuvor von einem Thread bewertet wurde. Außerdem müssen alle Zugriffsmethoden dieses Feldes bezüglich derselben Sperre synchronisiert sein.“

Hat Ihre Klasse mehr als eine Methode, die auf die kritischen Felder zugreift, so müssen alle relevanten Methoden synchronisiert sein. Wenn Sie nur eine Methode synchronisieren, sind die anderen Methoden nicht an die Sperre gebunden und können ungestraft aufgerufen werden. *Merke:* Jede Methode, die auf eine kritische gemeinsame Resource zugreift, muß synchronisiert werden, andernfalls funktioniert sie nicht richtig.

22.3.2.1 Synchronisierung der Klasse EvenGenerator

[122] Der unerwünschte Threadzugriff im Beispiel *EvenGenerator.java* läßt sich durch Synchronisierung der `next()`-Methode verhindern:

```
//: concurrency/SynchronizedEvenGenerator.java
// Simplifying mutexes with the synchronized keyword.
// {RunByHand}

public class
    SynchronizedEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    public synchronized int next() {
        ++currentEvenValue;
        Thread.yield(); // Cause failure faster
        ++currentEvenValue;
        return currentEvenValue;
    }
    public static void main(String[] args) {
        EvenChecker.test(new SynchronizedEvenGenerator());
    }
} ///:~
```

Der `yield()`-Aufruf zwischen den beiden Inkrementierungsanweisungen verbessert die Chancen für die Kontextumschaltung, während sich das `currentEvenValue`-Feld in einem unzulässigen Zustand befindet. Da aufgrund der Synchronisierung stets höchstens eine Aufgabe den kritischen Abschnitt „betreten“ kann, bewirkt `yield()` hier keinen Fehler. Die Verwendung der `yield()`-Methode begünstigt häufig das Auftreten von Fehlern.

[123] Die erste Aufgabe, die die `next()`-Methode aufruft, sperrt das Objekt und jede weitere Aufgabe, die das Objekt zu sperren versucht, wird blockiert bis die erste Aufgabe die Sperre freigibt. Nun wählt der Threadscheduler eine der Aufgaben aus, die auf die Sperre warten. Auf diese Weise kann stets höchstens eine Aufgabe die geschützten Anweisungen verarbeiten.

Übungsaufgabe 11: (3) Legen Sie eine Klasse mit zwei Feldern und einer Methode an, die diese Felder in einem Mehrschrittverfahren bewertet, so daß sich die Felder während der Verarbeitung der Methode in einem „unzulässigen Zustand“ befinden (welche Zustände zulässig beziehungsweise

Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea.

unzulässig sind obliegt dabei Ihrer Definition). Legen Sie Abfragemethoden für beide Felder an und erzeugen Sie mehrere Threads, um die verschiedenen Methoden aufzurufen. Zeigen Sie, daß es möglich ist, die Feldinhalte im „unzulässigen“ Zustand anzuzeigen. Beheben Sie das Problem mittels Synchronisierung. ■

22.3.2.2 Einsatz expliziter Sperrobjekte

[124] Das seit der SE 5 vorhandene Package `java.util.concurrent.locks` enthält einen expliziten Mutex-Mechanismus. Ein **Sperrobjekt**, das heißt ein Objekt vom Typ *`Lock`*, muß explizit erzeugt, ge- und entsperrt werden, wodurch der Quelltext, verglichen mit der traditionellen Synchronisierung, weniger elegant ausfällt. Das folgende Beispiel zeigt das Beispiel *SynchronizedEvenGenerator.java*, umgeschrieben für die Verwendung expliziter Sperrobjekte:

```
//: concurrency/MutexEvenGenerator.java
// Preventing thread collisions with mutexes.
// {RunByHand}
import java.util.concurrent.locks.*;

public class MutexEvenGenerator extends IntGenerator {
    private int currentEvenValue = 0;
    private Lock lock = new ReentrantLock();
    public int next() {
        lock.lock();
        try {
            ++currentEvenValue;
            Thread.yield(); // Cause failure faster
            ++currentEvenValue;
            return currentEvenValue;
        } finally {
            lock.unlock();
        }
    }
    public static void main(String[] args) {
        EvenChecker.test(new MutexEvenGenerator());
    }
} ///:~
```

Das `lock`-Feld der Klasse `MutexEvenGenerator` referenziert ein Sperrobjekt vom Typ `java.util.concurrent.locks.ReentrantLock` („eintrittsinvariantes[12] Sperrobjekt“). Die Methoden `lock()` und `unlock()` kennzeichnen den kritischen Abschnitt im Körper der `next()`-Methode. Wenn Sie mit expliziten Sperrobjekten arbeiten möchten, ist es wichtig, sich die hier vorgeführte Schreibweise einzuprägen: Unmittelbar nach dem Aufruf der `lock()`-Methode beginnt eine `try`/`finally`-Kombination, wobei `unlock()` in der `finally`-Klausel aufgerufen werden muß. Dies ist die einzige Möglichkeit, um den Aufruf der `unlock()`-Methode zu garantieren, also die Sperre aufzuheben. Beachten Sie, daß die `return`-Anweisung in der `try`-Klausel stehen muß, um zu gewährleisten, daß `unlock()` nicht zu früh aufgerufen wird und die Daten für eine andere Aufgabe erreichbar sind.

[12] Anmerkung des Übersetzers: „Eine Routine beziehungsweise Methode wird als eintrittsinvariant (*reentrant*) oder auch wiedereintrittsfähig bezeichnet, wenn sie so implementiert ist, daß sie von mehreren Prozessen gleichzeitig ausgeführt werden kann. Dabei dürfen sich die gleichzeitig ausgeführten Instanzen nicht in die Quere kommen. Die Ausführung jeder Instanz läuft also gleich ab, egal wie viele andere Instanzen es noch von dieser Methode gibt.

Das Ziel eines Designs für eine eintrittsinvariante Methode ist es, sicherzustellen, dass kein Teil des Programmcodes selbst durch die Methode geändert wird und dass prozesseigene Informationen wie beispielsweise lokale Variablen in getrennten Speicherbereichen gehalten werden.

Eintrittsinvariante Programmkonstrukte sind die Basis für viele Multitasking-Systeme (Threadsicherheit).“ Aus: Wikipedia, Artikel „Eintrittsinvarianz“ (Weitergeleitet von „Reentrant“)

[125] Die `try`/`finally`-Kombination erfordert zwar mehr Quelltext als die synchronisierte Variante, repräsentiert aber auch einen der Vorteile expliziter Sperrobjekte. Wenn bei der synchronisierten Variante etwas schief geht, wird zwar eine Ausnahme ausgeworfen, aber Sie haben keine Gelegenheit zum Aufräumen und Ihr Programm wieder in einen definierten Zustand zu versetzen. Bei expliziten Sperrobjekten können Sie den Zustand Ihres Programms über die `finally`-Klausel pflegen.

[126] In der Regel bedeutet traditionelle Synchronisierung weniger Quelltext und weniger Programmierfehler, so daß Sie üblicherweise nur dann explizite Sperrobjekte verwenden, wenn Sie spezielle Anforderungen haben. Die traditionelle Synchronisierung gestattet beispielsweise keinen Versuch, ein Objekt zu sperren, der auch scheitern darf oder einen zeitlich befristeten Versuch, ein Objekt zu sperren und bei Mißerfolg aufzugeben. In solchen Fällen brauchen Sie die `java.util.concurrent`-Bibliothek:

```
//: concurrency/AttemptLocking.java
// Locks in the concurrent library allow you
// to give up on trying to acquire a lock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;

public class AttemptLocking {
    private ReentrantLock lock = new ReentrantLock();
    public void untimed() {
        boolean captured = lock.tryLock();
        try {
            System.out.println("tryLock(): " + captured);
        } finally {
            if(captured)
                lock.unlock();
        }
    }
    public void timed() {
        boolean captured = false;
        try {
            captured = lock.tryLock(2, TimeUnit.SECONDS);
        } catch(InterruptedException e) {
            throw new RuntimeException(e);
        }
        try {
            System.out.println("tryLock(2, TimeUnit.SECONDS): " + captured);
        } finally {
            if(captured)
                lock.unlock();
        }
    }
    public static void main(String[] args) {
        final AttemptLocking al = new AttemptLocking();
        al.untimed(); // True -- lock is available
        al.timed();   // True -- lock is available
        // Now create a separate task to grab the lock:
        new Thread() {
            { setDaemon(true); }
            public void run() {
                al.lock.lock();
                System.out.println("acquired");
            }
        }.start();
        Thread.yield(); // Give the 2nd task a chance
```

```
            al.untimed();   // False -- lock grabbed by task
            al.timed();     // False -- lock grabbed by task
        }
    } /* Output:
          tryLock(): true
          tryLock(2, TimeUnit.SECONDS): true
          acquired
          tryLock(): false
          tryLock(2, TimeUnit.SECONDS): false
       *///:~
```

[127] Ein Sperrobjekt vom Typ `ReentrantLock` gestattet den Versuch das Objekt zu sperren, der auch erfolglos bleiben darf, so daß Sie, wenn die Sperre bereits akquiriert ist, entscheiden können, etwas anderes zu tun, statt auf die Aufhebung der Sperre zu warten (siehe `untimed()`). Die `timed()`-Methode versucht, das Sperrobjekt zu sperren, gibt aber nach zwei Sekunden auf (beachten Sie die Angabe der Zeiteinheit mit Hilfe des Aufzählungstyps `java.util.concurrent.TimeUnit`). In der `main()`-Methode wird ein neuer Thread definiert (anonyme Klasse) der das Sperrobjekt sperrt, so daß `untimed()` und `timed()` etwas haben, worum sie wetteifern können.

[128] Explizite Sperrobjekte gestatten außerdem feiner granulare Kontrolle über das Sperren und Entsperren als die traditionelle Synchronisierung. Sie ermöglicht beispielsweise spezielle Synchronisierungsstrukturen wie **verschränkte Sperren** (*hand-over-hand locking*, *lock coupling*), die beim Traversieren verketteter Listen auftreten, wobei stets die Sperre des nächsten Knotens akquiriert werden muß, um die Sperre des aktuellen Knotens aufheben zu können.

22.3.3 Atomizität und Volatilität

[129] In Diskussionen zum Thema „Threadprogrammierung mit Java“ taucht immer wieder die falsche Aussage auf, atomare Operationen müßten nicht synchronisiert werden. Eine *atomare Operation* ist eine Operation, welche *vom Threadscheduler nicht unterbrochen werden kann*. Eine begonnene atomare Operation wird vollständig verarbeitet, bevor die nächste Gelegenheit zur Kontextumschaltung besteht. Es ist heikel und gefährlich, sich auf Atomizität (*atomicity*) zu verlassen. Sie sollten nur dann versuchen, Atomizität statt Synchronisierung zu verwenden, wenn Sie ein Experte auf dem Gebiet der Threadprogrammierung sind oder von einem Experten unterstützt werden. Wenn Sie glauben, daß Sie bereit sind, um mit dem Feuer zu spielen, dann unterziehen Sie sich dem *Goetz-Test*[13]

> „If you can write a high-performance JVM for a modern microprocessor, then you are qualified to *think about* whether you can avoid synchronizing.“
>
> Übersetzt etwa: „Wenn Sie in der Lage sind eine hochleistungsfähige Laufzeitumgebung für einen modernen Mikroprozessor zu schreiben, sind Sie hinreichend qualifiziert, um *darüber nachzudenken*, ob Sie die Synchronisierung umgehen können.“[14]

Es ist nützlich, den Begriff „Atomizität“ zu kennen und zu wissen, daß diese Eigenschaft, zusammen mit anderen fortgeschrittenen Dingen verwendet wurde, um einige der schlauer ausgeklügelten Komponenten der `java.util.concurrent`-Bibliothek zu implementieren. Widerstehen Sie aber dem Drang, sich auf diese Eigenschaft zu verlassen (siehe „Brian Goetz' Synchronisierungsregel“, Seite 893).

[13] Nach dem bereits erwähnten Brian Goetz (Fußnote 11 auf Seite 893), einem Experten für Threadprogrammierung, der mich bei der Arbeit an diesem Kapitel unterstützt hat.

[14] Daraus folgt: „Wenn jemand unterstellt, Threadprogrammierung sei einfach und unkompliziert, dann achten Sie darauf, daß diese Person in Ihrem Projekt keine wichtigen Entscheidungen fällt. Ist letzteres bereits eingetreten, dann haben Sie ein Problem.“

[130] Atomizität ist bei „einfachen Operationen“ auf Feldern primitiven Typs gegeben, mit Ausnahme der Typen `long` und `double`. Das Lesen und Schreiben von Feldern primitiven Typs außer `long` und `double` aus dem beziehungsweise in den Arbeitsspeicher ist garantiert eine unteilbare (atomare) Operation. Die Laufzeitumgebung kann allerdings Lese- und Schreiboperationen bei 64-Bit-Feldern (Feldern vom Typ `long` oder `double`) als zwei separate 32-Bit-Operationen ausführen. Dabei besteht die Möglichkeit, daß der Kontext zwischen den beiden Teiloperationen umgeschaltet wird (dieser Effekt wird hin und wieder als „Wortabriß“ (*word tearing*) bezeichnet, da erst ein Teil des Wertes aktualisiert wurde). Durch die Ergänzung der Deklaration eines `long`- oder `double`-Feldes mit dem Schlüsselwort `volatile` läßt sich aber Atomizität bei einfachen Wertzuweisungen und -rückgaben herbeiführen (beachten Sie, daß `volatile` vor der SE 5 nicht richtig funktioniert hat). Einige Laufzeitumgebungen mögen stärkere Garantien bieten, aber Sie sollten sich nicht auf plattformspezifische Eigenschaften oder Fähigkeiten verlassen.

[131] Der Threadscheduler ist folglich nicht in der Lage eine atomare Operation zu unterbrechen. Experten können diese Tatsache nutzen, um sogenannte nicht-blockierende Synchronisation (*lock-free code*) zu implementieren. ~~But even this is an oversimplification.~~ Es kommt zuweilen vor, daß sich eine scheinbar sichere atomare Operation, ~~it may not be~~. Ein typischer Leser dieses Buches wird den oben beschriebenen Goetz-Test nicht bestehen, ist also nicht für den Versuch qualifiziert, Synchronisierung durch atomare Operationen zu ersetzen. Der Versuch auf Synchronisierung zu verzichten ist in der Regel ein Indikator für voreilige Optimierung und bringt eine Menge Probleme aber kaum (wenn überhaupt) Vorteile mit sich.

[132] Bei Mehrprozessorsystemen (die gegenwärtig in der Gestalt von Mehrkernprozessoren, also mehreren Prozessoren auf einem einzelnen Chip auf dem Markt kommen) ist die *Sichtbarkeit von Feldern* verglichen mit der Atomizität ein erheblich wichtigeres Thema als bei Einprozessorsystemem. Die von einer Aufgabe bewirkten Änderungen sind, auch wenn sie von atomaren Operationen (also ohne Unterbrechungen) verursacht wurden, für andere Aufgaben nicht sichtbar (die Änderungen könnten beispielsweise in einem lokalen Zwischenspeicher des Prozessor liegen), so daß verschiedene Aufgaben die Anwendung in unterschiedlichen Zuständen „wahrnehmen“. Der Synchronisierungsmechanismus erzwingt dagegen bei Mehrprozessorsystemen, daß durch eine Aufgabe bewirkte Änderungen anwendungsübergreifend sichtbar sind. Ohne Synchronisierung ist unbestimmt, wann die Änderungen sichtbar werden.

[133] Auch das Schlüsselwort `volatile` gewährleistet anwendungsübergreifende Sichtbarkeit. Ist ein Feld als `volatile` deklariert, so ist ein geänderter Wert unmittelbar nach der Änderung für alle Leseoperationen sichtbar. Dies gilt sogar, wenn ein lokaler Zwischenspeicher beteiligt ist. Volatile Felder werden unmittelbar in den Hauptarbeitsspeicher fortgeschrieben und von dort gelesen.

[134] Es ist wichtig, zu verstehen, daß Atomizität und Volatilität verschiedene Konzepte sind. Eine atomare Operation auf einem nicht-volatilen Feld wird nicht notwendig in den Hauptarbeitsspeicher fortgeschrieben, so daß Aufgaben, die dieses Feld abfragen nicht notwendig den neuen Wert „sehen“. Greifen mehrere Aufgaben auf ein Feld zu, so sollte es als volatil deklariert werden. Andernfalls sollte der Zugriff auf dieses Feld nur unter Synchronisierung möglich sein. Auch die Synchronsierung verursacht Fortschreibung in den Hauptarbeitsspeicher. Geschieht der Zugriff auf ein Feld ausnahmslos über synchronisierte Methoden oder Blöcke, so ist es nicht erforderlich, das Feld als volatil zu deklieren.

[135] Alle Schreiboperationen einer Aufgabe sind für die Aufgabe selbst sichtbar, das heißt es ist nicht erforderlich ein Feld als volatil zu deklarieren, wenn es nur innerhalb einer Aufgabe verwendet wird.

[136] Die Volatilität wirkt weder, wenn der Wert eines Feldes von seinem vorigen Wert abhängt zum Beispiel Inkrementieren eines Zählers, noch wenn der Wert eines Feldes durch Werte anderer Felder

beschränkt wird, zum Beispiel such as the lower and upper bound of a Range class which must obey the constraint lower <= upper.

[137] Die Ersetzung von `synchronized` durch `volatile` ist typischerweise nur dann sicher, wenn die Klasse nur ein einziges veränderliches Feld hat. Wiederum sollte die Synchronisierung Ihre erste Wahl sein. Es ist die sicherste Vorgehensweise und es ist riskant, etwas anderes zu versuchen.

[138] Welche Operationen sind atomar? Die Wertzuweisung und -rückgabe sind in der Regel atomar. Bei C++ allerdings *können* atomar sein:

```
++;   // Might be atomic in C++
i +=2; // Might be atomic in C++
```

Ob diese Operationen bei C++ atomar sind, hängt von Compiler und Prozessor ab. Sie können im Hinblick auf Atomizität nicht plattformunabhängig programmieren, da C++ im Gegensatz zu Java (in den Java Standard Editions) kein konsistentes Speichermodell hat.[15]

[139] Bei Java sind die obigen Operation definitiv nicht atomar, wie der aus dem folgenden Beispiel erzeugte Bytecode beweist:

```
//: concurrency/Atomicity.java
// {Exec: javap -c Atomicity}

public class Atomicity {
    int i;
    void f1() { i++; }
    void f2() { i += 3; }
} /* Output: (Sample)
     ...
     void f1();
       Code:
        0:        aload_0
        1:        dup
        2:        getfield        #2; // Field i:I
        5:        iconst_1
        6:        iadd
        7:        putfield        #2; // Field i:I
        10:       return

     void f2();
       Code:
        0:        aload_0
        1:        dup
        2:        getfield        #2; // Field i:I
        5:        iconst_3
        6:        iadd
        7:        putfield        #2; // Field i:I
        10:       return
 *///:~
```

Jede der beiden Operationen zerfällt in eine `get`- sowie eine `put`-Instruktion mit weiteren Instruktionen dazwischen. Somit kann das Feld zwischen `get` und `put` von einer anderen Aufgabe verändert werden und die Operationen sind nicht atomar.

[140] Wenn Sie blindly apply the idea of atomicity, sehen Sie, daß die `getValue()`-Methode im folgenden Beispiel der Beschreibung entspricht:

[15] Diese Lücke wurde im neuen C++ Standard geschlossen.

```
//: concurrency/AtomicityTest.java
import java.util.concurrent.*;

public class AtomicityTest implements Runnable {
    private int i = 0;
    public int getValue() { return i; }
    private synchronized void evenIncrement() { i++; i++; }
    public void run() {
        while(true)
            evenIncrement();
    }
    public static void main(String[] args) {
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicityTest at = new AtomicityTest();
        exec.execute(at);
        while(true) {
            int val = at.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
} /* Output: (Sample)
      191583767
   *///:~
```

Aber das Programm findet ungerade Zahlen und beendet sich. Zwar ist `return i` tatsächlich eine atomare Operation, aber die fehlende Synchronisierung der `getValue()`-Methode ermöglicht, daß der Wert abgefragt wird, während sich das Objekt in einem instabilen Übergangszustand befindet. Überdies ist das Feld `i` kein volatiles Feld, wodurch sich Sichtbarkeitsprobleme ergeben können. Sowohl `getValue()` als auch `evenIncrement()` müssen synchronisiert werden. Nur Experten auf dem Gebiet der Threadprogrammierung sind erfahren genug, um in Situationen wie diesen Optimierungen zu versuchen. Denken Sie wiederum an „Brian Goetz' Synchronisierungsregel" (Seite 893).

[141] Ein zweites, sogar noch einfacheres Beispiel: Die folgende Klasse erzeugt Seriennummern.[16] Die `nextSerialNumber()`-Methode muß bei jedem Aufruf einen eindeutigen Wert an den Aufrufer zurückgegeben:

```
//: concurrency/SerialNumberGenerator.java
public class SerialNumberGenerator {
    private static volatile int serialNumber = 0;
    public static int nextSerialNumber() {
        return serialNumber++; // Not thread-safe
    }
} ///:~
```

[142] Die Klasse `SerialNumberGenerator` ist so einfach, wie eine Klasse nur sein kann und wenn Sie von C++ kommen oder einen anderen systemnahen Hintergrund haben, könnten Sie erwarten, daß die Inkrementierung eine atomare Operation ist, da die Inkrementierung bei C++ häufig als Mikroprozessorinstruktion implementiert ist (allerdings nicht in einer verläßlichen, plattformübergreifenden Weise). Wie wir gezeigt haben, ist die Inkrementierungsanweisung bei Java keine atomare Operation und beinhaltet eine Lese- und eine Schreibinstruktion, das heißt selbst bei so einfachen Operationen ist Raum für Threadprobleme. As you shall see, dreht sich dieses Problem

[16] Angeregt durch Joshua Blochs *Effective Java Language Programming Guide*, Addison-Wesley (2001), Seite 190.

nicht um Volatilität. Das tatsächliche Problem besteht darin, daß die Methode `nextSerialNumber()` unsynchronisiert auf ein gemeinsam verwendetes, veränderliches Feld zugreift.

[143] Das `serialNumber`-Feld ist ein volatiles Feld, da jeder Thread einen lokalen Stack haben kann und dort Kopien einiger Felder speichert. Indem Sie ein Feld als volatil deklarieren, weisen Sie den Compiler an, keine Optimierungen auszuführen, welche Lese- und Schreibinstruktionen für die Synchronisierung mit den lokalen Daten des Threads entfernen. Lese- und Schreibinstruktionen gelangen effektiv ohne Zwischenspeicherung in den Hauptspeicher. Die `volatile`-Deklaration also restricts compiler reordering of accesses während der Optimierung. Die `volatile`-Deklaration hat allerdings keinen Einfluß auf die Tatsache, daß die Inkrementierung keine atomare Operation ist.

[144] Im Grunde genommen, sollten Sie ein Feld als volatil deklarieren, wenn es von mehreren Aufgaben zugleich beansprucht wird und wenigstens einer dieser Zugriffe ein Schreibzugriff ist. Beispielsweise muß ein Feld, welches als Flag dient, um eine Aufgabe zu beenden, als volatiles Feld deklariert werden. Andernfalls könnte das Flag in einem Register zwischengespeichert werden, bei einer von außerhalb der Aufgabe verursachten Umschaltung des Flags würde der zwischengespeicherte Wert nicht aktualisiert werden und die Aufgabe nicht „erfahren“, daß sie sich beenden soll.

[145] Um die Klasse `SerialNumberGenerator` testen zu können, benötigen wir a set that doesn't run out of memory, falls wir lange warten müssen, um ein Problem festzustellen. Die Klasse `CircularSet` wiederverwendet den für die Verwaltung der `int`-Wert reservierten Speicher. Wir nehmen an, daß die Wahrscheinlichkeit, beim Überschlag eine Kollision mit einem der überschriebenen Werte hervorzurufen minimal ist. Die Methode `add()` und `contains()` sind sychronisiert, um Threadkollisionen zu vermeiden:

```
//: concurrency/SerialNumberChecker.java
// Operations that may seem safe are not,
// when threads are present.
// {Args: 4}
import java.util.concurrent.*;

// Reuses storage so we don't run out of memory:
class CircularSet {
    private int[] array;
    private int len;
    private int index = 0;
    public CircularSet(int size) {
        array = new int[size];
        len = size;
        // Initialize to a value not produced
        // by the SerialNumberGenerator:
        for(int i = 0; i < size; i++)
            array[i] = -1;
    }
    public synchronized void add(int i) {
        array[index] = i;
        // Wrap index and write over old elements:
        index = ++index % len;
    }
    public synchronized boolean contains(int val) {
        for(int i = 0; i < len; i++)
            if(array[i] == val) return true;
        return false;
    }
}

public class SerialNumberChecker {
```

```
    private static final int SIZE = 10;
    private static CircularSet serials =
        new CircularSet(1000);
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    static class SerialChecker implements Runnable {
        public void run() {
            while(true) {
                int serial = SerialNumberGenerator.nextSerialNumber();
                if(serials.contains(serial)) {
                    System.out.println("Duplicate: " + serial);
                    System.exit(0);
                }
                serials.add(serial);
            }
        }
    }
    public static void main(String[] args) throws Exception {
        for(int i = 0; i < SIZE; i++)
            exec.execute(new SerialChecker());
        // Stop after n seconds if there's an argument:
        if(args.length > 0) {
            TimeUnit.SECONDS.sleep(new Integer(args[0]));
            System.out.println("No duplicates detected");
            System.exit(0);
        }
    }
} /* Output: (Sample)
      Duplicate: 8468656
   *///:~
```

[146] Die Klasse `SerialNumberChecker` referenziert über ein statisches Feld ein Objekt der Klasse `CircularSet`, das die erzeugten Seriennummer enthält sowie eine statische innere Klasse namens `SerialChecker`, welche die Eindeutigkeit der Seriennummer garantiert. Bei einer Reihe von Aufgaben, die Serienummer in das `CircularSet`-Objekt eintragen, können Sie beobachten, daß letztendlich eine der Aufgaben eine doppelte Seriennummer vom Generator erhält, wenn Sie lange genug warten. Das Problem läßt sich lösen, indem Sie die `nextSerialNumber()`-Methode synchronisieren.

[147] Das Abfragen und das Zuweisen von Werten von beziehungsweise an Felder primitiven Typs darf sicher als atomare Operation betrachtet werden. Es ist dennoch möglich, ein Objekt während des Zugriffs über eine atomare Operation in einem instabilen intermediären Zustand vorzufinden (siehe Beispiel *AtomicityTest.java*). Es ist heikel und nicht ungefährlich, bei diesem Thema Annahmen zu treffen. Das sinnvollste ist, sich nach *Brian Goetz' Synchronisierungsregel* zu richten.

Übungsaufgabe 12: (3) Reparieren Sie das Beispiel *Atomicity.java* mit Hilfe des `synchronized`-Schlüsselwortes. Können Sie zeigen, daß das Programm jetzt richtig funktioniert? ■

Übungsaufgabe 13: (1) Reparieren Sie das Beispiel *SerialNumberChecker.java* mit Hilfe des `synchronized`-Schlüsselwortes. Können Sie zeigen, daß das Programm jetzt richtig funktioniert? ■

22.3.4 Atomare Klassen

[148] Die seit der SE 5 vorhandenen atomaren Klassen `AtomicInteger`, `AtomicLong`, `AtomicReference` und so weiter (alle im Package `java.util.concurrent.atomic`) gestatten atomare bedingte Aktualisierungsoperationen der Form

```
boolean compareAndSet(expectedValue, updateValue);
```

Die atomaren Klassen ermöglichen Feineinstellungen unter Nutzung der bei manchen modernen Prozessoren verfügbaren Atomizität auf Maschinenebene, das heißt Sie müssen sich (im Augenblick) noch nicht mit diesem Thema auseinandersetzen. Die atomaren Klassen sind gelegentlich auch in der regulären Programmierarbeit nützlich, allerdings auch hier nur dann, wenn es um Feineinstellungen geht. Das folgende Beispiel zeigt das Beispiel *AtomicityTest.java* von Seite 900, umgeschrieben um die Klasse `AtomicInteger` vorführen zu können:

```
//: concurrency/AtomicIntegerTest.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

public class AtomicIntegerTest implements Runnable {
    private AtomicInteger i = new AtomicInteger(0);
    public int getValue() { return i.get(); }
    private void evenIncrement() { i.addAndGet(2); }
    public void run() {
        while(true)
            evenIncrement();
    }
    public static void main(String[] args) {
        new Timer().schedule(new TimerTask() {
                public void run() {
                    System.err.println("Aborting");
                    System.exit(0);
                }
            }, 5000); // Terminate after 5 seconds
        ExecutorService exec = Executors.newCachedThreadPool();
        AtomicIntegerTest ait = new AtomicIntegerTest();
        exec.execute(ait);
        while(true) {
            int val = ait.getValue();
            if(val % 2 != 0) {
                System.out.println(val);
                System.exit(0);
            }
        }
    }
} ///:~
```

Die Synchronisierung wird durch die atomare Klasse `AtomicInteger` ersetzt. Nachdem das Programm nicht scheitert, ist die Laufzeit programmatisch auf fünf Sekunden begrenzt.

[149] Das nächste Beispiel zeigt das Beispiel *MutexEvenGenerator.java* von Seite 895, umgeschrieben zur Verwendung der Klasse `AtomicInteger`:

```
//: concurrency/AtomicEvenGenerator.java
// Atomic classes are occasionally useful in regular code.
// {RunByHand}
import java.util.concurrent.atomic.*;

public class AtomicEvenGenerator extends IntGenerator {
    private AtomicInteger currentEvenValue =
        new AtomicInteger(0);
    public int next() {
        return currentEvenValue.addAndGet(2);
```

```
    }
    public static void main(String[] args) {
        EvenChecker.test(new AtomicEvenGenerator());
    }
} ///:~
```

Wiederum wurde die Synchronisierung zugunsten der atomaren Klasse `AtomicInteger` ersetzt.

[150] Es soll betont werden, daß die atomaren Klassen entworfen wurden, um die Klassen im Package `java.util.concurrent` schreiben zu können. Sie sollten diese Klassen selbst nur unter bestimmten Voraussetzungen verwenden und auch dann nur, wenn Sie garantieren können, daß sich keine anderweitigen Probleme ergeben können. Es ist generell sicherer, sich auf einen Sperrmechanismus zu verlassen (entweder Synchronisierung oder explizite Sperrobjekte).

Übungsaufgabe 14: (4) Zeigen Sie, daß die Klasse `java.util.Timer` scales to large numbers, indem Sie ein Programm schreiben, welches viele `Timer`-Objekt erzeugt, die zum Zeitpunkt der Zeitüberschreitung eine einfache Aufgabe verarbeitet. ■

22.3.5 Synchronisierte Blöcke von Anweisungen (Kritische Abschnitte)

[151] Hin und wieder genügt es, statt einer ganzen Methoden nur einen Teil der Anweisungen vor dem gleichzeitigen Zugriff durch mehrere Threads zu schützen. Die isolierten Anweisungen werden als *kritischer Abschnitt* (*critical section*) bezeichnet. Ein kritischer Abschnitt wird mit Hilfe des Schlüsselwortes `synchronized` ausgezeichnet. Bei der Synchronisierung eines Blocks von Anweisungen muß ein Objekt angegeben werden, dessen Sperre verwendet wird, um den Zugriff durch mehrere Threads zeitlich abzustimmen:

```
synchronized(syncObject) {
    // This code can be accessed
    // by only one task at a time
}
```

Kritische Abschnitte werden auch als synchronisierte Blöcke bezeichnet. Vor dem Eintritt in einen solchen Block muß das von `syncObject` referenzierte Objekt gesperrt werden. Hat bereits eine andere Aufgabe dieses Objekt gesperrt, so kann der kritische Abschnitt vor der Aufhebung der Sperre nicht „betreten“ werden.

[152] Das folgende Beispiel vergleicht die Synchronisierung einer ganzen Methode mit der Synchronisierung eines kritischen Abschnitts hinsichtlich der für die übrigen Aufgaben verbleibenden Zugriffszeit auf das Objekt, welches die kritischen Anweisungen enthält. Es zeigt sich, daß die für die restlichen Aufgaben verbleibende Zugriffszeit bei der Beschränkung auf einen synchronisierten Block signifikant höher ausfällt, als bei einer synchronisierten Methode. Das Beispiel führt darüber hinaus vor, wie der Zugriff auf eine ungeschützte Klasse durch mehrere Threads mittels einer anderen Klasse kontrolliert und geschützt werden kann:

```
//: concurrency/CriticalSection.java
// Synchronizing blocks instead of entire methods. Also
// demonstrates protection of a non-thread-safe class
// with a thread-safe one.
package concurrency;
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;

class Pair { // Not thread-safe
```

```
    private int x, y;
    public Pair(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Pair() { this(0, 0); }
    public int getX() { return x; }
    public int getY() { return y; }
    public void incrementX() { x++; }
    public void incrementY() { y++; }
    public String toString() {
        return ”x: ” + x + ”, y: ” + y;
    }
    public class PairValuesNotEqualException
        extends RuntimeException {
        public PairValuesNotEqualException() {
            super(”Pair values not equal: ” + Pair.this);
        }
    }
    // Arbitrary invariant - both variables must be equal:
    public void checkState() {
        if(x != y)
            throw new PairValuesNotEqualException();
    }
}

// Protect a Pair inside a thread-safe class:
abstract class PairManager {
    AtomicInteger checkCounter = new AtomicInteger(0);
    protected Pair p = new Pair();
    private List<Pair> storage =
        Collections.synchronizedList(new ArrayList<Pair>());
    public synchronized Pair getPair() {
        // Make a copy to keep the original safe:
        return new Pair(p.getX(), p.getY());
    }
    // Assume this is a time consuming operation
    protected void store(Pair p) {
        storage.add(p);
        try {
            TimeUnit.MILLISECONDS.sleep(50);
        } catch(InterruptedException ignore) {}
    }
    public abstract void increment();
}

// Synchronize the entire method:
class PairManager1 extends PairManager {
    public synchronized void increment() {
        p.incrementX();
        p.incrementY();
        store(getPair());
    }
}

// Use a critical section:
class PairManager2 extends PairManager {
    public void increment() {
        Pair temp;
```

```
        synchronized(this) {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        }
        store(temp);
    }
}

class PairManipulator implements Runnable {
    private PairManager pm;
    public PairManipulator(PairManager pm) {
        this.pm = pm;
    }
    public void run() {
        while(true)
            pm.increment();
    }
    public String toString() {
        return ''Pair: '' + pm.getPair() +
            '' checkCounter = '' + pm.checkCounter.get();
    }
}

class PairChecker implements Runnable {
    private PairManager pm;
    public PairChecker(PairManager pm) {
        this.pm = pm;
    }
    public void run() {
        while(true) {
            pm.checkCounter.incrementAndGet();
            pm.getPair().checkState();
        }
    }
}

public class CriticalSection {
    // Test the two different approaches:
    static void
        testApproaches(PairManager pman1, PairManager pman2) {
        ExecutorService exec = Executors.newCachedThreadPool();
        PairManipulator
            pm1 = new PairManipulator(pman1),
            pm2 = new PairManipulator(pman2);
        PairChecker
            pcheck1 = new PairChecker(pman1),
            pcheck2 = new PairChecker(pman2);
        exec.execute(pm1);
        exec.execute(pm2);
        exec.execute(pcheck1);
        exec.execute(pcheck2);
        try {
            TimeUnit.MILLISECONDS.sleep(500);
        } catch(InterruptedException e) {
            System.out.println(''Sleep interrupted'');
        }
        System.out.println(''pm1: '' + pm1 + ''\npm2: '' + pm2);
        System.exit(0);
```

```
    }
    public static void main(String[] args) {
        PairManager
            pman1 = new PairManager1(),
            pman2 = new PairManager2();
        testApproaches(pman1, pman2);
    }
} /* Output: (Sample)
      pm1: Pair: x: 15, y: 15 checkCounter = 272565
      pm2: Pair: x: 16, y: 16 checkCounter = 3956974
   *///:~
```

[153] Die Klasse `Pair` ist nicht threadsicher, da die (willkürlich gewählte) Randbedingung verlangt, daß die beide `Pair`-Komponenten `x` und `y` übereinstimmen müssen. Außerdem sind die Inkrementierungsoperationen in den Methoden `incrementX()` und `incrementY()` ebenfalls nicht threadsicher. Da keine Methode synchronisiert ist, können Sie sich nicht darauf verlassen, daß ein `Pair`-Objekt beim Zugriff durch mehrere Threads nicht instabil wird.

[154] Stellen Sie sich vor, ein Kollege gibt Ihnen die obige nicht threadsichere Klasse `Pair` und Sie müssen Sie in einer Umgebung einsetzen, in der `Pair`-Objekte dem Zugriff durch mehrere Threads ausgesetzt sind. Sie können die Klasse `Pair` in der beschriebenen Situation dennoch verwenden, indem Sie die Hilfsklasse `PairManager` schreiben, die den Zugriff auf `Pair` kontrolliert. Beachten Sie, daß `PairManager` nur zwei öffentliche Methoden definiert, nämlich die bereits synchronisierte `getPair()`-Methode und die abstrakte `increment()`-Methode (Synchronisierung bei der Implementierung).

[155] Die Struktur der abstrakten Klasse `PairManager`, also das Implementieren von Funktionalität in der Basisklasse unter Verwendung einer oder mehrerer abstrakter Methoden, die in den abgeleiteten Klassen ausprogrammiert werden, wird in der im Entwurfsmusterkontext üblichen Sprechweise[17] als *Templatemethod* bezeichnet. Entwurfsmuster gestatten die Kapselung von Änderungen in Ihrem Quelltext (in diesem Beispiel ist die `increment()`-Methode der sich ändernde Teil). In der abgeleiteten Klasse `PairManager1` ist die ganze `increment()`-Methode synchronisiert, in der abgeleiteten Klasse `PairManager2` dagegen ist nur ein Teil der Anweisungen in einem synchronisierten Block zusammengefaßt. Beachten Sie, daß das Schlüsselwort `synchronized` nicht Bestandteil der Methodensignatur ist und daher beim Überschreiben ergänzt werden darf.

[156] Die Methode `store()` speichert ein `Pair`-Objekt in einem synchronisierten `ArrayList<Pair>`-Objekt, das heißt diese Operation ist threadsicher, muß nicht überwacht werden und steht daher außerhalb des synchronisierten Blocks in `PairManager2`.

[157] Die Klasse `PairManipulator` dient dazu, zwei verschiedene von `PairManager` abgeleitete Klassen zu testen, indem eine Aufgabe die `increment()`-Methode aufruft, während eine andere Aufgabe ein `PairChecker`-Objekt auf das entsprechende `PairManager`-Objekt anwendet. Das `PairChecker`-Objekt inkrementiert bei jeder Prüfung den Zähler `checkCounter`. In der `main()`-Methode werden zwei `PairManipulator`-Objekte (Aufgaben) erzeugt und eine Zeit lang verarbeitet, bevor das Ergebnis angezeigt wird.

[158] Obwohl sich die Ausgabe von Aufruf zu Aufruf des Programms deutlich unterscheiden kann, wird sich zeigen, daß `PairManager1.increment()` erheblich weniger Prüfungen ermöglicht als `PairManager2.increment()`, da die letztere Methode nur einen synchronisierten Block verwendet und daher weniger Sperrzeit beansprucht. Typischerweise werden synchronisierte Blöcke anstelle synchronisierter Methoden verwendet, um den übrigen Aufgaben mehr Zugriffszeit zu gewähren (so-

[17] Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995)

lange die gemeinsam verwendete Resource in konsistentem Zustand bleibt).

[159] Selbstverständlich können Sie explizite Sperrobjekte verwenden, um kritische Abschnitte zu definieren:

```
//: concurrency/ExplicitCriticalSection.java
// Using explicit Lock objects to create critical sections.
package concurrency;
import java.util.concurrent.locks.*;

// Synchronize the entire method:
class ExplicitPairManager1 extends PairManager {
    private Lock lock = new ReentrantLock();
    public synchronized void increment() {
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            store(getPair());
        } finally {
            lock.unlock();
        }
    }
}

// Use a critical section:
class ExplicitPairManager2 extends PairManager {
    private Lock lock = new ReentrantLock();
    public void increment() {
        Pair temp;
        lock.lock();
        try {
            p.incrementX();
            p.incrementY();
            temp = getPair();
        } finally {
            lock.unlock();
        }
        store(temp);
    }
}

public class ExplicitCriticalSection {
    public static void main(String[] args) throws Exception {
        PairManager
            pman1 = new ExplicitPairManager1(),
            pman2 = new ExplicitPairManager2();
        CriticalSection.testApproaches(pman1, pman2);
    }
} /* Output: (Sample)
      pm1: Pair: x: 15, y: 15 checkCounter = 174035
      pm2: Pair: x: 16, y: 16 checkCounter = 2608588
   *///:~
```

Dieses Beispiel wiederverwendet das Beispiel *CriticalSection.java* größtenteils und leitet lediglich zwei neue Klassen von `PairManager` ab, die explizite Sperrobjekte verwenden. `ExplicitPairManager2` zeigt die Kennzeichnung eines kritischen Abschnitts per Sperrobjekt. Die `store()`-Methode wird wiederum außerhalb des kritischen Abschnitts aufgerufen.

22.3.6 Synchronisierung bezüglich eines beliebigen Objektes

[160] Die Synchronisierung eines Blocks von Anweisungen bezieht sich stets auf ein Objekt. In der Regel ist dies das aktuelle Objekt zu dem der aufgerufene kritische Abschnitt gehört: `synchronized(this)` (siehe Klasse `PairManager2`). Ist das Objekt zu dem der synchronisierte Block gehört gesperrt, kann keine synchronisierte Methode und kein kritischer Abschnitt dieses Objektes aufgerufen werden. Die Wirkung eines bezüglich `this` synchronisierten kritischen Abschnitts besteht also lediglich darin, den Geltungsbereich der Synchronisierung zu verkleinern.

[161] Gelegentlich müssen Sie zur Synchronisierung ein anderes Bezugsobjekt als `this` verwenden. In diesem Fall müssen Sie darauf achten, daß sich die Synchronisierung aller relevanten Methoden beziehungsweise kritischen Abschnitte auf dasselbe Objekt bezieht. Das folgende Beispiel zeigt, daß zwei Aufgaben verschiedene Methoden eines Objektes aufrufen können, wenn sich die Synchronisierung dieser Methoden auf verschiedene Objekte bezieht:

```
//: concurrency/SyncObject.java
// Synchronizing on another object.
import static net.mindview.util.Print.*;

class DualSynch {
    private Object syncObject = new Object();
    public synchronized void f() {
        for(int i = 0; i < 5; i++) {
            print("f()");
            Thread.yield();
        }
    }
    public void g() {
        synchronized(syncObject) {
            for(int i = 0; i < 5; i++) {
                print("g()");
                Thread.yield();
            }
        }
    }
}

public class SyncObject {
    public static void main(String[] args) {
        final DualSynch ds = new DualSynch();
        new Thread() {
            public void run() {
                ds.f();
            }
        }.start();
        ds.g();
    }
} /* Output: (Sample)
      g()
      f()
      g()
      f()
      g()
      f()
      g()
      f()
      g()
```

```
        f()
    *///:~
```

Die Synchronisierung der Methode `f()` bezieht sich auf `this` (Synchronisierung der ganzen Methode), die Synchronisierung des kritischen Abschnitts in der Methode `g()` dagegen auf das von `syncObject` referenzierte Objekt. Beide Synchronisierungen sind also voneinander unabhängig. In der `main()`-Methode wird ein Thread erzeugt, der `f()` aufruft, während der `main`-Thread selbst `g()` aufruft. Die Ausgabe zeigt, daß beide Methoden parallel ausgeführt werden, das heißt keine Methode infolge der Synchronisierung der anderen blockiert wird.

Übungsaufgabe 15: (1) Schreiben Sie eine Klasse mit drei Methoden. Jede Methode enthält einen kritischen Abschnitt der bezüglich ein und desselben Objektes synchronisiert ist. Erzeugen Sie mehrere Aufgaben, um zu zeigen, daß stets höchstens eine der drei Methoden aufgerufen werden kann. Ändern Sie nun die Methoden so, daß sie bezüglich verschiedener Objekte synchronisiert werden und zeigen Sie, daß nun alle drei Methoden parallel verarbeitet werden können. ■

Übungsaufgabe 16: (1) Ändern Sie Übungsaufgabe 15 so, daß explizite Sperrobjekte verwendet werden. ■

22.3.7 Thread-lokale Felder

[162] Eine zweite Möglichkeit, um Aufgaben an der Kollision bei gemeinsam verwendeten Resourcen zu hindern besteht darin, die gemeinsame Verwendung von Feldern zu unterbinden. Thread-lokale Felder (*thread-local storage*) ist ein Mechanismus, welcher für jeden Thread, den ein Objekt verwendet, automatisch einen eigenen Speicherbereich für ein bestimmtes Feld anlegt. Greifen beispielsweise fünf Threads auf ein Objekt mit dem Feld `x` zu, so erzeugt der Mechanismus fünf separate Speicherplätze für das `x`-Feld.

[163] Das Erzeugen und die Verwaltung thread-lokaler Felder ist die Aufgabe der Klasse `java.lang.ThreadLocal`:

```
//: concurrency/ThreadLocalVariableHolder.java
// Automatically giving each thread its own storage.
import java.util.concurrent.*;
import java.util.*;

class Accessor implements Runnable {
    private final int id;
    public Accessor(int idn) { id = idn; }
    public void run() {
        while(!Thread.currentThread().isInterrupted()) {
            ThreadLocalVariableHolder.increment();
            System.out.println(this);
            Thread.yield();
        }
    }
    public String toString() {
        return "#" + id + ": " +
            ThreadLocalVariableHolder.get();
    }
}

public class ThreadLocalVariableHolder {
    private static ThreadLocal<Integer> value =
        new ThreadLocal<Integer>() {
            private Random rand = new Random(47);
```

```
                protected synchronized Integer initialValue() {
                    return rand.nextInt(10000);
                }
            };
        public static void increment() {
            value.set(value.get() + 1);
        }
        public static int get() { return value.get(); }
        public static void main(String[] args) throws Exception {
            ExecutorService exec = Executors.newCachedThreadPool();
            for(int i = 0; i < 5; i++)
                exec.execute(new Accessor(i));
            TimeUnit.SECONDS.sleep(3); // Run for a while
            exec.shutdownNow(); // All Accessors will quit
        }
    } /* Output: (Sample)
          #0: 9259
          #1: 556
          #2: 6694
          #3: 1862
          #4: 962
          #0: 9260
          #1: 557
          #2: 6695
          #3: 1863
          #4: 963
          ...
       *///:~
```

`ThreadLocal`-Objekte werden in der Regel von statischen Feldern referenziert. Der Inhalt eines `ThreadLocal`-Objektes ist nach der Objekterzeugung nur über die Methoden `get()` und `set()` zugänglich. Die `get()`-Methode liefert eine Kopie des mit diesem Thread verknüpften Objektes. Die `set()`-Methode setzt ihr Argument in das diesem Thread zugeordnete Objekt ein und gibt das zuvor gespeicherte Objekt zurück. (Siehe Methoden `increment()` und `get()` der Klasse `ThreadLocalVariableHolder`). Beachten Sie, daß die Methoden `increment()` und `get()` nicht synchronisiert sind, da die Klasse `ThreadLocal` bereits garantiert, daß Wettlaufsituationen ausgeschlossen sind.

[164] Wenn Sie das Programm ausführen, können Sie erkennen, daß jeder individuelle Thread einen eigenen Speicherplatz allokiert, da jeder Thread seinen Zähler ganzzahlig hält, obwohl nur ein `ThreadLocalVariableHolder`-Objekt vorhanden ist.

22.4 Beenden der Verarbeitung einer Aufgabe

[165] Bei einigen der bisherigen Beispiele enthielt eine für alle Aufgaben sichtbare Klasse die Hilfsmethoden `cancel()` und `isCanceled()`. Die Aufgaben fragten regelmäßig `isCanceled()` ab, um zu ermitteln, ob sie sich beenden sollten. Dies ist ein vernünftiger Ansatz, um die Verarbeitung einer Aufgabe zu beenden. In manchen Fällen muß eine Aufgabe aber abrupt beendet werden. In diesem Abschnitt lernen Sie die [////(issues)] und Schwierigkeiten einer solchen Beendigung kennen.

[166] Das erste Beispiel demonstriert nicht nur das Beendigungsproblem, sondern liefert ein weiteres Beispiel für die gemeinsame Verwendung von Resourcen.

22.4.1 Der Park

[167] In dieser Simulation möchte die für einen Park verantwortliche Verwaltung wissen, wieviele Besucher die Anlage pro Tag durch die verschiedenen Eingänge betreten. Jeder Eingang hat ein Drehkreuz oder einen anderen Zählmechanismus und nach jeder Inkrementierung des Zählers an einem der Eingänge wird ein globaler Zähler erhöht, der die Anzahl sämtlicher Besucher repräsentiert.

```
//: concurrency/OrnamentalGarden.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Count {
    private int count = 0;
    private Random rand = new Random(47);
    // Remove the synchronized keyword to see counting fail:
    public synchronized int increment() {
        int temp = count;
        if(rand.nextBoolean()) // Yield half the time
            Thread.yield();
        return (count = ++temp);
    }
    public synchronized int value() { return count; }
}

class Entrance implements Runnable {
    private static Count count = new Count();
    private static List<Entrance> entrances =
        new ArrayList<Entrance>();
    private int number = 0;
    // Doesn't need synchronization to read:
    private final int id;
    private static volatile boolean canceled = false;
    // Atomic operation on a volatile field:
    public static void cancel() { canceled = true; }
    public Entrance(int id) {
        this.id = id;
        // Keep this task in a list. Also prevents
        // garbage collection of dead tasks:
        entrances.add(this);
    }
    public void run() {
        while(!canceled) {
            synchronized(this) {
                ++number;
            }
            print(this + " Total: " + count.increment());
            try {
                TimeUnit.MILLISECONDS.sleep(100);
            } catch(InterruptedException e) {
                print("sleep interrupted");
            }
        }
        print("Stopping " + this);
    }
    public synchronized int getValue() { return number; }
    public String toString() {
```

```
        return ''Entrance '' + id + '': '' + getValue();
    }
    public static int getTotalCount() {
        return count.value();
    }
    public static int sumEntrances() {
        int sum = 0;
        for(Entrance entrance : entrances)
            sum += entrance.getValue();
        return sum;
    }
}

public class OrnamentalGarden {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Entrance(i));
        // Run for a while, then stop and collect the data:
        TimeUnit.SECONDS.sleep(3);
        Entrance.cancel();
        exec.shutdown();
        if(!exec.awaitTermination(250, TimeUnit.MILLISECONDS))
            print(''Some tasks were not terminated!'');
        print(''Total: '' + Entrance.getTotalCount());
        print(''Sum of Entrances: '' + Entrance.sumEntrances());
    }
} /* Output: (Sample)
      Entrance 0: 1 Total: 1
      Entrance 2: 1 Total: 3
      Entrance 1: 1 Total: 2
      Entrance 4: 1 Total: 5
      Entrance 3: 1 Total: 4
      Entrance 2: 2 Total: 6
      Entrance 4: 2 Total: 7
      Entrance 0: 2 Total: 8
      ...
      Entrance 3: 29 Total: 143
      Entrance 0: 29 Total: 144
      Entrance 4: 29 Total: 145
      Entrance 2: 30 Total: 147
      Entrance 1: 30 Total: 146
      Entrance 0: 30 Total: 149
      Entrance 3: 30 Total: 148
      Entrance 4: 30 Total: 150
      Stopping Entrance 2: 30
      Stopping Entrance 1: 30
      Stopping Entrance 0: 30
      Stopping Entrance 3: 30
      Stopping Entrance 4: 30
      Total: 150
      Sum of Entrances: 150
   *///:~
```

[168] Genau ein `Count`-Objekt stellt den globalen Zähler aller Parkbesucher dar und wird von einem statischen Feld der Klasse `Entrance` referenziert. Die `Count`-Methoden `increment()` und `value()` sind synchronisiert, um den Zugriff auf das von `count` referenzierte Objekt kontrollieren zu

können. Die `increment()`-Methode verwendet ein `Random`-Objekt, um im Mittel bei jedem zweiten Aufruf zwischen dem Kopieren des `count`-Feldes in das Hilfsfeld `temp` und dem Inkrementieren des `temp`-Feldes sowie dem Zurückkopieren in das `count`-Feld die `yield()`-Methode aufzurufen. Wenn Sie die Synchronisierung von `increment()` entfernen, arbeitet die Simulation nicht mehr korrekt, da mehrere Aufgaben gleichzeitig den globalen Zählerstand ändern können (die `yield()`-Methode bewirkt, daß das Problem noch schneller eintritt).

[169] Jedes `Entrance`-Objekt pflegt die Anzahl Besucher, die den Park durch diesen Eingang betreten haben, mittels eines Objektfeldes namens `number`. Das `number`-Feld gestattet insbesondere die Gegenprüfung des globalen Zählers. Die `run()`-Methode eines `Entrance`-Objektes inkrementiert lediglich den lokalen und den globalen Zähler und „schläft“ anschließend für eine Zehntelsekunde.

[170] Da das `Entrance`-Feld `cancelled` eine volatiles Feld vom Typ `boolean` ist, welches nur abgefragt und bewertet wird (niemals aber in Kombination mit einem anderen Feld), können wir auf die Synchronisierung des Zugriffs verzichten. Wenn Sie in einem Fall wie diesem Zweifel haben, ist es stets besser, den Zugriff zu synchronisieren.

[171] Bei dieser Simulation wurde besonderer Wert darauf gelegt, alle Threads in geordneter Weise außer Betrieb zu nehmen. Der Grund für diesen Aufwand besteht zu einem Teil in der Motivation, vorzuführen, welche Sorgfalt beim Beenden eines threadbasierten Programms erforderlich ist. Andererseits dient dieses Beispiel im Laufe dieses Abschnitts noch dazu, die Bedeutung der `interrupt()`-Methode zu demonstrieren.

[172] Nach drei Sekunden ruft `main()` die statische `cancel()`-Methode der Klasse `Entrance` auf, danach die `shutdown()`- und die `awaitTermination()`-Methode des von `exec` referenzierten Exekutors. Die im Interface *`ExecutorService`* deklarierte Methode `awaitTermination()` wartet, bis die Verarbeitung aller Aufgaben beendet ist. Sind alle Aufgaben vor der festgelegten Zeitüberschreitung abgeschlossen, so gibt die Methode `true` zurück, andernfalls `false`, um anzuzeigen, daß nicht alle Aufgaben zuende verarbeitet wurden. Obwohl hierdurch alle Aufgaben veranlaßt werden, ihre `run()`-Methoden zu verlassen und somit ihren „Status“ als Aufgabe ablegen, sind die `Entrance`-Objekte noch immer „gültig“, da der `Entrance`-Konstruktor eine Referenz auf jedes erzeugte Objekt im statischen *`List<Entrance>`*-Objekt speichert. Daher verarbeitet die Methode `sumEntrances()` gültige `Entrance`-Objekte.

[173] Während das Programm läuft, sehen Sie sowohl den globalen als auch den lokalen Zähler, wenn eine Besucher durch einen der Eingänge geht. Wenn Sie die Synchronisierung der `Count`-Methode `increment()` entfernen, stimmt der globale Zähler nicht mehr mit der Summe der lokalen Zähler überein. Die Zählung funktioniert, solange das Mutex-Objekt verwendet wird, um den Zugriff auf den globalen Zähler zu synchronisieren. Beachten Sie, daß die Verwendung der Hilfsvariablen `temp` und das Aufrufen der `yield()`-Methode in `increment()` die Chance des Scheiterns überhöhen. Bei realistischen Threadproblemen kann die Wahrscheinlichkeit für das Auftreten eines Fehlers sehr gering sein, so daß Sie irrtümlich glauben, das Programm arbeite korrekt. Wie im obigen Beispiel, sind wahrscheinlich versteckte Probleme vorhanden, die noch nicht zu Ihrem Erfahrungsschatz gehören. Seien Sie also beim Durchsehen eines threadbasierten Programms besonders gewissenhaft.

Übungsaufgabe 17: (2) Simulieren Sie einen Geigerzähler (*radiation counter*) mit beliebig vielen externen Sensoren. ■

22.4.2 Beenden eines blockierten Threads

[174] Die `run()`-Methode der Klasse `Entrance` im Beispiel *OrnamentalGarden.java* enthielt eine `while`-Schleife mit einem `sleep()`-Aufruf. Wir wissen, daß die Verarbeitung der Aufgabe nach dem „Erwachen“ fortgesetzt wird und die Prüfung im Schleifenkopf erreicht. Dort besteht, nach der

Bewertung des `cancelled`-Flags mit `true` die Möglichkeit, die Schleife zu verlassen. Die Ausführung der `sleep()`-Methode ist nur eine Situation, in die Verarbeitung einer Aufgabe blockiert wird. Es gibt Situationen, in denen Sie eine blockierte Aufgabe abbrechen müssen.

22.4.2.1 Threadzustände

[175] Ein Thread befindet sich stets in einem von vier Zuständen:

- `NEW`: Ein Thread befindet sich nach seiner Erzeugung vorübergehend in diesem Zustand. Der Thread allokiert alle benötigten Systemresourcen und initialisiert sich. Ab jetzt kommt der Thread für die Zuteilung von Rechenzeit in Frage. Der Threadscheduler überführt den Thread in einen der Zustände `RUNNABLE` oder `BLOCKED`.
- `RUNNABLE`: In diesem Zustand kann ein Thread verarbeitet werden, wenn der Threadscheduler dem Thread Rechenzeit zuweist. Der Thread kann jeden Augenblick zur Verarbeitung ausgewählt werden. Nichts hindert den Thread an seiner Ausführung, wenn der Threadscheduler ihn wählt, das heißt der Thread ist nicht „tot“ (`TERMINATED`) oder blockiert.
- `BLOCKED`: Der Thread kann zwar verarbeitet werden, aber etwas hindert ihn daran. Befindet sich ein Thread im blockierten Zustand, so wird er vom Threadscheduler einfach übersprungen und erhält keine Rechenzeit. Der Thread führt keine Operationen aus, bevor er wieder in den Zustand `RUNNABLE` versetzt wird.
- `TERMINATED`: Ein Thread in diesem Zustand steht nicht mehr zur Verarbeitung zur Verfügung und erhält keine Rechenzeit. Die Verarbeitung seiner Aufgabe ist abgeschlossen und der Thread ist nicht länger lauffähig. Die Rückkehr aus der `run()`-Methode ist eine Möglichkeit, auf die eine Aufgabe „sterben“ kann. Eine Aufgabe kann aber auch unterbrochen werden, wie Sie im nächsten Unterabschnitt lernen werden.

22.4.2.2 Übergang in den blockierten Zustand

[176] Die Verarbeitung einer Aufgabe kann aus den folgenden Gründen blockiert werden:

- Die Aufgabe „schläft“ nach einem Aufruf der `sleep()`-Methode. In diesem Fall ruht die Verarbeitung der Aufgabe für die festgelegte Zeit.
- Die Verarbeitung des Threads wurde durch Aufrufen der `wait()`-Methode ausgesetzt. Der Thread kann erst weiterverarbeitet werden, wenn er mittels `notify()` oder `notifyAll()` (beziehungsweise über die äquivalenten Methoden `signal()` oder `signalAll()` aus der `java.util.concurrent`-Bibliothek der SE 5) benachrichtigt wird. Wir behandeln diese Methoden in Abschnitt 22.5.
- Die Aufgabe wartet auf das Ende einer Ein-/Ausgabeoperation.
- Die Aufgabe versucht, eine synchronisierte Methode eines anderen Objektes aufzurufen und die Sperre dieses Objektes wird bereits von einer anderen Aufgabe beansprucht und ist daher nicht verfügbar.

[177] In älteren Quelltexten sehen Sie eventuell die Methoden `suspend()` und `resume()`, um Threads zu blockieren beziehungsweise die Blockierung aufzuheben. Beide Methoden sind im modernen Java Development Kit als *deprecated* deklariert, da sie das Programm für Verklemmungen (*deadlocks*) anfällig machen und werden in diesem Buch nicht behandelt. Auch die `stop()`-Methode ist als *deprecated* deklariert, da sie die von einem Thread bewirkten Sperren nicht aufhebt. Befinden sich Objekte in einem inkonsistenten („beschädigten“) Zustand, können andere Threads diese Objekte in

diesem Zustand erreichen und modifizieren. Die hieraus erwachsenden Probleme können subtil und schwierig zu entdecken sein.

[178] Wir betrachten folgendes Problem: Sie müssen eine Aufgabe im blockierten Zustand beenden. Wenn Sie nicht darauf warten können, daß die Aufgabe eine Stelle im Programm erreicht, an der sie eine Information über ihren Betriebszustand abfragen und sich selbst beenden kann, müssen Sie erzwingen, daß die Aufgabe den blockierten Zustand verläßt.

22.4.3 Unterbrechung eines Threads

[179] Vielleicht können Sie sich vorstellen, daß das „Ausbrechen“ aus der Mitte der `run()`-Methode einer *`Runnable`*-Aufgabe viel unordentlicher vor sich geht, als die Methode etwa bis zum Abfragen eines `cancel`-Flags oder Erreichen einer anderen Stelle weiterlaufen zu lassen, an der die benötigten Resourcen wieder freigegeben werden können. Beim „Ausbrechen“ aus einer blockierten Aufgabe müssen eventuell Resourcen aufgeräumt werden. Das „Ausbrechen“ aus der `run()`-Methode einer Aufgabe paßt mehr als jede andere Situation zum Auswerfen einer Ausnahme. Daher rufen Threads in Java bei dieser Art von Abbruch Ausnahmen hervor. (Die Entscheidung, hier eine Ausnahme auszuwerfen, bewegt sich auf dem schmalen Grat zwischen bestimmungsmäßigem und nicht bestimmungsmäßigem Gebrauch von Ausnahmen, da Sie in der beschriebenen Situation mit Hilfe von Ausnahmen in die Ablaufkontrolle eingreifen.[18]) Wenn Sie die Verarbeitung einer Aufgabe auf diese Weise beenden, müssen Sie gewissenhaft alle Ausführungspfade Ihres Programmes durchgehen und in Ihrer `catch`-Klausel alles aufräumen, um einen definierten guten Zustand zu gewährleisten.

[180] Die Klasse `Thread` verfügt über die Methode `interrupt()`, um blockierte Aufgaben abbrechen zu können. Die `interrupt()`-Methode setzt ein Flag, das den Zustand des Threads als „unterbrochen“ kennzeichnet. Ein Thread dessen Unterbrechungsflag gesetzt ist, wirft eine Ausnahme vom Typ `java.lang.InterruptedException` aus, falls er bereits im blockierten Zustand ist oder versucht, diesen Zustand einzunehmen. Das Unterbrechungsflag wird zurückgesetzt, wenn die Ausnahme ausgeworfen oder die statische `Thread`-Methode `interrupted()` aufgerufen wird. Wie Sie bald sehen werden, ist die `interrupted()`-Methode eine zweite Möglichkeit, um die Schleife einer `run()`-Methode zu verlassen, ohne eine Ausnahme hervorzurufen.

[181] Das Aufrufen der `interrupted()`-Methode setzt voraus, daß Sie Zugriff auf das `Thread`-Objekt haben. Eventuell haben Sie bemerkt, daß die neue `java.util.concurrent`-Bibliothek den direkten Umgang mit `Thread`-Objekten zu vermeiden scheint und statt dessen versucht, alles über Exekutoren abzuwickeln. Die `shutdownNow()`-Methode eines Exekutors bewirkt, daß auf jedem gestarteten Thread die `interrupt()`-Methode aufgerufen wird. Das ist sinnvoll, da Sie in der Regel alle von einem Exekutor überwachten Ausgaben auf einmal beenden möchten, wenn ein Teil eines Projektes oder eines Programmes abgeschlossen ist. Andererseits gibt es Situationen, in denen Sie nur eine einzelne Aufgabe unterbrechen müssen. Wenn Sie einen Exekutor verwenden, können Sie mit dem Kontext einer Aufgabe in Verbindung bleiben, indem Sie die Aufgabe mit der `submit()`- anstelle der `execute()`-Methode starten. Die `submit()`-Methode gibt ein generisches *`Future`*`<?>`-Objekt mit nicht spezifiziertem Parametertyp zurück, because you won't ever call get() on it. Sie bewahren die *`Future`*-Referenz auf, um seine `cancel()`-Methode aufrufen zu können, das heißt um die Verarbeitung einer bestimmten Aufgabe unterbrechen zu können. Wenn Sie die `cancel()`-Methode mit dem Argument `true` aufrufen, erhält die Methode die erforderliche Berechtigung, um die `interrupt()`-Methode des entsprechenden Threads aufzurufen und ihn anzuhalten. Die `cancel()`-Methode ist

[18] Andererseits werden Ausnahmen stets in zeitlicher Nähe zu ihrer Ursache hervorgerufen. Folglich besteht keine Gefahr, wenn ein Programm mitten während einer Anweisung oder eines Methodenaufrufs abgebrochen wird. Wenn Sie Sperrobjekte (im Gegensatz zur traditionellen Synchronisierung) zusammen mit einer `try`/`catch`-Kombination verwenden, werden die Sperrobjekte beim Auswerfen der Ausnahme automatisch entsperrt.

also eine Möglichkeit, um per Exekutor gestartete Threads zu unterbrechen.

[182] Das folgende Beispiel zeigt die basics of interrupt() using Exekutors:

```
//: concurrency/Interrupting.java
// Interrupting a blocked thread.
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class SleepBlocked implements Runnable {
    public void run() {
        try {
            TimeUnit.SECONDS.sleep(100);
        } catch(InterruptedException e) {
            print("InterruptedException");
        }
        print("Exiting SleepBlocked.run()");
    }
}

class IOBlocked implements Runnable {
    private InputStream in;
    public IOBlocked(InputStream is) { in = is; }
    public void run() {
        try {
            print("Waiting for read():");
            in.read();
        } catch(IOException e) {
            if(Thread.currentThread().isInterrupted()) {
                print("Interrupted from blocked I/O");
            } else {
                throw new RuntimeException(e);
            }
        }
        print("Exiting IOBlocked.run()");
    }
}

class SynchronizedBlocked implements Runnable {
    public synchronized void f() {
        while(true) // Never releases lock
            Thread.yield();
    }
    public SynchronizedBlocked() {
        new Thread() {
            public void run() {
                f(); // Lock acquired by this thread
            }
        }.start();
    }
    public void run() {
        print("Trying to call f()");
        f();
        print("Exiting SynchronizedBlocked.run()");
    }
}

public class Interrupting {
    private static ExecutorService exec = Executors.newCachedThreadPool();
```

```
        static void test(Runnable r) throws InterruptedException{
            Future<?> f = exec.submit(r);
            TimeUnit.MILLISECONDS.sleep(100);
            print("Interrupting " + r.getClass().getName());
            f.cancel(true); // Interrupts if running
            print("Interrupt sent to " + r.getClass().getName());
        }
        public static void main(String[] args) throws Exception {
            test(new SleepBlocked());
            test(new IOBlocked(System.in));
            test(new SynchronizedBlocked());
            TimeUnit.SECONDS.sleep(3);
            print("Aborting with System.exit(0)");
            System.exit(0); // ... since last 2 interrupts failed
        }
    } /* Output: (95% match)
          Interrupting SleepBlocked
          InterruptedException
          Exiting SleepBlocked.run()
          Interrupt sent to SleepBlocked
          Waiting for read():
          Interrupting IOBlocked
          Interrupt sent to IOBlocked
          Trying to call f()
          Interrupting SynchronizedBlocked
          Interrupt sent to SynchronizedBlocked
          Aborting with System.exit(0)
       *///:~
```

Jede Aufgabe stellt eine andere Art von Blockierung dar. Die Klasse `SleepBlocked` ist ein Beispiel für eine unterbrechbare Blockierung (*interruptible blocking*), während `IOBlocked` und `SynchronizedBlocked` Beispiele für eine nicht unterbrechbare Blockierung (*uninterruptible blocking*) sind.[19] Das Beispiel zeigt, daß Ein-/Ausgabeanweisungen und synchronisierungsbedingte Sperren nicht unterbrechbare Blockierungen sind. Diese Erkenntnis hätte allerdings auch anhand des Quelltextes vorgesehen werden können, da sowohl bei der Ein-/Ausgabeanweisung als auch bei dem Aufrufen der synchronisierten Methode kein Ausnahmebehandler für Ausnahmen vom Typ `java.lang.InterruptedException` erforderlich ist.

[183] Die beiden ersten Klassen sind leicht verständlich: Die `run()`-Methode der ersten Klasse ruft `sleep()` auf, die `run()`-Methode der zweiten Klasse `read()`. Um vorführen zu können, daß eine synchronisierte Methode eine Blockierung verursachen kann, müssen wir zunächst die Sperre akquirieren. Wir bewerkstelligen dies, indem wir im Konstruktor einen Thread aus einer anonymen, von `Thread` abgeleiteten Klasse starten, welcher die Methode `f()` aufruft und dadurch das Objekt sperrt (dieser Thread muß sich von demjenigen unterscheiden, der die `run()`-Methode von `SynchronizedBlocked` ausführt, da ein Thread ein Objekt mehrfach sperren kann). Da der im Konstruktor gestartete Thread nicht aus `f()` zurückkehrt, wird die Sperre nicht aufgehoben. Die `run()`-Methode der `SynchronizedBlocked`-Aufgabe versucht, `f()` aufzurufen und wartet im blockierten Zustand auf die Freigabe der Sperre.

[184] Die Ausgabe dokumentiert, daß Sie die `sleep()`-Methode unterbrechen können (wie jede andere Methode, bei deren Aufruf Sie eine mögliche Ausnahme vom Typ `InterruptedException`

[19] Einige Versionen des Java Development Kits unterstützen Ausnahmen vom Typ `java.io.InterruptedIOException`. Die Unterstützung dieses Ausnahmetyps war allerdings nur teilweise und nicht plattformunabhängig implementiert. Das Auswerfen einer Ausnahme dieses Typs bewirkt, daß Ein-/Ausgabeobjekte unbrauchbar werden. Es ist unwahrscheinlich, daß zukünftige JDK-Versionen diese Ausnahmen weiterhin unterstützen.

behandeln müssen). Aufgaben die versuchen eine synchronisierungsbedingte Sperre zu akquirieren oder eine Ein-/Ausgabeanweisung auszuführen, können dagegen nicht unterbrochen werden. Das ist ein wenig beunruhigend, vor allem bei Aufgaben mit Ein-/Ausgabeanweisungen, da Ein-/Ausgabeanweisungen offenbar in der Lage sind, Ihr threadbasiertes Programm zu sperren. Dieser Gesichtspunkt betrifft insbesondere webbasierte Anwendungen.

[185] Eine plumpe aber wirksame Lösung dieses Problems besteht darin, die Resource zu schließen, welche für die Blockierung der Aufgabe verantwortlich ist:

```
//: concurrency/CloseResource.java
// Interrupting a blocked task by
// closing the underlying resource.
// {RunByHand}
import java.net.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

public class CloseResource {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InputStream socketInput = new Socket(''localhost'', 8080).getInputStream();
        exec.execute(new IOBlocked(socketInput));
        exec.execute(new IOBlocked(System.in));
        TimeUnit.MILLISECONDS.sleep(100);
        print(''Shutting down all threads'');
        exec.shutdownNow();
        TimeUnit.SECONDS.sleep(1);
        print(''Closing '' + socketInput.getClass().getName());
        socketInput.close(); // Releases blocked thread
        TimeUnit.SECONDS.sleep(1);
        print(''Closing '' + System.in.getClass().getName());
        System.in.close(); // Releases blocked thread
    }
} /* Output: (85% match)
      Waiting for read():
      Waiting for read():
      Shutting down all threads
      Closing java.net.SocketInputStream
      Interrupted from blocked I/O
      Exiting IOBlocked.run()
      Closing java.io.BufferedInputStream
      Exiting IOBlocked.run()
   *///:~
```

Nach dem Aufruf der `shutdownNow()`-Methode verdeutlichen die beiden Verzögerungen vor den beiden `close()`-Aufrufen, daß die Blockierung der Aufgaben nach dem Schließen der unterliegenden Resourcen aufgehoben wird. Interessanterweise wird die `interrupt()`-Methode beim Schließen des Sockets offensichtlich aufgerufen, nicht aber beim Schließen des `System.in`-Stroms.

[186] Die Klassen aus der neuen Ein-/Ausgabebibliothek (`java.nio`-Package, siehe Kapitel 19) gestatten eine kultiviertere Unterbrechung von Ein-/Ausgabeanweisungen. Blockierte Kanäle reagieren automatisch auf Unterbrechungen:

```
//: concurrency/NIOInterruption.java
// Interrupting a blocked NIO channel.
import java.net.*;
```

```
import java.nio.*;
import java.nio.channels.*;
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;

class NIOBlocked implements Runnable {
    private final SocketChannel sc;
    public NIOBlocked(SocketChannel sc) { this.sc = sc; }
    public void run() {
        try {
            print("Waiting for read() in " + this);
            sc.read(ByteBuffer.allocate(1));
        } catch(ClosedByInterruptException e) {
            print("ClosedByInterruptException");
        } catch(AsynchronousCloseException e) {
            print("AsynchronousCloseException");
        } catch(IOException e) {
            throw new RuntimeException(e);
        }
        print("Exiting NIOBlocked.run() " + this);
    }
}

public class NIOInterruption {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        ServerSocket server = new ServerSocket(8080);
        InetSocketAddress isa =
            new InetSocketAddress("localhost", 8080);
        SocketChannel sc1 = SocketChannel.open(isa);
        SocketChannel sc2 = SocketChannel.open(isa);
        Future<?> f = exec.submit(new NIOBlocked(sc1));
        exec.execute(new NIOBlocked(sc2));
        exec.shutdown();
        TimeUnit.SECONDS.sleep(1);
        // Produce an interrupt via cancel:
        f.cancel(true);
        TimeUnit.SECONDS.sleep(1);
        // Release the block by closing the channel:
        sc2.close();
    }
} /* Output: (Sample)
      Waiting for read() in NIOBlocked@7a84e4
      Waiting for read() in NIOBlocked@15c7850
      ClosedByInterruptException
      Exiting NIOBlocked.run() NIOBlocked@15c7850
      AsynchronousCloseException
      Exiting NIOBlocked.run() NIOBlocked@7a84e4
   *///:~
```

Das Beispiel demonstriert, daß Sie auch den unterliegenden Kanal schließen können, um die Blockierung aufzuheben, obwohl dieser Weg nur selten notwendig sein sollte. Beachten Sie, daß Sie `execute()`-Methode verwenden können, um die Verarbeitung beider Aufgaben zu starten und `shutdownNow()`, um alle Aufgabe abzuschließen. Die Speicherung des *`Future`*-Objektes im obigen Beispiel war nur erforderlich, um einen bestimmten Thread statt aller unterbrechen zu können.[20]

[20] Ervin Varga hat meine Recherchen zu diesem Unterabschnitt unterstützt.

Übungsaufgabe 18: (2) Schreiben Sie eine Klasse, die keine Aufgabe ist und eine Methode besitzt, welche `sleep()` mit einem langen Zeitintervall aufruft. Schreiben Sie eine Aufgabenklasse, welche die Methode in der vorigen Klasse aufruft. Starten Sie die Verarbeitung der Aufgabe in der `main()`-Methode und rufen Sie anschließend die `interrupt()`-Methode auf, um die Verarbeitung zu unterbrechen. Achten Sie darauf, daß die Aufgabe in geordneter Weise beendet wird. ■

Übungsaufgabe 19: (4) Ändern Sie das Park-Beispiel *OrnamentalGarden.java* so, daß die `interrupt()`-Methode aufgerufen wird. ■

Übungsaufgabe 20: (1) Ändern Sie das Beispiel *CachedThreadPool.java* (Seite 867) so, daß alle Ausgabe vor Beendigung ihrer Verarbeitung unterbrochen werden. ■

22.4.3.1 Blockierung durch ein Sperrobjekt

[187] Das Beispiel *Interrupting.java* hat gezeigt, daß die Verarbeitung einer Aufgabe, die eine synchronisierte Methode eines Objektes aufrufen muß, dessen Sperre bereits von einer anderen Aufgabe akquiriert wurde, solange blockiert wird, bis die Sperre wieder aufgehoben wird. Das folgende Beispiel zeigt, daß eine Aufgabe ein und dieselbe Sperre mehrmals akquirieren kann:

```
//: concurrency/MultiLock.java
// One thread can reacquire the same lock.
import static net.mindview.util.Print.*;

public class MultiLock {
    public synchronized void f1(int count) {
        if(count-- > 0) {
            print("f1() calling f2() with count " + count);
            f2(count);
        }
    }
    public synchronized void f2(int count) {
        if(count-- > 0) {
            print("f2() calling f1() with count " + count);
            f1(count);
        }
    }
    public static void main(String[] args) throws Exception {
        final MultiLock multiLock = new MultiLock();
        new Thread() {
            public void run() {
                multiLock.f1(10);
            }
        }.start();
    }
} /* Output:
      f1() calling f2() with count 9
      f2() calling f1() with count 8
      f1() calling f2() with count 7
      f2() calling f1() with count 6
      f1() calling f2() with count 5
      f2() calling f1() with count 4
      f1() calling f2() with count 3
      f2() calling f1() with count 2
      f1() calling f2() with count 1
      f2() calling f1() with count 0
   *///:~
```

Der in der `main()`-Methode erzeugte Thread dient dazu, die Methode `f1()` aufrufen. Anschließend rufen sich die Methoden `f1()` und `f2()` wechselseitig auf, bis der Zähler `count` auf 0 steht. Da die Aufgabe durch den ersten Aufruf von `f1()` das `MultiLock`-Objekt bereits gesperrt hat, akquiriert die Aufgabe die Sperre durch den Aufruf der `f2()`-Methode ein weiteres Mal und so weiter. Dies ist sinnvoll, da eine Aufgabe in der Lage sein muß, andere synchronisierte Methoden desselben Objektes aufzurufen, welches die Sperre bereits an sich gebracht hat.

[188] Sie haben bei den nicht unterbrechbaren Ein-/Ausgabeanweisungen beobachtet, daß jede in dieser Weise blockierte Aufgabe in der Lage ist, ein Programm einzufrieren. Zu den ab der SE 5 verfügbaren neuen Eigenschaften und Fähigkeiten hinsichtlich der Threadunterstützung gehört, daß eine bezüglich eines eintrittsinvarianten Sperrobjektes (`ReentrantLock`-Objekt, siehe Fußnote **??** auf Seite **??**) gesperrte Aufgabe unterbrochen werden kann, im Gegensatz zur Blockierung durch eine synchronisierte Methode oder einen kritischen Abschnitt:

```
//: concurrency/Interrupting2.java
// Interrupting a task blocked with a ReentrantLock.
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class BlockedMutex {
    private Lock lock = new ReentrantLock();
    public BlockedMutex() {
        // Acquire it right away, to demonstrate interruption
        // of a task blocked on a ReentrantLock:
        lock.lock();
    }
    public void f() {
        try {
            // This will never be available to a second task
            lock.lockInterruptibly(); // Special call
            print("lock acquired in f()");
        } catch(InterruptedException e) {
            print("Interrupted from lock acquisition in f()");
        }
    }
}

class Blocked2 implements Runnable {
    BlockedMutex blocked = new BlockedMutex();
    public void run() {
        print("Waiting for f() in BlockedMutex");
        blocked.f();
        print("Broken out of blocked call");
    }
}

public class Interrupting2 {
    public static void main(String[] args) throws Exception {
        Thread t = new Thread(new Blocked2());
        t.start();
        TimeUnit.SECONDS.sleep(1);
        System.out.println("Issuing t.interrupt()");
        t.interrupt();
    }
} /* Output:
      Waiting for f() in BlockedMutex
      Issuing t.interrupt()
```

```
    Interrupted from lock acquisition in f()
    Broken out of blocked call
*///:~
```

Der Konstruktor der Klasse `BlockedMutex` akquiriert die objektspezifische Sperre und hebt sie nicht mehr auf. Jeder Versuch, die Methode `f()` von einer anderen Aufgabe aus aufrufen (einer anderen als der Aufgabe, die das `BlockedMutex`-Objekt erzeugt hat), wird blockiert, weil die Sperre aus diesem Grund nicht akquiriert werden kann. Die `run()`-Methode der Klasse `Blocked2` kommt nur bis zum Aufruf `blocked.f()`. Beim Ausführen des Programms können Sie beobachten, daß eine durch ein Sperrobjekt bedingte Blockierung unterbrochen werden kann (im Gegensatz zu einer blockierten Ein-/Ausgabeanweisung).[21]

22.4.4 Abfragen des Unterbrechungsflags

[189] Beachten Sie beim Aufrufen der `interrupt()`-Methode eines Threads, daß die Unterbrechung nur dann eintritt, wenn der Thread eine blockierende Anweisung verarbeitet oder sich unmittelbar vor dem Eintritt in die Verarbeitung einer blockierenden Anweisung befindet. (Mit Ausnahme nicht unterbrechbarer Ein-/Ausgabeanweisungen oder synchronisierungsbedingter Blockierungen. In diesen Fällen können Sie nichts ausrichten.) Aber was können Sie tun, wenn Ihr Programm anhand einer Bedingung entscheidet, ob es eine blockierende Methode aufruft oder nicht? Wenn Ihnen nur das Auswerfen einer Ausnahme zur Verfügung steht, um einen blockierten Methodenaufruf zu beenden, sind Sie nicht immer in der Lage, die Schleife in der `run()`-Methode zu verlassen. Wenn Sie die Verarbeitung einer Aufgabe per `interrupt()` abbrechen möchten und die Schleife in der `run()`-Methode eine nicht unterbrechbare Blockierung verursacht hat, brauchen Sie folglich einen zweiten Mechanismus, um die Schleife verlassen zu können.

[190] Dieser Mechanismus ist das Unterbrechungsflag (*interrupted status*), welches durch Aufrufen der `interrupt()`-Methode auf `true` gesetzt wird. Das Unterbrechungsflag wird durch Aufrufen der statischen `Thread`-Methode `interrupted()` abgefragt. Die `interrupted()`-Methode gibt nicht nur an, ob die `interrupt()`-Methode aufgerufen wurde, sondern setzt außerdem das Unterbrechungsflag auf `false` zurück. Das Zurücksetzen des Unterbrechungsflag bewirkt, daß Sie nicht zweimal benachrichtigt werden, wenn die Verarbeitung einer Aufgabe unterbrochen wurde. Die Benachrichtigung erfolgt entweder über eine einzige Ausnahme vom Typ `InterruptedException` oder über einen einzigen Aufruf der statischen `Thread`-Methode `interrupted()` bei dem `true` zurückgegeben wird. Falls Sie den Zustand des Unterbrechungsflags erneut prüfen müssen, können Sie den Rückgabewert der `interrupted()`-Methode speichern.

[191] Das folgende Beispiel führt die typische Schreibweise vor, die Sie in Ihrer `run()`-Methode verwenden sollten, um sowohl unterbrechbare als auch nicht unterbrechbare Blockierungen behandeln zu können, wenn das Unterbrechnungsflag auf `true` gesetzt wurde:

```
//: concurrency/InterruptingIdiom.java
// General idiom for interrupting a task.
// {Args: 1100}
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class NeedsCleanup {
    private final int id;
    public NeedsCleanup(int ident) {
        id = ident;
```

[21] Beachten Sie, daß die Anweisung `t.interrupt()` vor der Anweisung `blocked.f()` verarbeitet werden kann (dieser Fall ist allerdings unwahrscheinlich).

```
        print("NeedsCleanup " + id);
    }
    public void cleanup() {
        print("Cleaning up " + id);
    }
}

class Blocked3 implements Runnable {
    private volatile double d = 0.0;
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // point1
                NeedsCleanup n1 = new NeedsCleanup(1);
                // Start try-finally immediately after definition
                // of n1, to guarantee proper cleanup of n1:
                try {
                    print("Sleeping");
                    TimeUnit.SECONDS.sleep(1);
                    // point2
                    NeedsCleanup n2 = new NeedsCleanup(2);
                    // Guarantee proper cleanup of n2:
                    try {
                        print("Calculating");
                        // A time-consuming, non-blocking operation:
                        for(int i = 1; i < 2500000; i++)
                            d = d + (Math.PI + Math.E) / d;
                        print("Finished time-consuming operation");
                    } finally {
                        n2.cleanup();
                    }
                } finally {
                    n1.cleanup();
                }
            }
            print("Exiting via while() test");
        } catch(InterruptedException e) {
            print("Exiting via InterruptedException");
        }
    }
}

public class InterruptingIdiom {
    public static void main(String[] args) throws Exception {
        if(args.length != 1) {
            print("usage: java InterruptingIdiom delay-in-mS");
            System.exit(1);
        }
        Thread t = new Thread(new Blocked3());
        t.start();
        TimeUnit.MILLISECONDS.sleep(new Integer(args[0]));
        t.interrupt();
    }
} /* Output: (Sample)
      NeedsCleanup 1
      Sleeping
      NeedsCleanup 2
      Calculating
```

```
    Finished time-consuming operation
    Cleaning up 2
    Cleaning up 1
    NeedsCleanup 1
    Sleeping
    Cleaning up 1
    Exiting via InterruptedException
*///:~
```

Die Klasse `NeedsCleanup` stellt eine Resource dar, die nach Verlassen der Schleife infolge einer Ausnahme aufgeräumt werden muß. Beachten Sie, daß jedem erzeugten `NeedsCleanup`-Objekt in der `run()`-Methode unmittelbar eine `try`/`finally`-Kombination folgt, um zu gewährleisten, daß die `cleanup()`-Methode stets aufgerufen wird.

[192] Das Programm erwartet das Zeitintervall vor dem Aufruf der `interrupt()`-Methode als Kommandozeilenargument. Sie können die `run()`-Methode der `Blocked3`-Aufgabe durch verschiedene Zeitintervalle an unterschiedlichen Stellen verlassen, etwa während der blockierenden `sleep()`-Methode oder während der nicht blockierenden Berechnung. Erfolgt der Aufruf der `interrupt()`-Methode nach dem Kommentar „point2“ also während der nicht blockierenden Berechnung, so wird der aktuelle Schleifendurchgang beendet, alle lokal referenzierten Objekte zerstört und schließlich die `while`-Schleife nach Prüfen der Abbruchbedingung beendet. Erfolgt der Aufruf der `interrupt()`-Methode aber zwischen „point1“ und „point2“, also nach dem Schleifenkopf und bevor oder während der blockierenden `sleep()`-Methode, so wird die Verarbeitung der Aufgabe infolge einer Ausnahme vom Typ `InterruptedException` beendet. In diesem Fall werden nur die `NeedsCleanup`-Objekte aufgeräumt, die zum Zeitpunkt des Auswerfens der Ausnahme vorhanden waren und Sie haben die Gelegenheit, in der `catch`-Klausel beliebige Aufräumarbeiten durchzuführen.

[193] Eine Klasse, die von der Unterbrechung eines Threads betroffen sein kann, muß Richtlinien einhalten, um stets einen konsistenten Zustand zu gewährleisten. Das bedeutet im allgemeinen, daß der Erzeugung von Objekten, bei denen Aufräumarbeiten erforderlich sind, unmittel eine `try`/`finally`-Kombination folgt, um zu gewährleisten, daß die entsprechenden Anweisungen unabhängig davon ausgeführt werden, auf welche Weise die `run()`-Methode beendet wird. Diese Vorgehensweise führt zu funktionstüchtigem Quelltext, hängt aber, da Destruktoren in Java bedauerlicherweise nicht automatisch aufgerufen werden, davon ab, daß der Programmierer entsprechende `try`/`finally`-Kombinationen anlegt.

22.5 Kooperierende Aufgaben

[194] Wenn Sie Threads einsetzen, um zwei oder mehr Aufgaben parallel zu verarbeiten, können Sie eine Aufgabe daran hindern, eine Resource einer anderen Aufgabe störend zu beeinflussen, indem Sie eine Sperre (Mutex-Objekt) verwenden, um die Verarbeitung der beiden Aufgaben zu synchronisieren. Wenn also zwei Aufgaben zugleich auf eine gemeinsame Resource zugreifen (in der Regel ein Bereich des Arbeitsspeichers), sorgen Sie mit Hilfe einer Sperre dafür, daß stets höchstens eine Aufgabe Zugriff auf die Resource erhält.

[195] In diesem Unterabschnitt lernen Sie, wie Aufgaben miteinander kooperieren, so daß mehrere Aufgaben ein Problem durch Zusammenarbeit lösen. Das Thema dieses Unterabschnitts ist nicht die störende gegenseitige Beeinflussung, sondern die harmonische Zusammenarbeit von Aufgaben. Es ist möglich, daß einige Teile eines Problems bereits gelöst sein müssen, bevor die Verarbeitung der übrigen Teile beginnen kann. Die Situation ähnelt der Planung eines Projektes, beispielsweise dem Bau Hauses: Zuerst muß das Fundament eines Hauses muß ausgehoben werden. Das Stahlgerüst kann aufgebaut werden, während die Betongußformen zusammengesetzt werden und beide Arbeitsschritte

müssen abgeschlossen sein, bevor die Bodenplatte gegossen werden kann. Die Rohrleitungen müssen fixiert worden sein, bevor die einzelnen Betonplatten gegossen werden können. Die Platten müssen verbaut worden sein, bevor die Tür- und Fensterrahmen an die Reihe kommen. Die Arbeiten können teilweise parallel verrichtet werden. Bei einem Teil der Arbeitsschritte ist es aber notwendig, daß alle Vorarbeiten abgeschlossen sind, bevor es weitergeht.

[196] Die Schlüsselfrage bei der Kooperation von Aufgaben ist die Quittierungsüberwachung zwischen diesen Aufgaben. Wir bewerkstelligen die Quittierung mit Hilfe derselben Grundlage wie bei der Synchronisierung des Zugriffs auf eine gemeinsam genutzte Resource: Der Sperrmechanismus garantiert nun, daß nur eine Aufgabe auf eine Benachrichtigung reagiert und eliminiert damit jede mögliche Wettlaufsituation (*race condition*). Der Sperrmechanismus gestattet, einer Aufgabe ihre Verarbeitung selbsttätig zu suspendieren, bis ein externe Bedingung eintritt (zum Beispiel „die Rohrleitungen sind fixiert“) und anzeigt, daß die Verarbeitung fortgesetzt werden kann. Wir besprechen in diesem Abschnitt die Quittierungsüberwachung zwischen Aufgaben, die durch die `Object`-Methoden `wait()` und `notifyAll()` implementiert ist. Das seit der SE 5 vorhandene Package `java.util.concurrent.locks` bietet außerdem Bedingungsobjekte (*`Condition`*-Objekte) mit den Methoden `await()` und `signal()`. Wir führen die möglichen Probleme und ihre Lösungen vor.

22.5.1 Die Methoden wait() und notifyAll()

[197] Die Methode `wait()` gestattet Ihnen, auf das Eintreten einer Bedingung zu warten, die mit den Mitteln der aktuellen Methode nicht beeinflußt werden kann. Häufig hängt das Eintreten dieser Bedingung von einer anderen Aufgabe ab. Der Betrieb einer Schleife in einer Aufgabe, die pausenlos den Status einer Bedingung abfragt, wird als *aktives Warten* (*busy waiting*) bezeichnet und gilt in der Regel als Verschwendung von Prozessorzyklen. Die `wait()`-Methode suspendiert dagegen die Verarbeitung der Aufgabe, während des Wartens. Die Aufgabe beendet ihren durch `wait()` verursachten Wartezustand und prüft ihre Bedingung nur dann, wenn eine der beiden Methoden `notify()` oder `notifyAll()` aufgerufen und dadurch angezeigt wird, das etwas für die Aufgabe interessantes passiert sein könnte. Die `wait()`-Methode gestattet also, Aktivitäten zwischen Aufgaben zeitlich aufeinander abzustimmen.

[198] Es ist wichtig, zu verstehen, daß die Methoden `sleep()` und `yield()` die akquirierte Sperre nicht aufheben, wenn sie aufgerufen werden. Ruft eine Aufgabe aber die `wait()`-Methode auf, so wird ihre Verarbeitung suspendiert und die Sperre aufgehoben. Da `wait()` die Sperre aufhebt, kann sie von einer anderen Aufgabe akquiriert werden, das heißt während der durch `wait()` verursachten Wartezeit können auch bezüglich desselben (nun nicht mehr gesperrten) Objektes synchronisierte Methoden aufgerufen werden. Dieser Aspekt ist wesentlich, da es typischerweise diese anderen Methoden sind, welche die Änderungen hervorrufen, aufgrund derer es sinnvoll ist, die suspendierte Aufgabe zur Weiterverarbeitung „aufzuwecken“. Indem Sie die `wait()`-Methode aufrufen, dokumentieren Sie, daß Ihre Aufgabe alles getan hat, was sie im Augenblick tun kann und daher einerseits in einen Wartezustand übergeht und andererseits die Verarbeitung anderer synchronisierter Methoden ermöglichen soll.

[199] Die `wait()`-Methode hat zwei Versionen. Die eine Version erwartet ein Argument in Millisekunden, welches dieselbe Wirkung wie bei der `sleep()`-Methode hat, nämlich die Verarbeitung einer Aufgabe für eine Mindestzeit auszusetzen. Im Gegensatz zu `sleep()` gilt für `wait(pause)`:

- Die Sperre wird während der durch `wait()` verursachten Wartezeit aufgehoben.
- Neben dem Abwarten auf das Verstreichen der vereinbarten Zeitdauer können Sie die durch `wait()` bewirkte Wartezeit per `notify()` beziehungsweise `notifyAll()` vorzeitig beenden.

Die zweite und häufigere Version hat kein Argument. Die argumentlose `wait()`-Methode verursacht einen Wartezustand unbestimmter Dauer, bis der Thread per `notify()` beziehungsweise `notifyAll()` benachrichtig wird.

[200] Ein besonderer Aspekt ist, daß die Methoden `wait()`, `notify()` und `notifyAll()` *nicht* der Klasse `Thread`, sondern der Klasse `Object` angehören. Daß eine Methode, die exklusiv der Threadprogrammierung dient, Teil der universellen Basisklasse ist, mutet auf den ersten Blick seltsam an. Andererseits manipulieren die Methoden `wait()`, `notify()` und `notifyAll()` die Sperre, welche ebenfalls Teil jedes einzelnen Objektes ist. Sie können die `wait()`-Methode daher in jeder beliebigen synchronisierten Methode aufrufen, gleichgültig ob die Klasse von `Thread` abgeleitet ist oder `Runnable` implementiert. Genau genommen, sind synchronisierte Methoden und kritische Abschnitte die einzigen Kontexte, in denen Sie die Methoden `wait()`, `notify()` und `notifyAll()` aufrufen können (`sleep()` darf dagegen auch außerhalb synchronisierter Kontexte aufgerufen werden, da die Methode den Zustand der Sperre nicht ändert). Wenn Sie `wait()`, `notify()` oder `notifyAll()` außerhalb synchronisierter Kontexte aufrufen, läßt sich das Programm zwar übersetzen, wirft aber zur Laufzeit eine Ausnahme vom Typ `java.lang.IllegalMonitorStateException` mit der wenig intuitiven Meldung „current thread not owner“ aus. Die Meldung bedeutet, daß die Aufgabe, welche `wait()`, `notify()` oder `notifyAll()` aufgerufen hat die Sperre des Objekt „besitzen“ (akquiriert haben) muß, bevor sie eine dieser drei Methode aufrufen kann.

[201] *You can ask another object to perform an operation that manipulates its own lock. To do this, you must first capture that object's lock.* Um beispielsweise per `notifyAll()` eine Benachrichtigung an das von `x` referenzierte Objekt zu senden, müssen Sie die Methode einem bezüglich `x` synchronisierten Block aufrufen:

```
synchronized(x) {
    x.notifyAll();
}
```

[202] Wir betrachten nun ein einfaches Beispiel. Das Programm *WaxOMatic.java* beinhaltet zwei Vorgänge: Ein Vorgang trägt eine Schicht Wachs auf ein Auto auf der andere Vorgang poliert den Wagen. Der Poliervorgang kann nicht beginnen, bevor das Wachs aufgetragen ist und der Wachsvorgang muß warten bis der Poliervorgang beendet ist, bevor eine weitere Schicht Wachs aufgetragen werden kann. Die Aufgaben `WaxOn` und `WaxOff` verwenden ein gemeinsames `Car`-Objekt. Das `Car`-Objekt ruft die Methoden `wait()` beziehungsweise `notifyAll()` auf, um die Aufgaben zu suspendieren beziehungsweise neu zu starten, während sie auf ihre jeweilige Bedingung warten:

```
//: concurrency/waxomatic/WaxOMatic.java
// Basic task cooperation.
package concurrency.waxomatic;
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Car {
    private boolean waxOn = false;
    public synchronized void waxed() {
        waxOn = true; // Ready to buff
        notifyAll();
    }
    public synchronized void buffed() {
        waxOn = false; // Ready for another coat of wax
        notifyAll();
    }
    public synchronized void waitForWaxing()
        throws InterruptedException {
        while(waxOn == false)
```

```
            wait();
        }
        public synchronized void waitForBuffing()
            throws InterruptedException {
            while(waxOn == true)
                wait();
        }
    }

    class WaxOn implements Runnable {
        private Car car;
        public WaxOn(Car c) { car = c; }
        public void run() {
            try {
                while(!Thread.interrupted()) {
                    printnb("Wax On! ");
                    TimeUnit.MILLISECONDS.sleep(200);
                    car.waxed();
                    car.waitForBuffing();
                }
            } catch(InterruptedException e) {
                print("Exiting via interrupt");
            }
            print("Ending Wax On task");
        }
    }

    class WaxOff implements Runnable {
        private Car car;
        public WaxOff(Car c) { car = c; }
        public void run() {
            try {
                while(!Thread.interrupted()) {
                    car.waitForWaxing();
                    printnb("Wax Off! ");
                    TimeUnit.MILLISECONDS.sleep(200);
                    car.buffed();
                }
            } catch(InterruptedException e) {
                print("Exiting via interrupt");
            }
            print("Ending Wax Off task");
        }
    }

    public class WaxOMatic {
        public static void main(String[] args) throws Exception {
            Car car = new Car();
            ExecutorService exec = Executors.newCachedThreadPool();
            exec.execute(new WaxOff(car));
            exec.execute(new WaxOn(car));
            TimeUnit.SECONDS.sleep(5); // Run for a while...
            exec.shutdownNow(); // Interrupt all tasks
        }
    } /* Output: (95% match)
          Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On!
    Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax
    On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Exiting via interrupt
          Ending Wax On task
```

```
        Exiting via interrupt
        Ending Wax Off task
    *///:~
```

Das `boolean`-Feld `waxOn` der Klasse `Car` gibt den Zustand des Vorgangs „Wachsen und Polieren“ an.

[203] Die Methode `waitForWaxing()` wertet das `waxOn`-Feld aus und suspendiert, falls das `waxOn`-Feld auf `false` steht, die Verarbeitung der aufrufenden Aufgabe per `wait()`. Es ist wichtig, daß die `wait()`-Methode in einer synchronisierten Methode aufgerufen wird und die Aufgabe die Sperre akquiriert hat. Das Aufrufen der `wait()`-Methode bewirkt, daß der Thread suspendiert und die Sperre aufgehoben wird. Das Aufheben der Sperre ist wichtig, da sie für eine andere Aufgabe akquirierbar sein muß, um den Zustand des Objektes auf sichere Weise ändern zu können (zum Beispiel muß das `waxOn`-Feld auf `true` gesetzt werden, damit die Verarbeitung der suspendierten Aufgabe fortgesetzt werden kann). Ruft die `WaxOn`-Aufgabe die Methode `waxed()` auf, so muß die Sperre akquirierbar sein, um das `waxOn`-Feld auf `true` setzen zu können. Anschließend ruft `waxed()` die Methode `notifyAll()` auf, wodurch die durch `wait()` suspendierte Aufgabe „aufgeweckt“ wird. Das „Aufwachen“ aus dem durch `wait()` verursachten Wartezustand setzt voraus, daß die wartende Aufgabe die beim Aufruf der `wait()`-Methode aufgehobene Sperre akquirieren kann. Die Aufgabe „erwacht“ nicht, bevor die Sperre verfügbar wird.[22]

[204] Die `run()`-Methode der `WaxOn`-Aufgabe stellt den ersten Arbeitsschritt beim Wachsen eines Autos dar. Ein Aufruf der `sleep()`-Methode simuliert die zum Wachsen benötigte Zeit. Danach „teilt die `run()`-Methode dem Auto mit“, das der Wachsvorgang beendet ist und ruft die Methode `waitForBuffing()` auf, welche die `WaxOn`-Aufgabe per `wait()` suspendiert, bis die `WaxOff`-Aufgabe die `buffed()`-Methode aufruft, um das `waxOn`-Feld umzuschalten und die `notifyAll()`-Methode aufzurufen. Die `WaxOff`-Aufgabe ruft ihrerseits sofort die `waitForWaxing()`-Methode auf und wird suspendiert, bis das Wachs aufgetragen (`waxOn`-Feld auf `true` gesetzt) ist und die `waxed()`-Methode aufgerufen wird. Wenn Sie das Programm aufrufen, können Sie beobachten, wie sich dieser Zweischrittprozeß wiederholt, indem die Kontrolle zwischen den beiden Aufgabe hin- und herpendelt. Nach fünf Sekunden werden beide Aufgaben per `interrupt()` angehalten. (Die *`ExecutorService`*-Methode `shutdownNow()` ruft für jeden verwalteten Thread dessen `interrupt()`-Methode auf.)

[205] Das Beispiel *waxomatic/WaxOMatic.java* macht deutlich, daß Sie `wait()` stets im Körper einer `while`-Schleife aufrufen müssen, welche die spezifische(n) Bedingunge(n) prüft. Dies ist aus den folgenden Gründen wichtig:

- Es können mehrere Aufgaben aus ein und demselben Grund auf ein und dieselbe Sperre warten und die erste „erwachende“ Aufgabe kann die Situation verändern (wenn Sie es nicht selbst

[22] Bei manchen Plattformen gibt es eine dritte Möglichkeit, um den durch `wait()` verursachten Wartezustand zu verlassen, nämlich das sogenannte „Spurious Wakeup“, übersetzt etwa „Störendes/unberechtigtes Erwachen“. „Spurious Wakeup“ bedeuted im wesentlichen, daß ein Thread den blockierten Zustand (während des Wartens auf eine Bedingung oder Semaphore) vorzeitig verläßt, ohne per `notify()` oder `notifyAll()` (beziehungsweise `signal()` oder `signalAll()` im Fall der neuen *`Condition`*-Objekte) benachrichtigt worden zu sein. Der Thread „erwacht“ scheinbar von selbst. „Spurious Wakeups“ existieren, weil die Implementierung einer POSIX-konformen Threadunterstützung ~~or the equivalent~~ auf manchen Plattformen nicht so einfach möglich ist, wie es sein sollte. „Spurious Wakeups“ zuzulassen, erleichtert die Arbeit, eine Bibliothek wie *pthreads* für solche Plattformen zu schreiben.

Anmerkung des Übersetzers: Die API-Dokumentation der `Object`-Methode `wait()` sagt aus: „A thread can also wake up without being notified, interrupted, or timing out, a so-called *spurious wakeup*. While this will rarely occur in practice, applications must guard against it by testing for the condition that should have caused the thread to be awakened, and continuing to wait if the condition is not satisfied. In other words, waits should always occur in loops.“ Übersetzt etwa: „Ein Thread kann aus seinem Wartezustand zurückkehren, ohne zuvor benachrichtigt oder unterbrochen worden zu sein oder die für seinen Wartezustand festgesetzte Mindestzeitdauer überschritten zu haben. Dieser Effekt wird als „Spurious Wakeup“ bezeichnet. Obwohl dieser Effekt in der Praxis selten eintritt, müssen Anwendung entsprechend vorbereitet werden, indem sie die Bedingung prüfen, die das Erwachen des Threads auslöst und den Wartezustand fortsetzen, wenn die Bedingung nicht erfüllt ist, das heißt *`wait()`-Aufrufe sollten stets in Schleifen stehen*.“

tun, kann es jemand tun, der eine Klasse von Ihrer Klasse ableitet). In diesem Fall sollte diese Aufgabe wieder suspendiert werden, bis die für sie interessante Bedingung eintritt.

- „Erwacht“ eine Aufgabe aus ihrem per `wait()` verursachten Wartezustand, so kann eine andere Aufgabe Änderungen bewirkt haben, unter denen die erstere Aufgabe nicht verarbeitet werden kann oder ihre Verarbeitung unter den gegebenen Umständen uninteressant ist. Auch hier sollte die Aufgabe per `wait()` suspendiert werden.
- Es können mehrere Aufgaben aus verschiedenen Gründen auf ein und dieselbe Sperre warten (in diesem Fall *müssen* Sie `notifyAll()` aufrufen). In diesem Fall müssen Sie prüfen, ob die Aufgabe aus dem richtigen Grund „geweckt“ wurde und andernfalls wiederum `wait()` aufrufen.

Es ist wichtig, die jeweils interessante Bedingung zu prüfen und die Aufgabe bei Nichterfüllung mittels `wait()` wieder zu suspendieren. Die Schreibweise dafür ist die oben vorgeführte `while`-Schleife.

Übungsaufgabe 21: (2) Legen Sie zwei Aufgabenklassen an, die das Interface *`Runnable`* implementieren. Die `run()`-Methode der ersten Klasse ruft `wait()` auf. Die zweite Klasse muß über eine Referenz auf ein Objekt der ersten Klasse verfügen. Die `run()`-Methode der zweiten Klasse ruft die `notifyAll()`-Methode der ersten Aufgabe auf, nachdem einige Sekunden verstrichen sind und die erste Aufgabe eine Textmeldung ausgeben konnte. Testen Sie Ihre beiden Klassen mit Hilfe eines Exekutors. ■

Übungsaufgabe 22: (4) Entwickeln Sie ein Beispiel für aktives Warten (*busy waiting*). Eine Aufgabe „schläft“ für eine Zeitdauer und setzt anschließend ein Flag auf `true`. Die zweite Aufgabe überwacht das Flag mittels einer `while`-Schleife (dies ist das aktive Warten) und reagiert auf den Flagzustand `true`, indem sie das Flag auf `false` zurücksetzt und die Änderung auf der Konsole meldet. Beobachten Sie, wieviel Zeit das Programm im aktiven Wartezustand verliert und schreiben Sie eine zweite Version des Programms, die statt aktivem Warten die `wait()`-Methode verwendet. ■

22.5.1.1 Signalverlust

[206] Sind zwei Threads per `notify()`/`wait()` oder `notifyAll()`/`wait()` koordiniert, so ist es möglich, daß die Benachrichtigung durch `notify()` beziehungsweise `notifyAll()` verloren geht. Thread `T1` benachrichtige `T2` und beide seien wie folgt (fehlerhaft) implementiert:

```
T1:
synchronized(sharedMonitor) {
    <setup condition for T2>
    sharedMonitor.notify();
}

T2:
while(someCondition) {
    // Point 1
    synchronized(sharedMonitor) {
      sharedMonitor.wait();
    }
}
```

Die `<setup condition for T2>` ist ein Mechanismus, der `T2` daran hindert, die `wait()`-Methode aufzurufen, sofern der Aufruf nicht bereits geschehen ist.

[207] Angenommen, `T2` wertet die Bedingung `someCondition` aus und die Auswertung ergibt, daß die Bedingung erfüllt ist. Der Threadscheduler möge bei „Point 1“ auf die Verarbeitung von `T1` umschalten. `T1` führt seine Anweisungen aus und ruft schließlich `notify()` auf. Wenn die Verarbeitung

von T2 fortgesetzt wird, ist es für T2 zu spät, um zu bemerken, daß sich die Bedingung in der Zwischenzeit geändert hat und T2 ruft blindlinks wait() auf. Der Aufruf der notify()-Methode geht ungenutzt verloren und T2 wartet auf unbestimmte Zeit auf das Signal, welches bereits gesendet wurde und verursacht eine Verklemmung (*deadlock*).

[208] Die Lösung besteht darin, die durch someCondition gegebene Wettlaufsituation (*race condition*) zu meiden. Der korrekte Ansatz für T2 lautet:

```
synchronized(sharedMonitor) {
    while(someCondition)
        sharedMonitor.wait();
}
```

Wird T1 zuerst verarbeitet, so erkennt T2 während seiner Ausführung, daß sich die Bedingung geändert hat und ruft wait() *nicht* auf. Wird andererseits T2 zuerst verarbeitet, so ruft T2 die wait()-Methode auf und wird später von T1 benachrichtigt. Das Signal geht also nicht verloren.

22.5.2 Vergleich der Methoden notify() und notifyAll()

[209] Da aufgrund verschiedener technischer Vorgänge mehr als eine Aufgabe die wait()-Methode eines Car-Objektes aufgerufen haben kann, ist es sicherer statt notify() die notifyAll()-Methode aufzurufen. Beim obigen Programm befindet sich andererseits stets nur eine Aufgabe im Wartezustand, so daß die Wahl von notify() anstelle von notifyAll() vertretbar ist.

[210] Die Verwendung der notify()-Methode anstelle von notifyAll() ist eine Optimierung. Die notify()-Methode ruft stets nur eine der unter Umständen zahlreichen Aufgaben auf, die auf eine Sperre warten und Sie müssen sicher sein, daß Sie die richtige Aufgabe bekommen. Außerdem müssen bei notify() alle Aufgaben auf ein und dieselbe Bedingung warten, da Sie bei unterschiedlichen Bedingungen nicht wissen, ob die richtige Aufgabe benachrichtigt wird. Wenn Sie notify() wählen, darf nur eine Aufgabe vom Eintritt einer Bedingung profitieren. Schließlich müssen diese Bedingungen für alle abgeleiteten Klassen stets wahr sein. Kann eine dieser Forderungen nicht eingehalten werden, so müssen Sie notifyAll() statt notify() verwenden.

[211] Eine verwirrende Aussage, die bei Diskussionen über Threadprogrammierung unter Java häufig vorkommt ist, daß die notifyAll()-Methode „alle wartenden Aufgaben aufweckt“. Bedeutet dieses Aussage etwa, daß jede Aufgabe, die sich in einem beliebigen Teil des Programms im Wartezustand befindet, durch das Aufrufen der notifyAll()-Methode an einer beliebigen Stelle „aufwecken“ läßt? Die Ausgabeanweisung der Task2-Aufgabe im folgenden Beispiel zeigt, daß dies nicht zutrifft. Genau genommen „weckt“ notifyAll() nur die Aufgaben, die auf die Sperre des Objektes warten, zu dem die notifyAll()-Methode gehört:

```
//: concurrency/NotifyVsNotifyAll.java
import java.util.concurrent.*;
import java.util.*;

class Blocker {
    synchronized void waitingCall() {
        try {
            while(!Thread.interrupted()) {
                wait();
                System.out.print(Thread.currentThread() + " ");
            }
        } catch(InterruptedException e) {
            // OK to exit this way
        }
```

```
    }
    synchronized void prod() { notify(); }
    synchronized void prodAll() { notifyAll(); }
}

class Task implements Runnable {
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

class Task2 implements Runnable {
    // A separate Blocker object:
    static Blocker blocker = new Blocker();
    public void run() { blocker.waitingCall(); }
}

public class NotifyVsNotifyAll {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < 5; i++)
            exec.execute(new Task());
        exec.execute(new Task2());
        Timer timer = new Timer();
        timer.scheduleAtFixedRate(new TimerTask() {
                boolean prod = true;
                public void run() {
                    if(prod) {
                        System.out.print("\nnotify() ");
                        Task.blocker.prod();
                        prod = false;
                    } else {
                        System.out.print("\nnotifyAll() ");
                        Task.blocker.prodAll();
                        prod = true;
                    }
                }
            }, 400, 400); // Run every .4 second
        TimeUnit.SECONDS.sleep(5); // Run for a while...
        timer.cancel();
        System.out.println("\nTimer canceled");
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.print("Task2.blocker.prodAll() ");
        Task2.blocker.prodAll();
        TimeUnit.MILLISECONDS.sleep(500);
        System.out.println("\nShutting down");
        exec.shutdownNow(); // Interrupt all tasks
    }
} /* Output: (Sample)
      notify() Thread[pool-1-thread-1,5,main]
      notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main]
      Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-3,5,main]
      Thread[pool-1-thread-2,5,main]
      notify() Thread[pool-1-thread-1,5,main]
      notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main]
      Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-4,5,main]
      Thread[pool-1-thread-5,5,main]
      notify() Thread[pool-1-thread-1,5,main]
      notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main]
      Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-3,5,main]
```

```
    Thread[pool-1-thread-2,5,main]
    notify() Thread[pool-1-thread-1,5,main]
    notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main]
    Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-4,5,main]
    Thread[pool-1-thread-5,5,main]
    notify() Thread[pool-1-thread-1,5,main]
    notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-5,5,main]
    Thread[pool-1-thread-4,5,main] Thread[pool-1-thread-3,5,main]
    Thread[pool-1-thread-2,5,main]
    notify() Thread[pool-1-thread-1,5,main]
    notifyAll() Thread[pool-1-thread-1,5,main] Thread[pool-1-thread-2,5,main]
    Thread[pool-1-thread-3,5,main] Thread[pool-1-thread-4,5,main]
    Thread[pool-1-thread-5,5,main]
    Timer canceled
    Task2.blocker.prodAll() Thread[pool-1-thread-6,5,main]
    Shutting down
*///:~
```

Jede der Aufgabenklassen `Task` beziehungsweise `Task2` verfügt über ein eigenes `Blocker`-Objekt: Die Verarbeitung von Aufgaben des Typs `Task` wird an dem von dem statischen `Task.blocker`-Feld, die Verarbeitung von Aufgaben des Typs `Task2` dagegen an dem von dem statischen `Task2.blocker`-Feld referenzierten `Blocker`-Objekt blockiert. Die `main()`-Methode enthält ein `Timer`-Objekt, dessen `run()`-Methode alle 0.4 Sekunden einmal ausgeführt wird und (über die `prod`-Methoden) abwechselnd die Methoden `notify()` beziehungsweise `notifyAll()` des von `Task.blocker` referenzierten `Blocker`-Objektes aufruft.

[212] Sie erkennen an der Ausgabe, daß, obwohl eine `Task2`-Aufgabe vorhanden und bezüglich des von `Task2.blocker` referenzierten `Blocker`-Objektes blockiert ist, weder die auf dem von `Task.blocker` referenzierten `Blocker`-Objekt aufgerufene `notify()`- noch die `notifyAll()`-Methode die `Task2`-Aufgabe zum „Aufwachen“ veranlassen. In gleicher Weise läßt die am Ende der `main()`-Methode aufgerufene `cancel()`-Methode die ersten fünf Aufgaben unbeeinflußt weiterlaufen und erhält die Blockierung durch die `wait()`-Methode aufrecht. Die Ausgabe zur Methode `Task2.blocker.prodAll()` enthält keinen der Threads, die auf die Sperre des von `Task.blocker` referenzierten `Blocker`-Objektes warten.

[213] Ein Blick auf die `Blocker`-Methoden `prod()` und `prodAll()` zeigt, daß dieses Verhalten sinnvoll ist. Sowohl `prod()` als auch `prodAll()` ist synchronisiert, akquiriert also die Sperre des Objektes zu dem sie gehört. Ruft nun eine dieser Methoden `notify()` oder `notifyAll()` auf, so können sich die Aufrufe nur auf *diese* Sperre beziehen, also nur Threads „aufwecken“, die auf diese bestimmte Sperre warten.

[214] Die `Blocker`-Methode `waitingCall()` ist so einfach, daß die Schleifendefinition `while(!Thread.interrupted())` durch die einfachere Variante `for(;;)` ersetzt werden könnte. Der Effekt wäre in beiden Fällen identisch, da es in diesem Beispiel keinen Unterschied zwischen dem Verlassen der Schleife infolge einer Ausnahme und dem Verlassen aufgrund des Unterbrechungsflags gibt. Stets werden dieselben Anweisungen ausgeführt. Der Ordnung halber wird aber auch in diesem Beispiel das Unterbrechungsflag geprüft, weil es eben zwei Möglichkeiten gibt, um eine Schleife zu verlassen. Andernfalls riskieren Sie bei einer späteren Erweiterung des Programms, daß Sie einen Fehler einschleppen, wenn Sie nicht beide Austrittsmöglichkeiten berücksichtigen.

Übungsaufgabe 23: (7) Zeigen Sie, daß das Beispiel *WaxOMatic.java* funktionstüchtig ist, wenn Sie `notify()` statt `notifyAll()` verwenden. ■

22.5.3 Erzeuger/Verbraucher-Systeme

[215] Stellen Sie sich ein Restaurant mit einem Koch (*chef*) und einem Kellner (*waitperson*) vor. Der Kellner muß warten, bis der Koch eine Mahlzeit zubereitet hat. Ist der Koch fertig, so benachrichtigt er den Kellner, der die Mahlzeit abholt, serviert und erneut wartet. Dies ist ein Beispiel für zwei kooperierende Aufgaben: Der Koch repräsentiert den Erzeuger (*producer*) und der Kellner den Verbraucher (*consumer*). Beide Aufgaben müssen ihren Zustände gegenseitig quittieren, während die Mahlzeiten erzeugt beziehungsweise verbraucht werden. Außerdem muß sich die Anwendung in geordneter Weise beenden lassen. Das folgende Beispiel beschreibt diese Situation:

```
//: concurrency/Restaurant.java
// The producer-consumer approach to task cooperation.
import java.util.concurrent.*;
import static net.mindview.util.Print.*;

class Meal {
    private final int orderNum;
    public Meal(int orderNum) { this.orderNum = orderNum; }
    public String toString() { return ”Meal ” + orderNum; }
}

class WaitPerson implements Runnable {
    private Restaurant restaurant;
    public WaitPerson(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal == null)
                        wait(); // ... for the chef to produce a meal
                }
                print(”Waitperson got ” + restaurant.meal);
                synchronized(restaurant.chef) {
                    restaurant.meal = null;
                    restaurant.chef.notifyAll(); // Ready for another
                }
            }
        } catch(InterruptedException e) {
            print(”WaitPerson interrupted”);
        }
    }
}

class Chef implements Runnable {
    private Restaurant restaurant;
    private int count = 0;
    public Chef(Restaurant r) { restaurant = r; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    while(restaurant.meal != null)
                        wait(); // ... for the meal to be taken
                }
                if(++count == 10) {
                    print(”Out of food, closing”);
                    restaurant.exec.shutdownNow();
                }
```

```
                    printnb("Order up! ");
                    synchronized(restaurant.waitPerson) {
                        restaurant.meal = new Meal(count);
                        restaurant.waitPerson.notifyAll();
                    }
                    TimeUnit.MILLISECONDS.sleep(100);
                }
            } catch(InterruptedException e) {
                print("Chef interrupted");
            }
        }
    }

    public class Restaurant {
        Meal meal;
        ExecutorService exec = Executors.newCachedThreadPool();
        WaitPerson waitPerson = new WaitPerson(this);
        Chef chef = new Chef(this);
        public Restaurant() {
            exec.execute(chef);
            exec.execute(waitPerson);
        }
        public static void main(String[] args) {
            new Restaurant();
        }
    } /* Output:
          Order up! Waitperson got Meal 1
          Order up! Waitperson got Meal 2
          Order up! Waitperson got Meal 3
          Order up! Waitperson got Meal 4
          Order up! Waitperson got Meal 5
          Order up! Waitperson got Meal 6
          Order up! Waitperson got Meal 7
          Order up! Waitperson got Meal 8
          Order up! Waitperson got Meal 9
          Out of food, closing
          WaitPerson interrupted
          Order up! Chef interrupted
       *///:~
```

[216] Das Restaurant (`Restaurant`-Objekt) ist der Schwerpunkt des Systems Kellner (`WaitPerson`-Objekt)/Koch (`Chef`-Objekt). Beide Angestellten müssen wissen, in welchem Restaurant sie arbeiten, da die Bestellungen und Mahlzeiten an der Essensausgabe des Restaurants (`restaurant.meal`) abgewickelt werden. Die `run()`-Methode der `WaitPerson`-Aufgabe ruft die `wait()`-Methode auf, um erst durch die von der `Chef`-Aufgabe aufgerufene `notifyAll()`-Methode wieder „aufgeweckt“ zu werden. In diesem sehr einfachen Beispiel wissen wir, daß nur eine Aufgabe auf die Sperre des `WaitPerson`-Objektes wartet, nämlich die `WaitPerson`-Aufgabe selbst. Es ist daher theoretisch möglich, `notify()` statt `notifyAll()` aufzurufen. In komplexeren Situationen warten mehrere Aufgaben auf die Sperre eines bestimmten Objektes und es steht nicht fest, welche Aufgabe benachrichtigt werden soll. Es ist deshalb sicherer, die `notifyAll()`-Methode zu wählen, welche alle Aufgaben „aufweckt“, die auf diese Sperre warten. Anschließend muß jede Aufgabe für sich selbst entscheiden, ob die Benachrichtigung relevant ist.

[217] Hat der Koch eine Mahlzeit zubereitet und den Kellner benachrichtigt, so wartet er darauf, daß der Kellner die Mahlzeit abholt, eine neue Bestellung aufnimmt und ihn benachrichtigt, woraufhin der Koch die nächste Mahlzeit zubereitet.

[218] Beachten Sie, daß die `wait()`-Methode im Körper einer `while`-Schleife aufgerufen wird, deren Abbruchbedingung mit der Ursache des Wartezustandes identisch ist. *This seems a bit strange at first — if you're waiting for an order, once you wake up, the order must be available, right?* Bei einer threadbasierten Anwendung kann zwischenzeitlich eine andere Aufgabe zur Verarbeitung ausgewählt werden und die Bestellung aufgreifen, während die `WaitPerson`-Aufgabe „aufwacht". Der einzige sichere Ansatz besteht darin, beim Aufrufen der `wait()`-Methode *ausnahmslos* die folgende Schreibweise zu gebrauchen (entsprechende Synchronisierung und Vorkehrungen gegen den Verlust von Benachrichtigungen vorausgesetzt):

```
while (conditionIsNotMet)
    wait();
```

Diese Notation garantiert, daß die Bedingung tatsächlich erfüllt ist, wenn die Schleife beendet wird. Falls die wartende Aufgabe dagegen nicht im Zusammenhang mit dieser Bedingung benachrichtigt wurde (bei `notifyAll()` möglich), oder sich die Bedingung vor dem Verlassen der Schleife geändert hat, wird die Aufgabe garantiert wieder in den Wartezustand zurückversetzt.

[219] Der Aufruf der `notifyAll()`-Methode setzt die Sperrung des von `waitPerson` referenzierten Objektes voraus. Dies ist möglich, da die `wait()`-Methode in der `run()`-Methode von `WaitPerson` die Sperre aufhebt. Da die Sperre akquiriert werden muß, um `notifyAll()` aufrufen zu können, ist garantiert, daß beide Aufgaben die `notifyAll()`-Methode desselben Objektes aufrufen, sich nicht gegenseitig auf die Füße treten.

[220] In beiden `run()`-Methoden ist der Methodenkörper in einem `try`-Block eingeschlossen, um die geordnete Rückkehr aus den Methode zu gewährleisten. Die `catch`-Klauseln enden unmittelbar vor der schließenden Klammer der jeweiligen `run()`-Methode. Wird während der Verarbeitung einer der beiden `run()`-Methoden eine Ausnahme vom Typ `InterruptedException` ausgeworfen, so wird die Verarbeitung der Methode unmittelbar nach der Ausnahmebehandlung beendet.

[221] *In Chef, note that after calling shutdownNow() you could simply return from run(), and normally that's what you should do. However, it's a little more interesting to do it this way.* Die `shutdownNow()`-Methode ruft die `interrupt()`-Methode jedes von diesem Exekutor gestarteten Threads auf. Die `Chef`-Aufgabe wird allerdings nicht unmittelbar nach dem `interrupt()`-Aufruf beendet, da die Unterbrechung nur dann eine Ausnahme vom Typ `InterruptedException` hervorruft, wenn die Aufgabe in eine unterbrechbare blockierende Operation eintritt. Daher wird zuerst die Meldung „Order up!" ausgegeben. Die Ausnahme wird anschließend ausgeworfen, wenn die `Chef`-Aufgabe versucht die `sleep()`-Methode aufzurufen. Wenn Sie den Aufruf der `sleep()`-Methode auskommentieren, kehrt die Aufgabe zunächst zum Kopf der `while`-Schleife zurück und verläßt die `run()`-Methode aufgrund der Abbruchbedingung `Thread.interrupt()`, ohne eine Ausnahme auszuwerfen.

[222] Im obigen Beispiel gibt es nur eine Stelle, an der eine Aufgabe ein Objekt „deponiert" und an der es später von einer anderen Aufgabe abgeholt wird. In einem typischen Erzeuger/Verbraucher-System werden die produzierten Objekte in einer Warteschlange („first in, first out") gespeichert und auch wieder von dort abgeholt. Wir besprechen diese Warteschlangen in Unterabschnitt 22.5.4.

Übungsaufgabe 24: (1) Lösen Sie ein Erzeuger/Verbraucher-Problem mit einem Erzeuger und einem Verbraucher mit Hilfe von `wait()` und `notifyAll()`. Der Erzeuger darf den Puffer des Verbrauchers nicht überfüllen (dies ist möglich, wenn der Erzeuger schneller arbeitet als der Verbraucher). Ist der Verbraucher schneller als der Erzeuger, so dürfen dieselben Daten nicht mehrfach gelesen werden. Treffen Sie keine Annahmen über die Geschwindigkeiten von Erzeuger und Verbraucher. ■

Übungsaufgabe 25: (1) Ändern Sie die Klasse `Chef` im Beispiel *Restaurant.java*, so daß die Programmausführung nach dem Aufrufen der `shutdownNow()`-Methode aus der `run()`-Methode zurückkehrt und beobachten Sie das geänderte Verhalten. ■

Übungsaufgabe 26: (8) Ergänzen Sie das Beispiel *Restaurant.java* um eine Klasse `BusBoy` (Bedienungshilfe, Hilfskraft im Restaurant). Der Kellner benachrichtigt den `BusBoy`, wenn der Platz aufgeräumt werden soll. ■

22.5.3.1 Verwendung expliziter Lock- und Condition-Objekte

[223] Die `java.util.concurrent`-Bibliothek der SE 5 enthält einige Klassen, mit denen wir das Beispiel *WaxOMatic.java* umschreiben wollen. Die Basisklasse *`Condition`* ist mit einem Sperrobjekt verknüpft und gestattet, die Verarbeitung einer Aufgabe per `await()`-Methode zu suspendieren. Nach einer Änderung des externen Zustandes, infolge dessen die Verarbeitung einer suspendierten Aufgabe eventuell fortgesetzt werden kann, lassen sich entweder per `signal()`-Methode nur eine oder aber per `signalAll()`-Methode alle bezüglich eines *`Condition`*-Objektes suspendierten Aufgaben benachrichtigen (wie bei `notifyAll()` ist `signalAll()` die sicherere Variante).

[224] Die überarbeitete Version des Beispiels *WaxOMatic.java* enthält ein *`Condition`*-Objekt, welches zur Suspendierung von Aufgaben in den Methoden `waitForWaxing()` und `waitForBuffing()` verwendet wird:

```
//: concurrency/waxomatic2/WaxOMatic2.java
// Using Lock and Condition objects.
package concurrency.waxomatic2;
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import static net.mindview.util.Print.*;

class Car {
    private Lock lock = new ReentrantLock();
    private Condition condition = lock.newCondition();
    private boolean waxOn = false;
    public void waxed() {
        lock.lock();
        try {
            waxOn = true; // Ready to buff
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void buffed() {
        lock.lock();
        try {
            waxOn = false; // Ready for another coat of wax
            condition.signalAll();
        } finally {
            lock.unlock();
        }
    }
    public void waitForWaxing() throws InterruptedException {
        lock.lock();
        try {
            while(waxOn == false)
                condition.await();
        } finally {
            lock.unlock();
```

```
        }
    }
    public void waitForBuffing() throws InterruptedException{
        lock.lock();
        try {
            while(waxOn == true)
                condition.await();
        } finally {
            lock.unlock();
        }
    }
}

class WaxOn implements Runnable {
    private Car car;
    public WaxOn(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                printnb("Wax On! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.waxed();
                car.waitForBuffing();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax On task");
    }
}

class WaxOff implements Runnable {
    private Car car;
    public WaxOff(Car c) { car = c; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                car.waitForWaxing();
                printnb("Wax Off! ");
                TimeUnit.MILLISECONDS.sleep(200);
                car.buffed();
            }
        } catch(InterruptedException e) {
            print("Exiting via interrupt");
        }
        print("Ending Wax Off task");
    }
}

public class WaxOMatic2 {
    public static void main(String[] args) throws Exception {
        Car car = new Car();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new WaxOff(car));
        exec.execute(new WaxOn(car));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* Output: (90% match)
```

```
    Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
    Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
    Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off! Wax On! Wax Off!
    Wax On! Exiting via interrupt
    Ending Wax Off task
    Exiting via interrupt
    Ending Wax On task
*///:~
```

[225] Im Konstruktor der Klasse `Car` liefert ein *`Lock`*-Objekt eine Referenz auf ein *`Condition`*-Objekt, mit dessen Hilfe die Kommunikation zwischen den Aufgaben bewerkstelligt wird. Das *`Condition`*-Objekt enthält allerdings selbst keine Informationen über den Zustand des Wachsen/Polieren-Vorgangs. Der Zustand des Vorgangs wird durch das zusätzliche `boolean`-Feld `waxOn` abgebildet.

[226] Jedem Aufruf der `lock()`-Methode folgt unmittelbar eine `try`/`finally`-Kombination, um zu gewährleisten, daß die Sperre in jedem Fall aufgehoben wird. Wie beim „eingebauten“ `wait()`/`notify()`/`notifyAll()`-Mechanismus muß eine Aufgabe das Sperrobjekt erst sperren, bevor eine der Methoden `await()`, `signal()` oder `signalAll()` des Bedingungsobjektes aufgerufen werden kann.

[227] Diese Lösung ist komplizierter als die vorige Version (Seite 927). Beachten Sie, daß die zusätzliche Komplexität hier keinen Vorteil mit sich bringt. Die Sperr- und Bedingungsobjekt sind nur bei schwierigeren Problemen der Threadprogrammierung notwendig.

Übungsaufgabe 27: (2) Ändern Sie das Beispiel *Restaurant.java* so, daß es mit *`Lock`*- und *`Condition`*-Objekten arbeitet. ■

22.5.4 Erzeuger/Verbraucher-Systeme mit Warteschlange

[228] Die Methoden `wait()` und `notifyAll()` gestatten die Kooperation von Aufgaben auf einer relativ niedrigen Ebene durch Quittierung jeder einzelnen Interaktion. In vielen Fällen läßt sich das Problem von einer höheren Abstraktionsebene aus lösen, indem die Kooperation über eine *synchronisierten Warteschlange* abgewickelt wird, welche stets nur einer Aufgabe erlaubt, ein Element hinzuzufügen oder zu entnehmen. Das Interface *`java.util.concurrent.BlockingQueue`* repräsentiert eine synchronisierte Warteschlange und hat mehrere Standardimplementierungen. In der Regel wählen Sie die Klasse `LinkedBlockingQueue`, welche eine Warteschlange mit unbeschränkter Anzahl von Elementen darstellt. Die Klasse `ArrayBlockingQueue` repräsentiert dagegen eine Warteschlange fester Größe und blockiert, wenn ihre Kapazität erschöpft ist.

[229] Diese Warteschlangen suspendieren den Verbraucher, falls dieser versucht, ein Element aus einer leeren Warteschlange zu entnehmen und setzen die Verarbeitung fort, wenn neue Elemente verfügbar geworden sind. Blockierende Warteschlangen gestatten, eine bemerkenswerte Anzahl von Problemen erheblich einfacher und verläßlicher zu lösen, als `wait()` und `notify()`.

[230] Das folgende Beispiel sequentialisiert die Verarbeitung von `LiftOff`-Objekten. Die Klasse `LiftOffRunner` stellt den Verbraucher dar, welcher ein `LiftOff`-Objekt nach dem anderen aus der synchronisierten Warteschlange entnimmt und direkt startet. („Direkt starten“ bedeutet, daß der Thread, der die `LiftOffRunner`-Aufgabe verarbeitet, die `run()`-Methode des `LiftOff`-Objektes selbst aufruft, statt einen neuen Thread zu starten.)

```
//: concurrency/TestBlockingQueues.java
// {RunByHand}
import java.util.concurrent.*;
import java.io.*;
import static net.mindview.util.Print.*;
```

```
class LiftOffRunner implements Runnable {
    private BlockingQueue<LiftOff> rockets;
    public LiftOffRunner(BlockingQueue<LiftOff> queue) {
        rockets = queue;
    }
    public void add(LiftOff lo) {
        try {
            rockets.put(lo);
        } catch(InterruptedException e) {
            print("Interrupted during put()");
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                LiftOff rocket = rockets.take();
                rocket.run(); // Use this thread
            }
        } catch(InterruptedException e) {
            print("Waking from take()");
        }
        print("Exiting LiftOffRunner");
    }
}

public class TestBlockingQueues {
    static void getkey() {
        try {
            // Compensate for Windows/Linux difference in the
            // length of the result produced by the Enter key:
            new BufferedReader(new InputStreamReader(System.in)).readLine();
        } catch(java.io.IOException e) {
            throw new RuntimeException(e);
        }
    }
    static void getkey(String message) {
        print(message);
        getkey();
    }
    static void
        test(String msg, BlockingQueue<LiftOff> queue) {
        print(msg);
        LiftOffRunner runner = new LiftOffRunner(queue);
        Thread t = new Thread(runner);
        t.start();
        for(int i = 0; i < 5; i++)
            runner.add(new LiftOff(5));
        getkey("Press 'Enter' (" + msg + ")");
        t.interrupt();
        print("Finished " + msg + " test");
    }
    public static void main(String[] args) {
        test("LinkedBlockingQueue", // Unlimited size
             new LinkedBlockingQueue<LiftOff>());
        test("ArrayBlockingQueue", // Fixed size
             new ArrayBlockingQueue<LiftOff>(3));
        test("SynchronousQueue", // Size of 1
             new SynchronousQueue<LiftOff>());
```

```
    }
} ///:~
```

Die `LiftOff`-Objekte werden von der `main()`-Methode in der Warteschlange (dem *`BlockingQueue`*-Objekt) gespeichert und von der `LiftOffRunner`-Aufgabe aus der Warteschlange entnommen. Beachten Sie, daß sich die Klasse `LiftOffRunner` nicht um die Synchronisierung kümmern muß, da diese Probleme bereits durch das *`BlockingQueue`*-Objekt gelöst sind.

Übungsaufgabe 28: (3) Ändern Sie das Beispiel *TestBlockingQueues.java* in dem Sie eine weitere Aufgabenklasse anlegen, die anstelle der `main()`-Methode die `LiftOff`-Objekte in die Warteschlange einträgt. ■

22.5.4.1 Die Toastbrotmaschine: Ein Beispiel für blockierende Warteschlangen

[231] Im folgenden Beispiel betrachten wir eine Maschine mit drei Aufgaben als Beispiel für eine blockierende Warteschlange: Ein Toastbrot toasten, das getoastete Brot mit Butter und danach mit Marmelade zu bestreichen. Das Toastbrot wird während der Verarbeitung in verschiedenen blockierenden Warteschlangen zwischengespeichert:

```
//: concurrency/ToastOMatic.java
// A toaster that uses queues.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Toast {
    public enum Status { DRY, BUTTERED, JAMMED }
    private Status status = Status.DRY;
    private final int id;
    public Toast(int idn) { id = idn; }
    public void butter() { status = Status.BUTTERED; }
    public void jam() { status = Status.JAMMED; }
    public Status getStatus() { return status; }
    public int getId() { return id; }
    public String toString() {
        return "Toast " + id + ": " + status;
    }
}

class ToastQueue extends LinkedBlockingQueue<Toast> {}

class Toaster implements Runnable {
    private ToastQueue toastQueue;
    private int count = 0;
    private Random rand = new Random(47);
    public Toaster(ToastQueue tq) { toastQueue = tq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(100 + rand.nextInt(500));
                // Make toast
                Toast t = new Toast(count++);
                print(t);
                // Insert into queue
                toastQueue.put(t);
            }
        } catch(InterruptedException e) {
```

```
            print("Toaster interrupted");
        }
        print("Toaster off");
    }
}

// Apply butter to toast:
class Butterer implements Runnable {
    private ToastQueue dryQueue, butteredQueue;
    public Butterer(ToastQueue dry, ToastQueue buttered) {
        dryQueue = dry;
        butteredQueue = buttered;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = dryQueue.take();
                t.butter();
                print(t);
                butteredQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Butterer interrupted");
        }
        print("Butterer off");
    }
}

// Apply jam to buttered toast:
class Jammer implements Runnable {
    private ToastQueue butteredQueue, finishedQueue;
    public Jammer(ToastQueue buttered, ToastQueue finished) {
        butteredQueue = buttered;
        finishedQueue = finished;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = butteredQueue.take();
                t.jam();
                print(t);
                finishedQueue.put(t);
            }
        } catch(InterruptedException e) {
            print("Jammer interrupted");
        }
        print("Jammer off");
    }
}

// Consume the toast:
class Eater implements Runnable {
    private ToastQueue finishedQueue;
    private int counter = 0;
    public Eater(ToastQueue finished) {
        finishedQueue = finished;
    }
```

```
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until next piece of toast is available:
                Toast t = finishedQueue.take();
                // Verify that the toast is coming in order,
                // and that all pieces are getting jammed:
                if(t.getId() != counter++ ||
                   t.getStatus() != Toast.Status.JAMMED) {
                    print(">>>> Error: " + t);
                    System.exit(1);
                } else
                    print("Chomp! " + t);
            }
        } catch(InterruptedException e) {
            print("Eater interrupted");
        }
        print("Eater off");
    }
}

public class ToastOMatic {
    public static void main(String[] args) throws Exception {
        ToastQueue dryQueue = new ToastQueue(),
            butteredQueue = new ToastQueue(),
            finishedQueue = new ToastQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(new Toaster(dryQueue));
        exec.execute(new Butterer(dryQueue, butteredQueue));
        exec.execute(new Jammer(butteredQueue, finishedQueue));
        exec.execute(new Eater(finishedQueue));
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* (Execute to see output) *////:~
```

Die Klasse `Toast` ist ein hervorragendes Beispiel für den praktischen Wert von Aufzählungstypen. Beachten Sie, daß es keine explizite Synchronisierung (durch *`Lock`*-Objekte oder `synchronized`) gibt. Die Synchronisierung wird implizit durch die (intern synchronisierten) Warteschlangen und das Design der Anwendung bewerkstelligt (ein `Toast`-Objekt wird stets von höchstens einer Aufgabe beansprucht). Aufgrund der Blockierungsfähigkeit der Warteschlangen werden die beteiligten Arbeitsschritte automatisch suspendiert beziehungsweise wieder aufgenommen. Sie sehen, daß die Verwendung blockierender Warteschlangen den Quelltext dramatisch vereinfachen kann. Die per `wait()` und `notifyAll()` implementierte Kopplung zwischen den Klassen fällt weg, weil jede Klasse nur mit ihren blockierenden Warteschlangen kommuniziert.

Übungsaufgabe 29: (8) Ändern Sie das Beispiel *ToastOMatic.java* so, daß die Maschine auch Toasts mit Erdnußbutter oder Gelee herstellen kann, indem Sie die Fertigung verzweigen (ein Zweig für Erdnußbutter und einer für Gelee) und schließlich wieder zusammenführen. ■

22.5.5 Kommunikation zwischen Aufgaben über Pipes

[232] Es ist häufig nützlich, daß Aufgaben über das Ein-/Ausgabesystem miteinander kommunizieren können. Threadbibliotheken können die Kommunikation zwischen Aufgaben über das Ein-/Ausgabesystem mit Hilfe von Pipes unterstützen. Diese existieren in der Ein-/Ausgabebibliothek

von Java in Gestalt der Klassen `java.io.PipedWriter` (gestattet einer Aufgabe, in eine Pipe zu schreiben) und `java.io.PipedReader` (gestattet einer Aufgabe aus derselben Pipe zu lesen). Sie können sich die Situation als Variante des Erzeuger/Verbraucher-Problems vorstellen, wobei die Pipe die Rolle des Bezugsobjektes spielt. Eine Pipe ist im Grunde genommen eine blockierende Warteschlange und existierte bereits in den Java-Versionen vor Einführung des Interfaces *`BlockingQueue`*.

[233] Im folgenden einfachen Beispiel kommunizieren zwei Aufgaben über eine Pipe:

```
//: concurrency/PipedIO.java
// Using pipes for inter-task I/O
import java.util.concurrent.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Sender implements Runnable {
    private Random rand = new Random(47);
    private PipedWriter out = new PipedWriter();
    public PipedWriter getPipedWriter() { return out; }
    public void run() {
        try {
            while(true)
                for(char c = 'A'; c <= 'z'; c++) {
                    out.write(c);
                    TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                }
        } catch(IOException e) {
            print(e + " Sender write exception");
        } catch(InterruptedException e) {
            print(e + " Sender sleep interrupted");
        }
    }
}

class Receiver implements Runnable {
    private PipedReader in;
    public Receiver(Sender sender) throws IOException {
        in = new PipedReader(sender.getPipedWriter());
    }
    public void run() {
        try {
            while(true) {
                // Blocks until characters are there:
                printnb("Read: " + (char)in.read() + ", ");
            }
        } catch(IOException e) {
            print(e + " Receiver read exception");
        }
    }
}

public class PipedIO {
    public static void main(String[] args) throws Exception {
        Sender sender = new Sender();
        Receiver receiver = new Receiver(sender);
        ExecutorService exec = Executors.newCachedThreadPool();
        exec.execute(sender);
        exec.execute(receiver);
```

```
            TimeUnit.SECONDS.sleep(4);
            exec.shutdownNow();
        }
    } /* Output: (65% match)
          Read: A, Read: B, Read: C, Read: D, Read: E, Read: F, Read: G, Read: H,
          Read: I, Read: J, Read: K, Read: L, Read: M,
          java.lang.InterruptedException: sleep interrupted Sender sleep interrupted
          java.io.InterruptedIOException Receiver read exception
       *///:~
```

Die Klassen `Sender` und `Receiver` repräsentieren die beiden Aufgaben, die miteinander kommunizieren müssen. In `Sender` wird ein eigenständiges `PipedWriter`-Objekt erzeugt. In `Receiver` wird ein `PipedReader`-Objekt erzeugt und mit dem `PipedWriter`-Objekt aus `Sender` verknüpft. Die `Sender`-Aufgabe übergibt dem `PipedWriter`-Objekt ein Zeichen und „schläft“ anschließend für eine per Zufall bestimmte Zeitspanne. Dennoch benötigt die Klasse `Receiver` weder `wait()` noch `sleep()`. Die `read()`-Methode blockiert, wenn beim Lesen die Daten ausgehen.

[234] Beachten Sie, daß die Verarbeitung der `Sender`- und die `Receiver`-Aufgaben in der `main()`-Methode erst beginnt, nachdem die Konstruktion der Objekte abgeschlossen ist. Wenn Sie unvollständig konstruierte Objekte starten, kann sich die Pipe auf unterschiedlichen Plattformen inkonsistent verhalten. (Beachten Sie, daß blockierende Warteschlangen robuster und einfacher zu handhaben sind.)

[235] Beim Aufrufen der `shutdownNow()`-Methode zeigt sich ein wesentlicher Unterschied zwischen `PipedReader` und einer gewöhnlichen Ein-/Ausgabeoperation. Die `read()`-Methode der Klasse `PipedReader` ist unterbrechbar, während die `interrupt()`-Methode beispielsweise bei `System.in.read()` anstelle von `in.read()` nicht in der Lage ist, die aufgerufene `read()`-Methode zu unterbrechen.

Übungsaufgabe 30: (1) Ändern Sie das Beispiel *PipedIO.java*, so daß statt der Pipe eine blockierende Warteschlange verwendet wird. ■

22.6 Verklemmungen (Deadlocks)

[236] Sie verstehen nun, daß ein Objekt synchronisierte Methoden oder eine andere Form von Sperrmechanismus implementieren kann, um Aufgaben den Zugriff solange zu verwehren, bis die Sperre aufgehoben wird. Sie haben außerdem gelernt, daß die Verarbeitung einer Aufgabe blockiert werden kann. Es ist daher möglich, daß die Verarbeitung einer Aufgabe steckenbleibt, während sie auf eine zweite Aufgabe wartet, die wiederum auf eine dritte Aufgabe wartet und so weiter, wobei die letzte Aufgabe dieser Kette auf die erste wartet. Sie erhalten eine ununterbrochene Verkettung von Aufgaben, die aufeinander warten, wobei sich aber keine „bewegen“ kann. Diese Situation wird als Verklemmung (*deadlock*) bezeichnet.[23]

[237] Stellt sich die Verklemmung sofort nach dem Programmstart ein, so können Sie unverzüglich mit der Fehlersuche beginnen. Das eigentliche Problem besteht aber dann, wenn Ihr Programm scheinbar fehlerlos arbeitet, aber eine verborgene Möglichkeit für eine Verklemmung existiert. In diesem Fall gibt es keine Anzeichen für die Gefahr einer Verklemmung und der Makel bleibt solange unentdeckt, bis er bei Ihrem Kunden unerwartet in Erscheinung tritt (auf eine mit an Sicherheit grenzender Wahrscheinlichkeit nicht reproduzierbare Weise). Vorbeugung durch gewissenhaftes Design ist im Hinblick auf Verklemmungen ein wesentlicher Teil der Entwicklung einer threadbasierten

[23] Das Livelock ist eine Variante der Verklemmung bei der zwei oder mehr Aufgaben zwar in der Lage sind, ihre Zustände zu verändern (also nicht blockiert sind), aber dennoch keinen sinnvollen Fortschritt zustande bringen.

Anwendung.

[238] Das *Philosophenproblem* (*dining philosophers problem*) von Edsger Dijkstra ist das klassische Beispiel für eine Verklemmung. Die einfachste Variante umfaßt fünf Philosophen (das Beispiel in diesem Abschnitt gestattet eine frei wählbare Anzahl). Die Philosophen verbringen einen Teil ihrer Zeit mit Denken und einen weiteren Teil mit Essen. Während des Denkens benötigen sie keine gemeinsam genutzten Resourcen, wohl aber beim Essen. In der ursprünglichen Beschreibung des Problems sind diese gemeinsam genutzten Resourcen Gabeln und ein Philosoph braucht zwei Gabeln, um Spaghetti aus einer Schlüssel in der Mitte des Tischs zu entnehmen. Wir ersetzen in unserem Beispiel die Gabeln durch Eßstäbchen, da ein Philosoph offensichtlich zwei Stäbchen benötigt, um essen zu können.

[239] Da die Philosophen nur wenig Geld haben, können sie sich nur fünf Eßstäbchen leisten (allgemeiner formuliert, die Anzahl der Stäbchen entspricht der Anzahl der Philosophen). Die Stäbchen sind auf die Plätze am Tisch verteilt. Wenn ein Philosoph essen möchte, muß er die beiden Stäbchen links und rechts seines Tellers in die Hand nehmen. Verwendet einer der benachbarten Philosophen gerade eines der Stäbchen, so muß unser Philosoph warten, bis das Stäbchen wieder zur Verfügung steht.

```
//: concurrency/Chopstick.java
// Chopsticks for dining philosophers.

public class Chopstick {
    private boolean taken = false;
    public synchronized void take() throws InterruptedException {
        while(taken)
            wait();
        taken = true;
    }
    public synchronized void drop() {
        taken = false;
        notifyAll();
    }
} ///:~
```

Es ist nicht erlaubt, daß zwei Philosophen (`Philosopher`-Objekte) ein und dasselbe Stäbchen (`Chopstick`-Objekt) zugleich nehmen. Außerdem kann ein Philosoph auf ein im Augenblick nicht verfügbares Stäbchen warten (dessen `wait()`-Methode aufrufen), bis dessen Besitzer es wieder zurücklegt (die `drop()`-Methode des Stäbchens aufruft).

[240] Ruft ein Philosoph die `take()`-Methode eines Stäbchens auf, so wartet er, bis das Flag `taken` auf `false` zurückgesetzt wird (bis der Philosoph der das Stäbchen im Augenblick benutzt es zurücklegt). Anschließend wird das Flag auf `true` gesetzt, um anzuzeigen, daß das Stäbchen nun von einem anderen Philosophen verwendet wird. Beendet dieser Philosoph seine Mahlzeit, so ruft er die `drop()`-Methode auf, um das Flag umzuschalten und per `notifyAll()` die Kollegen zu benachrichtigen, die eventuell auf das Stäbchen warten.

```
//: concurrency/Philosopher.java
// A dining philosopher
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class Philosopher implements Runnable {
    private Chopstick left;
    private Chopstick right;
    private final int id;
```

```
    private final int ponderFactor;
    private Random rand = new Random(47);
    private void pause() throws InterruptedException {
        if(ponderFactor == 0) return;
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(ponderFactor * 250));
    }
    public Philosopher(Chopstick left, Chopstick right,
        int ident, int ponder) {
        this.left = left;
        this.right = right;
        id = ident;
        ponderFactor = ponder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                print(this + '' '' + ''thinking'');
                pause();
                // Philosopher becomes hungry
                print(this + '' '' + ''grabbing right'');
                right.take();
                print(this + '' '' + ''grabbing left'');
                left.take();
                print(this + '' '' + ''eating'');
                pause();
                right.drop();
                left.drop();
            }
        } catch(InterruptedException e) {
            print(this + '' '' + ''exiting via interrupt'');
        }
    }
    public String toString() { return ''Philosopher '' + id; }
} ///:~
```

[241] In der `Philosopher`-Methode `run()` übt der Philosoph zwei Tätigkeiten aus: Denken und Essen. Die Methode `pause()` ruft die `sleep()`-Methode mit einem Zufallswert auf, sofern der Grübelfaktor `ponderFactor` nicht Null ist. Ein Philosoph denkt also für einen zufällig bestimmten Zeitraum nach, versucht mittels `take()` das linke und rechte Eßstäbchen aufzunehmen, ißt für einen wiederum zufällig bestimmten Zeitraum, legt die Stäbchen zurück beginnt diesen Zyklus erneut.

[242] Die erste Version unseres Programms verfängt sich in einer Verklemmung:

```
//: concurrency/DeadlockingDiningPhilosophers.java
// Demonstrates how deadlock can be hidden in a program.
// {Args: 0 5 timeout}
import java.util.concurrent.*;

public class DeadlockingDiningPhilosophers {
    public static void main(String[] args) throws Exception {
        int ponder = 5;
        if(args.length > 0)
            ponder = Integer.parseInt(args[0]);
        int size = 5;
        if(args.length > 1)
            size = Integer.parseInt(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Chopstick[] sticks = new Chopstick[size];
```

```
            for(int i = 0; i < size; i++)
                sticks[i] = new Chopstick();
            for(int i = 0; i < size; i++)
                exec.execute(new Philosopher(
                    sticks[i], sticks[(i+1) % size], i, ponder));
            if(args.length == 3 && args[2].equals("timeout"))
                TimeUnit.SECONDS.sleep(5);
            else {
                System.out.println("Press 'Enter' to quit");
                System.in.read();
            }
            exec.shutdownNow();
        }
    } /* (Execute to see output) *////:~
```

Wenn die Philosophen sehr wenig Zeit mit Denken verbringen, wetteifern alle um die Stäbchen, während sie versuchen zu essen und es kommt schnell zu einer Verklemmung.

[243] Das erste Kommandozeilenargument ist der Grübelfaktor (`ponderFactor`), der die Länge der Zeitspanne angibt, die jeder Philosoph zum Denken verwendet. Bei vielen Philosophen oder langer Denkzeit kommt es möglicherweise niemals zu einer Verklemmung, obwohl die Möglichkeit besteht. Wenn Sie eine Null als erstes Kommandozeilenargument übergeben, tritt die Verklemmung dagegen schnell ein.

[244] Beachten Sie, daß die `Chopstick`-Objekte keine internen Bezeichnungen brauchen, da sie über ihre Position in dem von `sticks` referenzierten `Chopstick`-Array identifiziert werden können. Der Konstruktor der Klasse `Philosopher` erwartet je eine Referenz auf das linke und das rechte `Chopstick`-Objekt. Die Philosophen werden (mit Ausnahme des letzten) zwischen den Eßstäbchen platziert. Der letzte Philosoph wird so platziert, daß das nullte Stäbchen neben seiner rechten Hand liegt. Der Tisch ist rund und der letzte Philosoph sitzt neben dem ersten, so daß sich beide das nullte Stäbchen teilen. Es ist nun möglich, daß alle Philosophen essen wollen und jeder darauf wartet, daß sein Tischnachbar sein Stäbchen zurücklegt. Das Programm bleibt stecken.

[245] Verbringen die Philosophen mehr Zeit mit Denken als mit Essen, so sind Zugriffe auf die gemeinsam genutzten Resourcen (Stäbchen) weniger wahrscheinlich. Sie können sich davon überzeugen, daß das Programm verklemmungsfrei arbeitet (indem Sie einen Grübelfaktor größer als Null verwenden), obwohl die Möglichkeit einer Verklemmung vorhanden ist. Dieses Beispiel ist genau deshalb interessant, weil es zeigt, daß ein Programm scheinbar korrekt funktionieren kann, wobei aber tatsächlich die Gefahr einer Verklemmung besteht.

[246] Um das Problem beheben zu können, müssen Sie verstehen, daß eine Verklemmung eintritt, wenn die folgenden vier Voraussetzung gleichzeitig erfüllt sind:

- Gegenseitiger Ausschluß. Mindestens eine von den Aufgaben verwendete Resource muß darf nicht gemeinsam benutzbar sein. (Ein Eßstäbchen kann stets nur von höchstens einem Philosophen benutzt werden.)
- Mindestens eine Aufgabe muß eine Resource beanspruchen und zugleich auf eine von einer anderen Aufgabe beanspruchte Resource warten. (Das Eintreten einer Verklemmung setzt voraus, daß ein Philosoph ein Stäbchen in die Hand genommen hat und auf das zweite Stäbchen wartet.)
- A resource cannot be preemptively taken away from a task. Tasks only release resources as a normal event. (Die Philosophen sind höflich und nehmen keinem Kollegen sein Stäbchen weg.)
- Zirkuläres Warten ist möglich, das heißt eine Aufgabe wartet auf eine von einer zweiten Aufga-

be beanspruchte Resource, welche wiederum auf eine von einer dritten Aufgabe beanspruchte Resource wartet und so weiter, bis eine Aufgabe in dieser Kette auf die von der ersten Aufgabe beanspruchte Resource wartet, so daß völliger Stillstand eintritt. (Im Beispiel *DeadlockingDiningPhilosophers.java* ergibt sich das zirkuläre Warten dadurch, daß jeder Philosoph zuerst das rechte Stäbchen in die Hand nimmt und dann auf das linke Stäbchen wartet.)

[247] Da alle vier Bedingungen erfüllt sein müssen, um eine Verklemmung zu verursachen, genügt es, eine davon zu verhindern, um die Verklemmung zu vermeiden. Beim Philosophenproblem ist die vierte Bedingung der leichteste Weg. Das zirkuläre Warten wird dadurch ermöglicht, daß jeder Philosoph seine Stäbchen in einer bestimmten Reihenfolge aufzunehmen versucht: erst rechts, dann links. Dadurch kann sich die Situation ergeben, daß jeder Philosoph sein rechtes Stäbchen in die Hand nimmt und auf das linke Stäbchen wartet. Wird aber nun der letzte Philosoph so „initialisiert“, daß er erst das linke und dann das rechte Stäbchen aufnimmt, so kann dieser Philosoph seinen rechten Kollegen nicht mehr daran hindern, ihr gemeinsames Stäbchen aufzunehmen und die Gefahr des zirkulären Wartens ist gebannt. Dies ist nur eine Möglichkeit zur Lösung des Philosophenproblems. Sie können das Problem auch lösen, indem Sie eine der anderen Voraussetzungen vermeiden (siehe Bücher über fortgeschrittene Threadprogrammierung):

```
//: concurrency/FixedDiningPhilosophers.java
// Dining philosophers without deadlock.
// {Args: 5 5 timeout}
import java.util.concurrent.*;

public class FixedDiningPhilosophers {
    public static void main(String[] args) throws Exception {
        int ponder = 5;
        if(args.length > 0)
            ponder = Integer.parseInt(args[0]);
        int size = 5;
        if(args.length > 1)
            size = Integer.parseInt(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Chopstick[] sticks = new Chopstick[size];
        for(int i = 0; i < size; i++)
            sticks[i] = new Chopstick();
        for(int i = 0; i < size; i++)
            if(i < (size-1))
                exec.execute(new Philosopher(sticks[i], sticks[i+1], i, ponder));
            else
                exec.execute(new Philosopher(sticks[0], sticks[i], i, ponder));
        if(args.length == 3 && args[2].equals("timeout"))
            TimeUnit.SECONDS.sleep(5);
        else {
            System.out.println("Press 'Enter' to quit");
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* (Execute to see output) *////:~
```

Indem wir dafür sorgen, daß der letzte Philosoph das linke Stäbchen vor dem rechten aufnimmt beziehungsweise zurücklegt, wird das Verklemmungspotential eliminiert und das Programm läuft reibungslos.

[248] Die Programmiersprache unterstützt das Vermeiden von Verklemmungen nicht. Dies sind wenig tröstende Worte für jemanden, der die Ursache einer Verklemmung sucht

Übungsaufgabe 31: (8) Ändern Sie das Beispiel *DeadlockingDiningPhilosophers.java* so, daß ein Philosoph seine Stäbchen nach dem Essen in einen Behälter legt. Möchte ein Philosoph essen, so nimmt er die nächstbesten beiden Stäbchen aus dem Behälter. Ist die Gefahr einer Verklemmung damit beseitigt? Läßt sich eine Verklemmung herbeiführen, indem Sie einfach die Anzahl der Stäbchen reduzieren? ■

22.7 Die Synchronisierungsklassen in java.util.concurrent (Auswahl)

[249] Das seit der SE 5 vorhandene Package `java.util.concurrent` umfaßt eine beachtliche Anzahl neuer Klassen, die zur Lösung von Problemen in der Threadprogrammierung entwickelt wurden. Wenn Sie den Umgang mit diesen Klassen erlernen, sind Sie in der Lage, einfachere und robustere threadbasierte Programme zu schreiben.

[250] Dieser Abschnitt dokumentiert eine repräsentative Auswahl dieser Klassen anhand von Beispielen. Einige Klassen, die Sie wahrscheinlich weniger häufig verwenden oder vorfinden werden, werden an dieser Stelle nicht diskutiert.

[251] Da jede Klasse für ein anderes Problem gedacht ist, gibt es keinen roten Faden, um die Beispiele anzuordnen. Ich habe daher versucht, mit den einfacheren Beispielen zu beginnen und die Beispiele nach zunehmender Komplexität zu sortieren.

22.7.1 CountDownLatch

[252] Die Klasse `CountDownLatch` dient der Synchronisierung einer oder mehrerer Aufgaben, in dem sie diese zwingt, auf die Verarbeitung einer Anzahl von Anweisungen aus anderen Aufgaben zu warten.

[253] Ein `CountDownLatch`-Objekt wird mit einem Anfangszählerstand initialisiert. Jede Aufgabe, welche die `await()`-Methode dieses Objektes aufruft, wird solange blockiert, bis der Zähler Null erreicht. Andere Aufgaben können über einen Aufruf der `countDown()`-Methode den Zählerstand dekrementieren (in der Regel nachdem ihre Verarbeitung beendet ist). Ein `CountDownLatch`-Objekt kann entwurfsbedingt nur einmal verwendet werden, das heißt der Zähler kann nicht zurückgesetzt werden. Wenn Sie eine Version brauchen, bei der Sie den Zähler zurücksetzen können, wählen Sie ein `CyclicBarrier`-Objekt (siehe Unterabschnitt 22.7.2).

[254] Eine Aufgabe, welche die `countDown()`-Methode aufruft, wird nicht blockiert. Die Blockierung betrifft diejenigen Aufgaben, welche die `await()`-Methode aufrufen und gilt bis der Zähler den Wert Null erreicht hat.

[255] Eine typische Anwendungssituation besteht darin, ein Problem in n voneinander unabhängig lösbare Teilprobleme aufzuteilen und ein `CountDownLatch`-Objekt mit dem Startwert n zu erzeugen. Ist die Verarbeitung einer Teilaufgabe beendet, so ruft diese Teilaufgabe die `countDown()`-Methode des `CountDownLatch`-Objektes auf. Aufgaben, die auf die Lösung des Gesamtproblems warten, „registrieren“ sich per `await()` beim `CountDownLatch`-Objekt und warten auf das Ende der Verarbeitung. Das folgende Beispiel ist ein Gerüst für diese Vorgehensweise:

```
//: concurrency/CountDownLatchDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// Performs some portion of a task:
class TaskPortion implements Runnable {
```

```
    private static int counter = 0;
    private final int id = counter++;
    private static Random rand = new Random(47);
    private final CountDownLatch latch;
    TaskPortion(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            doWork();
            latch.countDown();
        } catch(InterruptedException ex) {
            // Acceptable way to exit
        }
    }
    public void doWork() throws InterruptedException {
        TimeUnit.MILLISECONDS.sleep(rand.nextInt(2000));
        print(this + ''completed'');
    }
    public String toString() {
        return String.format(''%1$-3d '', id);
    }
}

// Waits on the CountDownLatch:
class WaitingTask implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final CountDownLatch latch;
    WaitingTask(CountDownLatch latch) {
        this.latch = latch;
    }
    public void run() {
        try {
            latch.await();
            print(''Latch barrier passed for '' + this);
        } catch(InterruptedException ex) {
            print(this + '' interrupted'');
        }
    }
    public String toString() {
        return String.format(''WaitingTask %1$-3d '', id);
    }
}

public class CountDownLatchDemo {
    static final int SIZE = 100;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // All must share a single CountDownLatch object:
        CountDownLatch latch = new CountDownLatch(SIZE);
        for(int i = 0; i < 10; i++)
            exec.execute(new WaitingTask(latch));
        for(int i = 0; i < SIZE; i++)
            exec.execute(new TaskPortion(latch));
        print(''Launched all tasks'');
        exec.shutdown(); // Quit when all tasks complete
    }
```

```
} /* (Execute to see output) *////:~
```

Jede Aufgabe vom Typ `TaskPortion` „schläft“ für eine per Zufall bestimmte Zeitspanne, um die Verarbeitung des Teilproblems zu simulieren. Die Klasse `WaitingTask` stellt eine Aufgabe dar, die warten muß, bis der erste Teil des Problems gelöst ist. Alle Aufgaben beziehen sich auf dasselbe in der `main()`-Methode erzeugte `CountDownLatch`-Objekt.

Übungsaufgabe 32: (7) Lösen Sie das Problem ~~of correlating the results~~ der Eingänge aus dem Beispiel *OrnamentalGarden.java* mit Hilfe eines `CountDownLatch`-Objektes. Entfernen Sie die nicht benötigten Anweisungen aus der neuen Version des Beispiels. ■

22.7.1.1 Threadsicherheit und die Standardbibliothek von Java

[256] Beachten Sie, daß die Klasse `TaskPortion` ein statisches `Random`-Objekt referenziert, also viele Aufgaben die `nextInt()`-Methode dieses Objektes aufrufen. Ist das Aufrufen dieser Methode threadsicher?

[257] Falls die Klasse `Random` nicht threadsicher ist, läßt sich das Problem dadurch lösen, daß jedes `TaskPortion`-Objekt ein eigenes `Random`-Objekt erhält, indem Sie das Schlüsselwort `static` entfernen. Die Frage bleibt dennoch bestehen und gilt für die Methoden der Standardbibliothek von Java im allgemeinen: Welche Methoden sind threadsicher und welche nicht?

[258] Bedauerlicherweise ist die JDK-Dokumentation in diesem Punkt nicht gerade entgegenkommend. Die `Random`-Methode `nextInt()` *ist* zwar threadsicher, aber Sie müssen sich diese Informationen für jede Methode einzeln beschaffen, indem Sie entweder im Internet suchen oder im Quelltext der entsprechenden Klasse nachlesen. Keine besonders gute Situation für eine Programmiersprache, die (zumindest theoretisch) vom Entwurf her Threadprogrammierung unterstützt.

22.7.2 CyclicBarrier

[259] Ein `CyclicBarrier`-Objekt ist in Situationen nützlich, in denen Sie eine Anzahl von Aufgaben bis zu einem bestimmten Punkt parallel verarbeiten können und warten müssen, bis jede Aufgabe diesen Punkt erreicht hat, bevor der nächste Arbeitsschritt folgen kann (ähnlich wie `join()`). Es richtet die parallelen Aufgaben an einer Linie aus, von wo aus das Programm anschließend weiterverarbeitet werden. Die Funktionsweise ähnelt `CountDownLatch`, mit dem Unterschied, daß ein `CountDownLatch`-Objekt entwurfsbedingt nur einmal verwendet werden kann, während ein `CyclicBarrier`-Objekt wiederverwendbar ist.

[260] Simulationen haben mich seit meinen ersten Erfahrungen mit Computern fasziniert und die Threadprogrammierung ist der Schlüssel, durch den Simulationen möglich werden. Das erste selbstgeschriebene Programm, an das ich mich erinnere[24], simulierte ein Pferderennen, war in Basic geschrieben und hieß (aufgrund der Einschränkungen bei Dateinamen) *HOSRAC.BAS*. Die folgende objektorierentierte und threadbasierte Version verwendet ein `CyclicBarrier`-Objekt:

```
//: concurrency/HorseRace.java
// Using CyclicBarriers.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Horse implements Runnable {
```

[24] Als Anfänger in der High-Scholl. Im Klassenzimmer gab es einen ASR-33 Fernschreiber mit Zugriff auf eine HP-1000 über ein 110 Baud Akustikkopplermodem.

```
    private static int counter = 0;
    private final int id = counter++;
    private int strides = 0;
    private static Random rand = new Random(47);
    private static CyclicBarrier barrier;
    public Horse(CyclicBarrier b) { barrier = b; }
    public synchronized int getStrides() { return strides; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                synchronized(this) {
                    strides += rand.nextInt(3); // Produces 0, 1 or 2
                }
                barrier.await();
            }
        } catch(InterruptedException e) {
            // A legitimate way to exit
        } catch(BrokenBarrierException e) {
            // This one we want to know about
            throw new RuntimeException(e);
        }
    }
    public String toString() { return ''Horse '' + id + '' ''; }
    public String tracks() {
        StringBuilder s = new StringBuilder();
        for(int i = 0; i < getStrides(); i++)
            s.append(''*'');
        s.append(id);
        return s.toString();
    }
}

public class HorseRace {
    static final int FINISH__LINE = 75;
    private List<Horse> horses = new ArrayList<Horse>();
    private ExecutorService exec = Executors.newCachedThreadPool();
    private CyclicBarrier barrier;
    public HorseRace(int nHorses, final int pause) {
        barrier = new CyclicBarrier(nHorses, new Runnable() {
                public void run() {
                    StringBuilder s = new StringBuilder();
                    for(int i = 0; i < FINISH__LINE; i++)
                        s.append(''='');// The fence on the racetrack
                    print(s);
                    for(Horse horse : horses)
                        print(horse.tracks());
                    for(Horse horse : horses)
                        if(horse.getStrides() >= FINISH__LINE) {
                            print(horse + ''won!'');
                            exec.shutdownNow();
                            return;
                        }
                    try {
                        TimeUnit.MILLISECONDS.sleep(pause);
                    } catch(InterruptedException e) {
                        print(''barrier-action sleep interrupted'');
                    }
                }
```

```
                });
            for(int i = 0; i < nHorses; i++) {
                Horse horse = new Horse(barrier);
                horses.add(horse);
                exec.execute(horse);
            }
        }
        public static void main(String[] args) {
            int nHorses = 7;
            int pause = 200;
            if(args.length > 0) { // Optional argument
                int n = new Integer(args[0]);
                nHorses = n > 0 ? n : nHorses;
            }
            if(args.length > 1) { // Optional argument
                int p = new Integer(args[1]);
                pause = p > -1 ? p : pause;
            }
            new HorseRace(nHorses, pause);
        }
    } /* (Execute to see output) *////:~
```

[261] Ein `CyclicBarrier`-Objekt kann mit einer Aktion (*barrier action*) verknüpft werden (einem *`Runnable`*-Objekt) welche automatisch aufgerufen wird, wenn der Zähler den Wert Null erreicht (ein weiterer Unterschied zwischen den Klassen `CyclicBarrier` und `CountDownLatch`). Die Aktion ist im obigen Beispiel eine anonyme Klasse und wird dem Konstruktor der Klasse `CyclicBarrier` übergeben.

[262] Ich habe versucht, jedes Pferd seine Position selbst drucken zu lassen, wobei Reihenfolge der Pferde aber vom Threadscheduler abhing. Das `CyclicBarrier`-Objekt erlaubt dagegen jedem Pferd seine individuelle Anzahl von Schritten zurückzulegen und anschließend an der Barriere zu warten, bis auch die übrigen Pferde ihre Schritte absolviert haben. Wenn alle Pferde die Barriere erreicht haben, ruft das `CyclicBarrier`-Objekt automatisch sein *`Runnable`*-Objekt auf, welches den Zaun neben der Rennbahn und die Pferde in der richtigen Reihenfolge druckt.

[263] Nachdem alle Aufgaben (Pferde) die Barriere passiert haben, ist das Programm bereit für die nächste Runde.

[264] Wenn Sie das Konsolenfenster so klein machen, daß es nur die sieben Pferde anzeigt, entsteht der Eindruck einer einfachen Simulation.

22.7.3 DelayQueue

[265] Die Klasse `DelayQueue` repräsentiert eine blockierende Warteschlange (*`BlockingQueue`*) mit unbegrenzter Kapazität, deren Elemente das Interface *`java.util.concurrent.Delayed`* implementieren. Ein Element kann nur dann aus der Warteschlange entnommen werden, wenn seine Wartezeit abgelaufen ist. Die Warteschlange ist derart sortiert, daß das Element, welches nach dem Ablaufen seiner Wartezeit am längsten gewartet hat, am Kopfende angeordnet ist (als nächstes entnommen wird). Enthält die Warteschlange nur Elemente, deren Wartezeit noch nicht abgelaufen ist, so existiert kein Kopfelement und die `poll()`-Methode gibt `null` zurück (aus diesem Grund können keine `null`-Referenzen gespeichert werden).

[266] Im folgenden Beispiel sind die *`Delayed`*-Objekte Aufgaben. Ein Objekt der Klasse `DelayedTaskConsumer` entnimmt die dringlichste Aufgabe (diejenige deren Wartezeit am längsten abgelaufen

ist) aus der Warteschlange und verarbeitet sie. Die Klasse `DelayQueue` ist demnach eine Prioritätswarteschlange.

```
//: concurrency/DelayQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static java.util.concurrent.TimeUnit.*;
import static net.mindview.util.Print.*;

class DelayedTask implements Runnable, Delayed {
    private static int counter = 0;
    private final int id = counter++;
    private final int delta;
    private final long trigger;
    protected static List<DelayedTask> sequence = new ArrayList<DelayedTask>();
    public DelayedTask(int delayInMilliseconds) {
        delta = delayInMilliseconds;
        trigger = System.nanoTime() + NANOSECONDS.convert(delta, MILLISECONDS);
        sequence.add(this);
    }
    public long getDelay(TimeUnit unit) {
        return unit.convert(trigger - System.nanoTime(), NANOSECONDS);
    }
    public int compareTo(Delayed arg) {
        DelayedTask that = (DelayedTask)arg;
        if(trigger < that.trigger) return -1;
        if(trigger > that.trigger) return 1;
        return 0;
    }
    public void run() { printnb(this + '' ''); }
    public String toString() {
        return String.format(''[%1$-4d]'', delta) + '' Task '' + id;
    }
    public String summary() {
        return ''('' + id + ''::'' + delta + ''))'';
    }
    public static class EndSentinel extends DelayedTask {
        private ExecutorService exec;
        public EndSentinel(int delay, ExecutorService e) {
            super(delay);
            exec = e;
        }
        public void run() {
            for(DelayedTask pt : sequence) {
                printnb(pt.summary() + '' '');
            }
            print();
            print(this + '' Calling shutdownNow()'');
            exec.shutdownNow();
        }
    }
}

class DelayedTaskConsumer implements Runnable {
    private DelayQueue<DelayedTask> q;
    public DelayedTaskConsumer(DelayQueue<DelayedTask> q) {
        this.q = q;
    }
```

```
    public void run() {
        try {
            while(!Thread.interrupted())
                q.take().run(); // Run task with the current thread
        } catch(InterruptedException e) {
            // Acceptable way to exit
        }
        print("Finished DelayedTaskConsumer");
    }
}

public class DelayQueueDemo {
    public static void main(String[] args) {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        DelayQueue<DelayedTask> queue = new DelayQueue<DelayedTask>();
        // Fill with tasks that have random delays:
        for(int i = 0; i < 20; i++)
            queue.put(new DelayedTask(rand.nextInt(5000)));
        // Set the stopping point
        queue.add(new DelayedTask.EndSentinel(5000, exec));
        exec.execute(new DelayedTaskConsumer(queue));
    }
} /* Output:
      [128 ] Task 11 [200 ] Task 7 [429 ] Task 5 [520 ] Task 18 [555 ] Task 1
      [961 ] Task 4 [998 ] Task 16 [1207] Task 9 [1693] Task 2 [1809] Task 14
      [1861] Task 3 [2278] Task 15 [3288] Task 10 [3551] Task 12 [4258] Task 0
      [4258] Task 19 [4522] Task 8 [4589] Task 13 [4861] Task 17 [4868] Task 6
      (0:4258) (1:555) (2:1693) (3:1861) (4:961) (5:429) (6:4868) (7:200)
      (8:4522) (9:1207) (10:3288) (11:128) (12:3551) (13:4589) (14:1809)
      (15:2278) (16:998) (17:4861) (18:520) (19:4258) (20:5000)
      [5000] Task 20 Calling shutdownNow()
      Finished DelayedTaskConsumer
   *///:~
```

Das `sequence`-Feld der Klasse `DelayedTask` referenziert ein *`List<DelayedTask>`*-Objekt, welches die erzeugten `DelayedTask`-Objekte in ihrer ursprünglichen Reihenfolge speichert, so daß wir die Sortierung der Aufgaben in der Warteschlange bestätigen können.

[267] Das Interface *`Delayed`* deklariert nur eine einzige Methode: `getDelay()`. Die `getDelay()`-Methode gibt an, wie lange es noch dauert, bis die Wartezeit abläuft beziehungsweise wieviel Zeit seit dem Ende der Wartezeit verstrichen ist. Die `getDelay()`-Methode, genauer der Typ ihres Argumentes, erzwingen die Verwendung des Aufzählungstyps `java.util.concurrent.TimeUnit`. Der Aufzählungstyp `TimeUnit` erweist sich als eine sehr nützliche Klasse mit einer einfachen Konvertierungsmethode, um Zeiträume von einer Zeiteinheit in eine andere umwandeln zu können, ohne selbst rechnen zu müssen. Beispielsweise ist die Länge der Wartezeit (`delta`) in Millisekunden gegeben, während die `System`-Methode `nanoTime()` eine Zeitangabe in Nanosekunden zurückgibt. Sie können den Inhalt von `delta` umwandeln, indem Sie sie die Ausgangs- und die Zieleinheit angeben:

```
NANOSECONDS.convert(delta, MILLISECONDS);
```

Die `getDelay()`-Methode erwartet die gewünschte Zieleinheit als Argument (`unit`) und wird im obigen Beispiel verwendet, um den zeitlichen Abstand zum Ende der Wartezeit (das heißt die verbleibende Wartezeit) in die vom Aufrufer angeforderte Zeiteinheit zu konvertieren, ohne zu wissen, welche Zeiteinheit übergeben wird. (Dies ist ein einfaches Beispiel für das Entwurfsmuster *Strategy*, bei dem ein Teil des Algorithmus als Argument übergeben wird.)

[268] Um die Sortierbarkeit der Elemente in der Warteschlange zu gewährleisten, erweitert *Delayed* das Interface *Comparable*, das heißt die `compareTo()`-Methode muß implementiert werden. Die Methoden `toString()` und `summary()` dienen der formatierten Ausgabe und die geschachtelte statische Klasse `EndSentinel` sorgt als letztes Element in der Warteschlange dafür, daß die Anwendung in geordneter Weise beendet wird.

[269] Da die Klasse `DelayedTaskConsumer` selbst eine Aufgabe ist, wird sie mit Hilfe eines eigenen Threads verarbeitet, der verwendet werden kann, um die einzeln aus der Warteschlange entnommenen Aufgaben zu verarbeiten. Da die Aufgaben in der durch die Warteschlange aufgeprägten Reihenfolge verarbeitet werden, besteht in diesem Beispiel kein Bedarf, zu diesem Zweck eigenständige Threads zu erzeugen.

[270] Sie können an der Ausgabe erkennen, daß die Reihenfolge in der die Aufgaben erzeugt werden, keine Auswirkung auf die Verarbeitungsreihenfolge hat. Die Aufgaben werden erwartungsgemäß entsprechend ihrer Wartezeit verarbeitet.

22.7.4 PriorityBlockingQueue

[271] Die Klasse `PriorityBlockingQueue` repräsentiert im Grunde genommen eine Prioritätswarteschlange mit blockierenden Entnahmemethoden. Die Elemente der Prioritätswarteschlange im folgenden Beispiel können in der durch ihre Priorität induzierten Reihenfolge entnommen werden. Die Klasse `PrioritizedTask` stellt eine Aufgabe mit einer Prioritätszahl dar, nach der diese Aufgaben sortiert werden:

```
//: concurrency/PriorityBlockingQueueDemo.java
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class PrioritizedTask implements Runnable, Comparable<PrioritizedTask> {
    private Random rand = new Random(47);
    private static int counter = 0;
    private final int id = counter++;
    private final int priority;
    protected static List<PrioritizedTask> sequence =
        new ArrayList<PrioritizedTask>();
    public PrioritizedTask(int priority) {
        this.priority = priority;
        sequence.add(this);
    }
    public int compareTo(PrioritizedTask arg) {
        return priority < arg.priority ? 1 : (priority > arg.priority ? -1 : 0);
    }
    public void run() {
        try {
            TimeUnit.MILLISECONDS.sleep(rand.nextInt(250));
        } catch(InterruptedException e) {
            // Acceptable way to exit
        }
        print(this);
    }
    public String toString() {
        return String.format(''[%1$-3d]'', priority) + '' Task '' + id;
    }
    public String summary() {
        return ''('' + id + '':'' + priority + '')'';
```

```
    }
    public static class EndSentinel extends PrioritizedTask {
        private ExecutorService exec;
        public EndSentinel(ExecutorService e) {
            super(-1); // Lowest priority in this program
            exec = e;
        }
        public void run() {
            int count = 0;
            for(PrioritizedTask pt : sequence) {
                printnb(pt.summary());
                if(++count % 5 == 0)
                    print();
            }
            print();
            print(this + '' Calling shutdownNow()'');
            exec.shutdownNow();
        }
    }
}

class PrioritizedTaskProducer implements Runnable {
    private Random rand = new Random(47);
    private Queue<Runnable> queue;
    private ExecutorService exec;
    public PrioritizedTaskProducer(Queue<Runnable> q, ExecutorService e) {
        queue = q;
        exec = e; // Used for EndSentinel
    }
    public void run() {
        // Unbounded queue; never blocks.
        // Fill it up fast with random priorities:
        for(int i = 0; i < 20; i++) {
            queue.add(new PrioritizedTask(rand.nextInt(10)));
            Thread.yield();
        }
        // Trickle in highest-priority jobs:
        try {
            for(int i = 0; i < 10; i++) {
                TimeUnit.MILLISECONDS.sleep(250);
                queue.add(new PrioritizedTask(10));
            }
            // Add jobs, lowest priority first:
            for(int i = 0; i < 10; i++)
                queue.add(new PrioritizedTask(i));
            // A sentinel to stop all the tasks:
            queue.add(new PrioritizedTask.EndSentinel(exec));
        } catch(InterruptedException e) {
            // Acceptable way to exit
        }
        print(''Finished PrioritizedTaskProducer'');
    }
}

class PrioritizedTaskConsumer implements Runnable {
    private PriorityBlockingQueue<Runnable> q;
    public PrioritizedTaskConsumer(PriorityBlockingQueue<Runnable> q) {
        this.q = q;
```

```
    }
    public void run() {
        try {
            while(!Thread.interrupted())
                // Use current thread to run the task:
                q.take().run();
        } catch(InterruptedException e) {
            // Acceptable way to exit
        }
        print("Finished PrioritizedTaskConsumer");
    }
}

public class PriorityBlockingQueueDemo {
    public static void main(String[] args) throws Exception {
        Random rand = new Random(47);
        ExecutorService exec = Executors.newCachedThreadPool();
        PriorityBlockingQueue<Runnable> queue =
            new PriorityBlockingQueue<Runnable>();
        exec.execute(new PrioritizedTaskProducer(queue, exec));
        exec.execute(new PrioritizedTaskConsumer(queue));
    }
} /* (Execute to see output) *////:~
```

[272] Wie beim vorigen Beispiel wird die Reihenfolge, in der die `PrioritizedTask`-Objekt erzeugt werden, in einem *`List`*-Objekt gespeichert, um sie mit der Reihenfolge der tatsächlichen Verarbeitung vergleichen zu können. Die `run()`-Methode der Klasse `PrioritizedTask` „schläft“ für eine kurze zufällig gewählte Zeitspanne und gibt anschließend die Identifikation des Objektes zurück. Die geschachtelte statische Klasse `EndSentinel` hat wie dieselbe Aufgabe wie beim vorigen Beispiel und repräsentiert wieder das letzte Element in der Warteschlange.

[273] Die Klassen `PrioritizedTaskProducer` und `PrioritizedTaskConsumer` sind über ein Objekt er `PriorityBlockingQueue` miteinander verbunden. Da die Warteschlange durch ihr blockierendes Verhalten die erforderliche Synchronisierung liefert, ist keine explizite Synchronisierung erforderlich. Sie brauchen sich keine Gedanken darüber zu machen, ob die Warteschlange Elemente enthält, wenn Sie Elemente entnehmen wollen, da die Warteschlange die Entnahmemethode einfach blockiert, wenn keine Elemente vorhanden sind.

22.7.5 ScheduledThreadPoolExecutor (GreenhouseController-Beispiel)

[274] In Kapitel 11 wurde das Gewächshausbeispiel eingeführt, welches die Steuerung eines Gewächshauses simuliert, in dem verschiedene Geräte ein- und ausgeschaltet oder justiert werden. Dieses Beispiel kann auch aus der Perpektive der Threadprogrammierung betrachtet werden, wobei jedes Ereignis auf eine Aufgabe abgebildet wird, die zu einem vordefinierten Zeitpunkt verarbeitet wird. Die Klasse `ScheduledThreadPoolExecutor` liefert die zur Lösung dieses Problems benötigte Funktionalität. Mit Hilfe der Methoden `schedule()` (für einmalige Aufgaben) beziehungsweise `scheduleAtFixedRate()` (für periodisch zu wiederholende Aufgaben) verfügen Sie, daß ein *`Runnable`*-Objekt an einem festgesetzten zukünftigen Zeitpunkt verarbeitet wird. Vergleichen Sie die folgende Lösung mit dem Ansatz aus Kapitel 11, um zu sehen, wieviel einfacher die Lösung mit einer vordefinierten Klasse wie `ScheduledThreadPoolExecutor` fällt:

```
//: concurrency/GreenhouseScheduler.java
// Rewriting innerclasses/GreenhouseController.java
// to use a ScheduledThreadPoolExecutor.
// {Args: 5000}
```

```
import java.util.concurrent.*;
import java.util.*;

public class GreenhouseScheduler {
    private volatile boolean light = false;
    private volatile boolean water = false;
    private String thermostat = ''Day'';
    public synchronized String getThermostat() {
        return thermostat;
    }
    public synchronized void setThermostat(String value) {
        thermostat = value;
    }
    ScheduledThreadPoolExecutor scheduler = new ScheduledThreadPoolExecutor(10);
    public void schedule(Runnable event, long delay) {
        scheduler.schedule(event,delay,TimeUnit.MILLISECONDS);
    }
    public void repeat(Runnable event, long initialDelay, long period) {
        scheduler.scheduleAtFixedRate(
            event, initialDelay, period, TimeUnit.MILLISECONDS);
    }
    class LightOn implements Runnable {
        public void run() {
            // Put hardware control code here to
            // physically turn on the light.
            System.out.println(''Turning on lights'');
            light = true;
        }
    }
    class LightOff implements Runnable {
        public void run() {
            // Put hardware control code here to
            // physically turn off the light.
            System.out.println(''Turning off lights'');
            light = false;
        }
    }
    class WaterOn implements Runnable {
        public void run() {
            // Put hardware control code here.
            System.out.println(''Turning greenhouse water on'');
            water = true;
        }
    }
    class WaterOff implements Runnable {
        public void run() {
            // Put hardware control code here.
            System.out.println(''Turning greenhouse water off'');
            water = false;
        }
    }
    class ThermostatNight implements Runnable {
        public void run() {
            // Put hardware control code here.
            System.out.println(''Thermostat to night setting'');
            setThermostat(''Night'');
        }
    }
```

```
class ThermostatDay implements Runnable {
    public void run() {
        // Put hardware control code here.
        System.out.println("Thermostat to day setting");
        setThermostat("Day");
    }
}
class Bell implements Runnable {
    public void run() { System.out.println("Bing!"); }
}
class Terminate implements Runnable {
    public void run() {
        System.out.println("Terminating");
        scheduler.shutdownNow();
        // Must start a separate task to do this job,
        // since the scheduler has been shut down:
        new Thread() {
            public void run() {
                for(DataPoint d : data)
                    System.out.println(d);
            }
        }.start();
    }
}
// New feature: data collection
static class DataPoint {
    final Calendar time;
    final float temperature;
    final float humidity;
    public DataPoint(Calendar d, float temp, float hum) {
        time = d;
        temperature = temp;
        humidity = hum;
    }
    public String toString() {
        return time.getTime() +
            String.format(" temperature: %1$.1f humidity: %2$.2f",
                temperature, humidity);
    }
}
private Calendar lastTime = Calendar.getInstance();
{ // Adjust date to the half hour
    lastTime.set(Calendar.MINUTE, 30);
    lastTime.set(Calendar.SECOND, 00);
}
private float lastTemp = 65.0f;
private int tempDirection = +1;
private float lastHumidity = 50.0f;
private int humidityDirection = +1;
private Random rand = new Random(47);
List<DataPoint> data = Collections.synchronizedList(new ArrayList<DataPoint>());
class CollectData implements Runnable {
    public void run() {
        System.out.println("Collecting data");
        synchronized(GreenhouseScheduler.this) {
            // Pretend the interval is longer than it is:
            lastTime.set(Calendar.MINUTE, lastTime.get(Calendar.MINUTE) + 30);
```

```
                // One in 5 chances of reversing the direction:
                if(rand.nextInt(5) == 4)
                    tempDirection = -tempDirection;
                // Store previous value:
                lastTemp = lastTemp + tempDirection * (1.0f + rand.nextFloat());
                if(rand.nextInt(5) == 4)
                    humidityDirection = -humidityDirection;
                lastHumidity = lastHumidity + humidityDirection * rand.nextFloat();
                // Calendar must be cloned, otherwise all
                // DataPoints hold references to the same lastTime.
                // For a basic object like Calendar, clone() is OK.
                data.add(new DataPoint((Calendar) lastTime.clone(),
                    lastTemp, lastHumidity));
            }
        }
    }
    public static void main(String[] args) {
        GreenhouseScheduler gh = new GreenhouseScheduler();
        gh.schedule(gh.new Terminate(), 5000);
        // Former ''Restart'' class not necessary:
        gh.repeat(gh.new Bell(), 0, 1000);
        gh.repeat(gh.new ThermostatNight(), 0, 2000);
        gh.repeat(gh.new LightOn(), 0, 200);
        gh.repeat(gh.new LightOff(), 0, 400);
        gh.repeat(gh.new WaterOn(), 0, 600);
        gh.repeat(gh.new WaterOff(), 0, 800);
        gh.repeat(gh.new ThermostatDay(), 0, 1400);
        gh.repeat(gh.new CollectData(), 500, 500);
    }
} /* (Execute to see output) *////:~
```

[275] Neben der Neuorganisation des Quelltextes bietet diese Version eine zusätzliche Funktionalität, nämlich die Aufzeichnung von Temperatur- und Luftfeuchtigkeitswerten im Gewächshaus. Die Klasse `DataPoint` repräsentiert einen einzelnen Datenpunkt. Die Aufgabenklasse `CollectData` erzeugt simulierte Meßwerte und speichert sie in einem *`List<DataPoint>`*-Objekt.

[276] Beachten Sie die Verwendung von `volatile` und `synchronized` an den entsprechenden Stellen, um zu verhindern, daß sich die Aufgaben störend beeinflussen. Alle Methoden des *`List<DataPoint>`*-Objektes sind aufgrund der `Collections`-Hilfsmethode `synchronizedList()` beim Erzeugen des Listenobjektes synchronisiert.

Übungsaufgabe 33: (7) Ändern Sie das Beispiel *GreenhouseScheduler.java*, so daß statt eines `ScheduledThreadPoolExecutor`-Objektes ein `DelayQueue`-Objekt verwendet wird. ■

22.7.6 Semaphore

[277] Eine gewöhnliche Sperre (Sperrobjekt oder traditionelle Synchronisierung) gestattet stets nur einer Aufgabe den Zugriff auf eine Resource. Ein *zählendes Semaphor* gestattet dagegen n Aufgaben gleichzeitig auf eine Resource zuzugreifen. Sie können sich vorstellen, daß ein Semaphor „Berechtigungen“ (*permits*) für den Zugriff auf eine Resource vergibt, wobei aber tatsächlich keine Berechtigungsobjekte verwendet werden.

[278] Wir betrachten einen Vorrat von Objekten (*object pool*) als Beispiel, der eine begrenzte Anzahl von Objekten verwaltet, die zur Benutzung entnommen und anschließend wieder zurückgegeben werden können. Die Klasse `Pool` kapselt diese Funktionalität:

```
//: concurrency/Pool.java
// Using a Semaphore inside a Pool, to restrict
// the number of tasks that can use a resource.
import java.util.concurrent.*;
import java.util.*;

public class Pool<T> {
    private int size;
    private List<T> items = new ArrayList<T>();
    private volatile boolean[] checkedOut;
    private Semaphore available;
    public Pool(Class<T> classObject, int size) {
        this.size = size;
        checkedOut = new boolean[size];
        available = new Semaphore(size, true);
        // Load pool with objects that can be checked out:
        for(int i = 0; i < size; ++i)
            try {
                // Assumes a default constructor:
                items.add(classObject.newInstance());
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
    }
    public T checkOut() throws InterruptedException {
        available.acquire();
        return getItem();
    }
    public void checkIn(T x) {
        if(releaseItem(x))
            available.release();
    }
    private synchronized T getItem() {
        for(int i = 0; i < size; ++i)
            if(!checkedOut[i]) {
                checkedOut[i] = true;
                return items.get(i);
            }
        return null; // Semaphore prevents reaching here
    }
    private synchronized boolean releaseItem(T item) {
        int index = items.indexOf(item);
        if(index == -1) return false; // Not in the list
        if(checkedOut[index]) {
            checkedOut[index] = false;
            return true;
        }
        return false; // Wasn't checked out
    }
} ///:~
```

In diesem vereinfachten Beispiel verwendet der Konstruktor die Methode `newInstance()` des Klassenobjektes, um den Vorrat mit Objekt aufzufüllen. Wenn Sie ein neues Objekt brauchen, können Sie es per `checkOut()` entnehmen und nach Gebrauch mittels `checkIn()` wieder zurücklegen.

[279] Das `boolean`-Array `checkOut` verwaltet den Vergabestatus der Objekte und wird von den Methoden `getItem()` und `releaseItem()` gepflegt. Die Objekte werden wiederum mit Hilfe des Semaphors `available` überwacht: In der `checkOut()`-Methode verursacht das Semaphor eine Blo-

ckierung des Methodenaufrufs, wenn keine Berechtigungen mehr verfügbar sind (der Vorrat an Objekten also erschöpft ist). In der `checkIn()`-Methode wird eine Berechtigung an das Semaphor zurückgegeben, wenn das zurückgegebene Objekt gültig ist.

[280] Der Vorrat besteht in diesem Beispiel aus Objekten der folgenden Klasse `Fat`. Das Erzeugen eines `Fat`-Objektes ist teuer, da der Konstruktor eine aufwändige Operation ausführt:

```
//: concurrency/Fat.java
// Objects that are expensive to create.

public class Fat {
    private volatile double d; // Prevent optimization
    private static int counter = 0;
    private final int id = counter++;
    public Fat() {
        // Expensive, interruptible operation:
        for(int i = 1; i < 10000; i++) {
            d += (Math.PI + Math.E) / (double)i;
        }
    }
    public void operation() { System.out.println(this); }
    public String toString() { return "Fat id: " + id; }
} ///:~
```

Wir bevorraten diese Objekte, um die Auswirkung des Konstruktoraufrufs zu limitieren. Wir testen die Klasse `Pool`, indem wir eine Aufgabenklasse `CheckoutTask` anlegen, die `Fat`-Objekte aus dem Vorrat entnimmt, eine Zeitspanne verstreichen läßt und sie anschließend wieder zurückgibt:

```
//: concurrency/SemaphoreDemo.java
// Testing the Pool class
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// A task to check a resource out of a pool:
class CheckoutTask<T> implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private Pool<T> pool;
    public CheckoutTask(Pool<T> pool) {
        this.pool = pool;
    }
    public void run() {
        try {
            T item = pool.checkOut();
            print(this + "checked out " + item);
            TimeUnit.SECONDS.sleep(1);
            print(this +"checking in " + item);
            pool.checkIn(item);
        } catch(InterruptedException e) {
            // Acceptable way to terminate
        }
    }
    public String toString() {
        return "CheckoutTask " + id + " ";
    }
}

public class SemaphoreDemo {
```

```
    final static int SIZE = 25;
    public static void main(String[] args) throws Exception {
        final Pool<Fat> pool = new Pool<Fat>(Fat.class, SIZE);
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < SIZE; i++)
            exec.execute(new CheckoutTask<Fat>(pool));
        print("All CheckoutTasks created");
        List<Fat> list = new ArrayList<Fat>();
        for(int i = 0; i < SIZE; i++) {
            Fat f = pool.checkOut();
            printnb(i + ": main() thread checked out ");
            f.operation();
            list.add(f);
        }
        Future<?> blocked = exec.submit(new Runnable() {
                public void run() {
                    try {
                        // Semaphore prevents additional checkout,
                        // so call is blocked:
                        pool.checkOut();
                    } catch(InterruptedException e) {
                        print("checkOut() Interrupted");
                    }
                }
            });
        TimeUnit.SECONDS.sleep(2);
        blocked.cancel(true); // Break out of blocked call
        print("Checking in objects in " + list);
        for(Fat f : list)
            pool.checkIn(f);
        for(Fat f : list)
            pool.checkIn(f); // Second checkIn ignored
        exec.shutdown();
    }
} /* (Execute to see output) *////:~
```

In der `main()`-Methode wird ein `Pool<Fat>`-Objekt erzeugt und eine Anzahl von `CheckoutTask`-Aufgaben beginnt, den Vorrat zur verarbeiten. Der `main`-Thread entnimmt die bevorrateten `Fat`-Objekte, *ohne sie anschließend zurückzugeben*. Nachdem alle Objekte aus dem Vorrat entnommen wurden, gestattet das Semaphor keine weitere Entnahme mehr. Aus diesem Grund wird die `run()`-Methode des von `blocked` referenzierten *`Future`*-Objektes blockiert und nach zwei Sekunden per `cancel()` abgebrochen. Beachten Sie, daß das `Pool`-Objekt redundantes Zurückgeben ignoriert.

[281] Dieses Beispiel setzt gewissenhaften Umgang mit den Elementen des `Pool`-Objektes voraus, das heißt deren Rückgabe. Falls Sie sich nicht darauf verlassen können, finden Sie in *Thinking in Patterns* (*http://www.mindview.net*) weiterführende Möglichkeiten zur Verwaltung von aus einem Vorrat entnommenen Objekten.

22.7.7 Exchanger

[282] Die Klasse `Exchanger` stellt eine Barriere dar, an der Objekte zwischen Aufgaben ausgetauscht werden können. Beim Erreichen der Sperre hat jede Aufgabe ihr Objekt. Beim Verlassen der Sperre hat jede Aufgabe das Objekt, welches zuvor zu der anderen Aufgabe gehörte. `Exchanger`-Objekte werden typischerweise verwendet, wenn eine Aufgabe Objekte erzeugt, deren Erzeugung teuer ist, und eine andere Aufgabe diese Objekte verbraucht. Auf diese Weise können Objekte parallel zu

ihrer Verwendung erzeugt werden.

[283] Zur Vorführung der Klasse `Exchanger` schreiben wir je eine Erzeuger- und eine Verbraucheraufgabe, welche durch generische Typen und Generatoren (Interface *`Generator`*) auf jeden beliebigen Objekttyp angewendet werden können und wählen Objekte der Klasse `Fat` aus dem vorigen Unterabschnitt als Elemente. Die Klassen `ExchangerProducer` und `ExchangerConsumer` tauschen ein *`List<T>`*-Objekt aus. Beide Klassen verfügen zu diesem Zweck über ein `Exchanger`-Objekt. Die `Exchanger`-Methode `exchange()` wird solange blockiert, bis der Partner ebenfalls seine `exchange()`-Methode aufruft. Nach der Verarbeitung beider `exchange()`-Methoden sind die *`List<T>`*-Objekte vertauscht:

```
//: concurrency/ExchangerDemo.java
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class ExchangerProducer<T> implements Runnable {
    private Generator<T> generator;
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    ExchangerProducer(Exchanger<List<T>> exchg,
        Generator<T> gen, List<T> holder) {
        exchanger = exchg;
        generator = gen;
        this.holder = holder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                for(int i = 0; i < ExchangerDemo.size; i++)
                    holder.add(generator.next());
                // Exchange full for empty:
                holder = exchanger.exchange(holder);
            }
        } catch(InterruptedException e) {
            // OK to terminate this way.
        }
    }
}

class ExchangerConsumer<T> implements Runnable {
    private Exchanger<List<T>> exchanger;
    private List<T> holder;
    private volatile T value;
    ExchangerConsumer(Exchanger<List<T>> ex, List<T> holder) {
        exchanger = ex;
        this.holder = holder;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                holder = exchanger.exchange(holder);
                for(T x : holder) {
                    value = x; // Fetch out value
                    holder.remove(x); // OK for CopyOnWriteArrayList
                }
            }
        } catch(InterruptedException e) {
```

```
            // OK to terminate this way.
        }
        System.out.println("Final value: " + value);
    }
}

public class ExchangerDemo {
    static int size = 10;
    static int delay = 5; // Seconds
    public static void main(String[] args) throws Exception {
        if(args.length > 0)
            size = new Integer(args[0]);
        if(args.length > 1)
            delay = new Integer(args[1]);
        ExecutorService exec = Executors.newCachedThreadPool();
        Exchanger<List<Fat>> xc = new Exchanger<List<Fat>>();
        List<Fat>
            producerList = new CopyOnWriteArrayList<Fat>(),
            consumerList = new CopyOnWriteArrayList<Fat>();
        exec.execute(new ExchangerProducer<Fat>(
            xc, BasicGenerator.create(Fat.class), producerList));
        exec.execute(new ExchangerConsumer<Fat>(xc,consumerList));
        TimeUnit.SECONDS.sleep(delay);
        exec.shutdownNow();
    }
} /* Output: (Sample)
      Final value: Fat id: 29999
   *///:~
```

In der `main()`-Methode werden ein einzelnes `Exchanger`-Objekt für beide Aufgaben (`ExchangerProducer` und `ExchangerConsumer`) sowie zwei `CopyOnWriteArrayList`-Objekte erzeugt, die anschließend ausgetauscht werden sollen. Diese spezielle *`List`*-Variante toleriert das Aufrufen der `remove()`-Methode, während die Liste durchlaufen wird, ohne eine Ausnahme vom Typ `java.util.ConcurrentModificationException` auszuwerfen. Die Klasse `ExchangerProducer` füllt ein *`List`*-Objekt mit Elementen und tauscht anschließend die volle Liste gegen die von der Klasse `ExchangerConsumer` erhaltene leere Liste aus. Die Klasse `Exchanger` sorgt dafür, daß das Erzeugen und Verbrauchen der Liste gleichzeitig geschieht.

Übungsaufgabe 34: (1) Ändern Sie das Beispiel *ExchangerDemo.java*, so daß es einer Ihrer eigenen Klasse anstelle von `Fat` verwendet. ■

22.8 Simulationen

[284] Simulationen sind eines der interessantesten und aufregendsten Anwendungsgebiete der Threadprogrammierung. Im Rahmen der Threadprogrammierung läßt sich jede Komponente einer Simulation auf eine eigene Aufgabe abbilden, wodurch sich die Simulation erheblich leichter entwickeln läßt. Viele Videospiele und CGI Animationen (Computer Generated Imagery) in Filmen sind Simulationen und auch die Beispiele *HorseRace.java* und *GreenhouseScheduler.java* können als Simulationen betrachtet werden.

22.8.1 Bankschalter

[285] Dieses klassische Simulationsbeispiel repräsentiert eine Situation, in der Objekte zufällig auftreten, eine zufällig bestimmte Bearbeitungszeit benötigen und von einer begrenzten Anzahl anderer Objekt bedient werden können. It's possible to build the simulation to determine the ideal number of servers.

[286] In diesem Beispiel benötigt jeder Kunde am Schalter eine bestimmte Bearbeitungszeit, um seine Geschäfte abzuwickeln. Der Angestellte am Schalter muß die erforderliche Anzahl von Zeiteinheiten investieren, um die Wünsche des Kunden zu bearbeiten. Die benötigte Zeit ist von Kunde zu Kunde verschieden und wird mit Hilfe von Zufallszahlen bestimmt. Außerdem ist die Anzahl Kunden, die während eines bestimmten Zeitraumes die Schalterhalle betreten, nicht vorhersehbar und wird ebenfalls mit Hilfe von Zufallszahlen bestimmt.

```
//: concurrency/BankTellerSimulation.java
// Using queues and multithreading.
// {Args: 5}
import java.util.concurrent.*;
import java.util.*;

// Read-only objects don't require synchronization:
class Customer {
    private final int serviceTime;
    public Customer(int tm) { serviceTime = tm; }
    public int getServiceTime() { return serviceTime; }
    public String toString() {
        return "[" + serviceTime + "]";
    }
}

// Teach the customer line to display itself:
class CustomerLine extends ArrayBlockingQueue<Customer> {
    public CustomerLine(int maxLineSize) {
        super(maxLineSize);
    }
    public String toString() {
        if(this.size() == 0)
            return "[Empty]";
        StringBuilder result = new StringBuilder();
        for(Customer customer : this)
            result.append(customer);
        return result.toString();
    }
}

// Randomly add customers to a queue:
class CustomerGenerator implements Runnable {
    private CustomerLine customers;
    private static Random rand = new Random(47);
    public CustomerGenerator(CustomerLine cq) {
        customers = cq;
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(300));
                customers.put(new Customer(rand.nextInt(1000)));
            }
```

```
        } catch(InterruptedException e) {
            System.out.println("CustomerGenerator interrupted");
        }
        System.out.println("CustomerGenerator terminating");
    }
}

class Teller implements Runnable, Comparable<Teller> {
    private static int counter = 0;
    private final int id = counter++;
    // Customers served during this shift:
    private int customersServed = 0;
    private CustomerLine customers;
    private boolean servingCustomerLine = true;
    public Teller(CustomerLine cq) { customers = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                Customer customer = customers.take();
                TimeUnit.MILLISECONDS.sleep(customer.getServiceTime());
                synchronized(this) {
                    customersServed++;
                    while(!servingCustomerLine)
                        wait();
                }
            }
        } catch(InterruptedException e) {
            System.out.println(this + "interrupted");
        }
        System.out.println(this + "terminating");
    }
    public synchronized void doSomethingElse() {
        customersServed = 0;
        servingCustomerLine = false;
    }
    public synchronized void serveCustomerLine() {
        assert !servingCustomerLine:"already serving: " + this;
        servingCustomerLine = true;
        notifyAll();
    }
    public String toString() { return "Teller " + id + " "; }
    public String shortString() { return "T" + id; }
    // Used by priority queue:
    public synchronized int compareTo(Teller other) {
        return customersServed < other.customersServed ? -1 :
            (customersServed == other.customersServed ? 0 : 1);
    }
}

class TellerManager implements Runnable {
    private ExecutorService exec;
    private CustomerLine customers;
    private PriorityQueue<Teller> workingTellers = new PriorityQueue<Teller>();
    private Queue<Teller> tellersDoingOtherThings = new LinkedList<Teller>();
    private int adjustmentPeriod;
    private static Random rand = new Random(47);
    public TellerManager(ExecutorService e,
        CustomerLine customers, int adjustmentPeriod) {
```

```
        exec = e;
        this.customers = customers;
        this.adjustmentPeriod = adjustmentPeriod;
        // Start with a single teller:
        Teller teller = new Teller(customers);
        exec.execute(teller);
        workingTellers.add(teller);
    }
    public void adjustTellerNumber() {
        // This is actually a control system. By adjusting
        // the numbers, you can reveal stability issues in
        // the control mechanism.
        // If line is too long, add another teller:
        if(customers.size() / workingTellers.size() > 2) {
            // If tellers are on break or doing
            // another job, bring one back:
            if(tellersDoingOtherThings.size() > 0) {
                Teller teller = tellersDoingOtherThings.remove();
                teller.serveCustomerLine();
                workingTellers.offer(teller);
                return;
            }
            // Else create (hire) a new teller
            Teller teller = new Teller(customers);
            exec.execute(teller);
            workingTellers.add(teller);
            return;
        }
        // If line is short enough, remove a teller:
        if(workingTellers.size() > 1 &&
            customers.size() / workingTellers.size() < 2)
            reassignOneTeller();
        // If there is no line, we only need one teller:
        if(customers.size() == 0)
            while(workingTellers.size() > 1)
                reassignOneTeller();
    }
    // Give a teller a different job or a break:
    private void reassignOneTeller() {
        Teller teller = workingTellers.poll();
        teller.doSomethingElse();
        tellersDoingOtherThings.offer(teller);
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                TimeUnit.MILLISECONDS.sleep(adjustmentPeriod);
                adjustTellerNumber();
                System.out.print(customers + " { ");
                for(Teller teller : workingTellers)
                    System.out.print(teller.shortString() + " ");
                System.out.println("}");
            }
        } catch(InterruptedException e) {
            System.out.println(this + "interrupted");
        }
        System.out.println(this + "terminating");
```

```
    }
    public String toString() { return ''TellerManager ''; }
}

public class BankTellerSimulation {
    static final int MAX__LINE__SIZE = 50;
    static final int ADJUSTMENT__PERIOD = 1000;
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        // If line is too long, customers will leave:
        CustomerLine customers = new CustomerLine(MAX__LINE__SIZE);
        exec.execute(new CustomerGenerator(customers));
        // Manager will add and remove tellers as necessary:
        exec.execute(new TellerManager(
                                    exec, customers, ADJUSTMENT__PERIOD));
        if(args.length > 0) // Optional argument
            TimeUnit.SECONDS.sleep(new Integer(args[0]));
        else {
            System.out.println(''Press 'Enter' to quit'');
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* Output: (Sample)
      [429][200][207] { T0 T1 }
      [861][258][140][322] { T0 T1 }
      [575][342][804][826][896][984] { T0 T1 T2 }
      [984][810][141][12][689][992][976][368][395][354] { T0 T1 T2 T3 }
      Teller 2 interrupted
      Teller 2 terminating
      Teller 1 interrupted
      Teller 1 terminating
      TellerManager interrupted
      TellerManager terminating
      Teller 3 interrupted
      Teller 3 terminating
      Teller 0 interrupted
      Teller 0 terminating
      CustomerGenerator interrupted
      CustomerGenerator terminating
   *///:~
```

[287] Die Klasse `Customer` ist sehr einfach aufgebaut und enthält lediglich ein finales `int`-Feld (`serviceTime`). Da sich der Zustand eines `Customer`-Objektes nicht ändert, sind es nur-lesbare Objekte und es ist weder Synchronisierung noch die Deklaration von `serviceTime` als volatiles Feld erforderlich. Die Klasse `Teller` entnimmt stets nur ein einziges `Customer`-Objekt aus der Warteschlange und verarbeitet es vollständig, das heißt, daß stets nicht mehr als ein Thread auf ein `Customer`-Objekt zugreift.

[288] Die Klasse `CustomerLine` repräsentiert eine Warteschlange im Schalterraum, in der die Kunden am Schalter stehen. Die Klasse `CustomerLine` ist von der Klasse `ArrayBlockingQueue` abgeleitet und erhält lediglich eine `toString()`-Methode um die Warteschlange in ansprechender Formatierung ausgeben zu können.

[289] Die Klasse `CustomerGenerator` bevölkert ein `CustomerLine`-Objekt mit `Customer`-Elementen.

[290] Die Klasse `Teller` entnimmt stets genau ein `Customer`-Element aus der `CustomerLine`-Warteschlange, verarbeitet es und führt Buch über die Anzahl bedienter Kunden. Ein `Teller`-

Objekt kann per `doSomethingElse()` veranlaßt werden, eine andere Tätigkeit auszuüben, oder per `serveCustomerLine()` einen Kunden aus der Warteschlange zu bedienen, wenn viele Kunden warten. To choose the next teller to put back on the line, the `compareTo()` method looks at the number of customers served so that a `PriorityQueue` can automatically put the least-worked teller at the forefront.

[291] Die Klasse `TellerManager` stellt den Verteilerknoten der Aktivität dar. Sie überwacht die Schalterangestellten und Kunden. Eine interessante Eigenschaft dieser Simulation besteht darin, daß sie versucht, die optimale Anzahl von Schalterangestellten für einen gegebenen Kundenandrang zu ermitteln. Diese Funktionalität ist in der Methode `adjustTellerNumber()` implementiert, einem Kontrollsystem, das Angestellte auf stabile Weise in beziehungsweise außer Dienst stellen kann. Stabilität ist eine wesentliche Eigenschaft von Kontrollsystemen. Reagiert das System zu schnell, so wird das gesteuerte Programm instabil. Reagiert das System zu langsam, so nimmt das gesteuerte Programm extremes Verhalten an.

Übungsaufgabe 35: (8) Ändern Sie das Beispiel *BankTellerSimulation.java* so, daß Webclients Anfragen an eine feste Anzahl von Servern senden. Das Ziel besteht darin, die Auslastung zu bestimmen, die eine Gruppe von Servern bewältigen kann. ■

22.8.2 Restaurant

[292] Diese Simulation kleidet das einfache Beispiel *Restaurant.java* aus Unterabschnitt 22.5.3 mit einigen zusätzlichen Komponenten wie Bestellungen (`Order`-Objekten) und Tellern (`Plate`-Objekten) aus und stützt sich darüber hinaus auf die `enumerated.menu`-Klassen aus Kapitel 20.

[293] Das Beispiel führt außerdem die Klasse `SynchronousQueue` (seit der SE 5) vor, eine blockierende Warteschlange ohne interne Kapazität, so daß jeder `put()`-Aufruf auf einen `take()`-Aufruf warten muß und umgekehrt. Stellen Sie sich vor, daß Sie jemandem ein Buch geben wollen und kein Tisch vorhanden ist, um es darauf zu legen. Die Übergabe funktioniert nur, wenn die Zielperson eine Hand ausstreckt, um das Buch entgegenzunehmen. In diesem Beispiel stellt das `SynchronousQueue`-Objekt die Situation im Restaurant dar, genauer, daß stets nur ein Teller serviert werden kann.

[294] Die übrigen Klassen und die restliche Funktionalität ergeben sich entweder aus der Struktur des Beispiels *Restaurant.java* oder bilden die Vorgänge in einem Restaurant direkt ab:

```
//: concurrency/restaurant2/RestaurantWithQueues.java
// {Args: 5}
package concurrency.restaurant2;
import enumerated.menu.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

// This is given to the waiter, who gives it to the chef:
class Order { // (A data-transfer object)
    private static int counter = 0;
    private final int id = counter++;
    private final Customer customer;
    private final WaitPerson waitPerson;
    private final Food food;
    public Order(Customer cust, WaitPerson wp, Food f) {
        customer = cust;
        waitPerson = wp;
        food = f;
    }
```

```
    public Food item() { return food; }
    public Customer getCustomer() { return customer; }
    public WaitPerson getWaitPerson() { return waitPerson; }
    public String toString() {
        return ''Order: '' + id + '' item: '' + food +
            '' for: '' + customer + '' served by: '' + waitPerson;
    }
}

// This is what comes back from the chef:
class Plate {
    private final Order order;
    private final Food food;
    public Plate(Order ord, Food f) {
        order = ord;
        food = f;
    }
    public Order getOrder() { return order; }
    public Food getFood() { return food; }
    public String toString() { return food.toString(); }
}

class Customer implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final WaitPerson waitPerson;
    // Only one course at a time can be received:
    private SynchronousQueue<Plate> placeSetting = new SynchronousQueue<Plate>();
    public Customer(WaitPerson w) { waitPerson = w; }
    public void deliver(Plate p) throws InterruptedException {
        // Only blocks if customer is still
        // eating the previous course:
        placeSetting.put(p);
    }
    public void run() {
        for(Course course : Course.values()) {
            Food food = course.randomSelection();
            try {
                waitPerson.placeOrder(this, food);
                // Blocks until course has been delivered:
                print(this + ''eating '' + placeSetting.take());
            } catch(InterruptedException e) {
                print(this + ''waiting for '' + course + '' interrupted'');
                break;
            }
        }
        print(this + ''finished meal, leaving'');
    }
    public String toString() {
        return ''Customer '' + id + '' '';
    }
}

class WaitPerson implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    BlockingQueue<Plate> filledOrders = new LinkedBlockingQueue<Plate>();
    public WaitPerson(Restaurant rest) { restaurant = rest; }
```

```
    public void placeOrder(Customer cust, Food food) {
        try {
            // Shouldn't actually block because this is
            // a LinkedBlockingQueue with no size limit:
            restaurant.orders.put(new Order(cust, this, food));
        } catch(InterruptedException e) {
            print(this + '' placeOrder interrupted'');
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until a course is ready
                Plate plate = filledOrders.take();
                print(this + ''received '' + plate +
                    '' delivering to '' + plate.getOrder().getCustomer());
                plate.getOrder().getCustomer().deliver(plate);
            }
        } catch(InterruptedException e) {
            print(this + '' interrupted'');
        }
        print(this + '' off duty'');
    }
    public String toString() {
        return ''WaitPerson '' + id + '' '';
    }
}

class Chef implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    private final Restaurant restaurant;
    private static Random rand = new Random(47);
    public Chef(Restaurant rest) { restaurant = rest; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until an order appears:
                Order order = restaurant.orders.take();
                Food requestedItem = order.item();
                // Time to prepare order:
                TimeUnit.MILLISECONDS.sleep(rand.nextInt(500));
                Plate plate = new Plate(order, requestedItem);
                order.getWaitPerson().filledOrders.put(plate);
            }
        } catch(InterruptedException e) {
            print(this + '' interrupted'');
        }
        print(this + '' off duty'');
    }
    public String toString() { return ''Chef '' + id + '' ''; }
}

class Restaurant implements Runnable {
    private List<WaitPerson> waitPersons = new ArrayList<WaitPerson>();
    private List<Chef> chefs = new ArrayList<Chef>();
    private ExecutorService exec;
    private static Random rand = new Random(47);
```

```
    BlockingQueue<Order> orders = new LinkedBlockingQueue<Order>();
    public Restaurant(ExecutorService e, int nWaitPersons, int nChefs) {
        exec = e;
        for(int i = 0; i < nWaitPersons; i++) {
            WaitPerson waitPerson = new WaitPerson(this);
            waitPersons.add(waitPerson);
            exec.execute(waitPerson);
        }
        for(int i = 0; i < nChefs; i++) {
            Chef chef = new Chef(this);
            chefs.add(chef);
            exec.execute(chef);
        }
    }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // A new customer arrives; assign a WaitPerson:
                WaitPerson wp = waitPersons.get(rand.nextInt(waitPersons.size()));
                Customer c = new Customer(wp);
                exec.execute(c);
                TimeUnit.MILLISECONDS.sleep(100);
            }
        } catch(InterruptedException e) {
            print(''Restaurant interrupted'');
        }
        print(''Restaurant closing'');
    }
}

public class RestaurantWithQueues {
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        Restaurant restaurant = new Restaurant(exec, 5, 2);
        exec.execute(restaurant);
        if(args.length > 0) // Optional argument
            TimeUnit.SECONDS.sleep(new Integer(args[0]));
        else {
            print(''Press 'Enter' to quit'');
            System.in.read();
        }
        exec.shutdownNow();
    }
} /* Output: (Sample)
      WaitPerson 0 received SPRING__ROLLS delivering to Customer 1
      Customer 1 eating SPRING__ROLLS
      WaitPerson 3 received SPRING__ROLLS delivering to Customer 0
      Customer 0 eating SPRING__ROLLS
      WaitPerson 0 received BURRITO delivering to Customer 1
      Customer 1 eating BURRITO
      WaitPerson 3 received SPRING__ROLLS delivering to Customer 2
      Customer 2 eating SPRING__ROLLS
      WaitPerson 1 received SOUP delivering to Customer 3
      Customer 3 eating SOUP
      WaitPerson 3 received VINDALOO delivering to Customer 0
      Customer 0 eating VINDALOO
      WaitPerson 0 received FRUIT delivering to Customer 1
      ...
```

```
*///:~
```

[295] Eine wichtige Beobachtung an diesem Beispiel betrifft die Abbildung der komplexen Kommunikation zwischen den einzelnen Aufgaben auf die Warteschlangen. Dieser Ansatz vereinfacht die Threadprogrammierung deutlich, in dem die Steuerung umgekehrt wird: Die Aufgaben kommunizieren nicht direkt miteinander, sondern übergeben sich gegenseitig über Warteschlangen Objekte. Die empfangende Aufgabe verarbeitet das Objekt indem es dieses wie eine Benachrichtigung behandelt, statt daß ihm eine Benachrichtigung aufgedrängt wird. Wenn Sie diese Vorgehensweise so oft wie möglich anwenden, haben Sie erheblich bessere Aussichten, robuste threadbasierte Anwendungen zu entwickeln.

Übungsaufgabe 36: (10) Ändern Sie das Beispiel *RestaurantWithQueues.java* so, daß jeder Tisch über ein `OrderTicket`-Objekt verfügt. Ersetzen Sie `order` durch `orderTicket` und schreiben Sie eine `Table`-Klassen mit mehreren Gästen (`Customer`-Objekten) pro Tisch. ■

22.8.3 Arbeitsteilung

[296] Die nächste Simulation verknüpft viele in diesem Kapitel vorgestellte Konzepte. Stellen Sie sich eine robotische Fertigungsstraße für Autos vor. Jedes Auto (`Car`-Objekt) durchläuft mehrere Fertigungsschritte: Bau des Fahrgestells, Einsetzen des Motors und des Antriebsstrangs (*drive train*) sowie der Räder.

```
//: concurrency/CarBuilder.java
// A complex example of tasks working together.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

class Car {
    private final int id;
    private boolean
        engine = false, driveTrain = false, wheels = false;
    public Car(int idn) { id = idn; }
    // Empty Car object:
    public Car() { id = -1; }
    public synchronized int getId() { return id; }
    public synchronized void addEngine() { engine = true; }
    public synchronized void addDriveTrain() {
        driveTrain = true;
    }
    public synchronized void addWheels() { wheels = true; }
    public synchronized String toString() {
        return ”Car ” + id + ” [” + ” engine: ” + engine
            + ” driveTrain: ” + driveTrain
            + ” wheels: ” + wheels + ” ]”;
    }
}

class CarQueue extends LinkedBlockingQueue<Car> {}

class ChassisBuilder implements Runnable {
    private CarQueue carQueue;
    private int counter = 0;
    public ChassisBuilder(CarQueue cq) { carQueue = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
```

```
                TimeUnit.MILLISECONDS.sleep(500);
                // Make chassis:
                Car c = new Car(counter++);
                print("ChassisBuilder created " + c);
                // Insert into queue
                carQueue.put(c);
            }
        } catch(InterruptedException e) {
            print("Interrupted: ChassisBuilder");
        }
        print("ChassisBuilder off");
    }
}

class Assembler implements Runnable {
    private CarQueue chassisQueue, finishingQueue;
    private Car car;
    private CyclicBarrier barrier = new CyclicBarrier(4);
    private RobotPool robotPool;
    public Assembler(CarQueue cq, CarQueue fq, RobotPool rp) {
        chassisQueue = cq;
        finishingQueue = fq;
        robotPool = rp;
    }
    public Car car() { return car; }
    public CyclicBarrier barrier() { return barrier; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                // Blocks until chassis is available:
                car = chassisQueue.take();
                // Hire robots to perform work:
                robotPool.hire(EngineRobot.class, this);
                robotPool.hire(DriveTrainRobot.class, this);
                robotPool.hire(WheelRobot.class, this);
                barrier.await(); // Until the robots finish
                // Put car into finishingQueue for further work
                finishingQueue.put(car);
            }
        } catch(InterruptedException e) {
            print("Exiting Assembler via interrupt");
        } catch(BrokenBarrierException e) {
            // This one we want to know about
            throw new RuntimeException(e);
        }
        print("Assembler off");
    }
}

class Reporter implements Runnable {
    private CarQueue carQueue;
    public Reporter(CarQueue cq) { carQueue = cq; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                print(carQueue.take());
            }
        } catch(InterruptedException e) {
```

```
            print("Exiting Reporter via interrupt");
        }
        print("Reporter off");
    }
}

abstract class Robot implements Runnable {
    private RobotPool pool;
    public Robot(RobotPool p) { pool = p; }
    protected Assembler assembler;
    public Robot assignAssembler(Assembler assembler) {
        this.assembler = assembler;
        return this;
    }
    private boolean engage = false;
    public synchronized void engage() {
        engage = true;
        notifyAll();
    }
    // The part of run() that's different for each robot:
    abstract protected void performService();
    public void run() {
        try {
            powerDown(); // Wait until needed
            while(!Thread.interrupted()) {
                performService();
                assembler.barrier().await(); // Synchronize
                // We're done with that job...
                powerDown();
            }
        } catch(InterruptedException e) {
            print("Exiting " + this + " via interrupt");
        } catch(BrokenBarrierException e) {
            // This one we want to know about
            throw new RuntimeException(e);
        }
        print(this + " off");
    }
    private synchronized void powerDown() throws InterruptedException {
        engage = false;
        assembler = null; // Disconnect from the Assembler
        // Put ourselves back in the available pool:
        pool.release(this);
        while(engage == false) // Power down
            wait();
    }
    public String toString() { return getClass().getName(); }
}

class EngineRobot extends Robot {
    public EngineRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + " installing engine");
        assembler.car().addEngine();
    }
}

class DriveTrainRobot extends Robot {
    public DriveTrainRobot(RobotPool pool) { super(pool); }
```

```
    protected void performService() {
        print(this + '' installing DriveTrain'');
        assembler.car().addDriveTrain();
    }
}

class WheelRobot extends Robot {
    public WheelRobot(RobotPool pool) { super(pool); }
    protected void performService() {
        print(this + '' installing Wheels'');
        assembler.car().addWheels();
    }
}

class RobotPool {
    // Quietly prevents identical entries:
    private Set<Robot> pool = new HashSet<Robot>();
    public synchronized void add(Robot r) {
        pool.add(r);
        notifyAll();
    }
    public synchronized void
        hire(Class<? extends Robot> robotType, Assembler d)
        throws InterruptedException {
        for(Robot r : pool)
            if(r.getClass().equals(robotType)) {
                pool.remove(r);
                r.assignAssembler(d);
                r.engage(); // Power it up to do the task
                return;
            }
        wait(); // None available
        hire(robotType, d); // Try again, recursively
    }
    public synchronized void release(Robot r) { add(r); }
}

public class CarBuilder {
    public static void main(String[] args) throws Exception {
        CarQueue chassisQueue = new CarQueue(),
            finishingQueue = new CarQueue();
        ExecutorService exec = Executors.newCachedThreadPool();
        RobotPool robotPool = new RobotPool();
        exec.execute(new EngineRobot(robotPool));
        exec.execute(new DriveTrainRobot(robotPool));
        exec.execute(new WheelRobot(robotPool));
        exec.execute(new Assembler(chassisQueue, finishingQueue, robotPool));
        exec.execute(new Reporter(finishingQueue));
        // Start everything running by producing chassis:
        exec.execute(new ChassisBuilder(chassisQueue));
        TimeUnit.SECONDS.sleep(7);
        exec.shutdownNow();
    }
} /* (Execute to see output) *////:~
```

[297] Die Autos (`Car`-Objekte) werden mit Hilfe der Warteschlange (`CarQueue`, abgeleitet von `LinkedBlockingQueue`) von einem Fertigungsprozeß zum nächsten transportiert. Die Klasse `ChassisBuilder` erzeugt rudimentäre `Car`-Objekte und platziert sie in der `CarQueue`-Warteschlange. Die

Klasse `Assembler` entnimmt einzelne `Car`-Objekte aus der Warteschlange und beauftragt Roboter (`Robot`-Objekte) mit den einzelnen Fertigungsschritten. Ein `CyclicBarrier`-Objekt gestattet der Klasse `Assembler`, zu warten, bis die einzelnen Roboter fertig sind, woraufhin sie die `Car`-Objekte über eine zweite `CarQueue`-Warteschlange zur nächsten Operation transportiert. Die Klasse `Reporter` ist der Verbraucher dieser zweiten Warteschlange und zeigt das `Car`-Objekt an, um zu dokumentieren, daß alle Aufgaben ordentlich verarbeitet wurden.

[298] Die Roboter werden in einem Pool bevorratet. Wenn ein Arbeitsschritt anfällt, wird ein Roboter des entsprechenden Typs aus dem Pool gewählt und nach beendeter Arbeit wieder an den Pool zurückgegeben.

[299] Die `main()`-Methode erzeugt alle benötigten Objekte und initialisiert die Aufgaben, wobei die `ChassisBuilder`-Aufgabe zuletzt gestartet wird. (Durch das Verhalten von Objekten der Klasse `LinkedBlockingQueue` hätten wir die Aufgabe auch früher starten können.) Beachten Sie, daß dieses Beispiel alle in diesem Kapitel vorgestellten Richtlinien hinsichtlich des Lebenszyklus' von Objekten und Aufgaben einhält, so daß das Programm sicher beendet wird.

[300] Beachten Sie, daß alle Methoden der Klasse `Car` synchronisiert sind. Die Synchronisierung ist *in diesem Beispiel* redundant, sich die Autos in Warteschlangen durch die Fabrik bewegen und stets nur ein Roboter an einem Auto arbeiten kann. Die Warteschlangen erzwingen sequentiellen Zugriff auf die Autos. Aber genau hier ist die Falle: Sie können sich für die Optimierung entscheiden und die nicht benötigte Synchronisierung fortlassen. Die Anwendung scheitert allerdings, wenn sie später mit einer anderen kombiniert wird, die Synchronisierung voraussetzt.

[301] Brian Goetz kommentiert:

> Es ist leichter, zu verlangen, daß ein `Car`-Objekt von mehreren Threads beansprucht werden kann und daher offensichtlich threadsicher sein muß. Ich charakterisiere diesen Ansatz so: In einem öffentlichen Park sind gefährliche Stellen mit einem Geländer und dem Hinweis „Nicht auf das Geländer stützen!“ gesichert. Der Zweck dieser Regel besteht selbstverständlich nicht darin, Sie davon abzuhalten, sich auf das Geländer zu stützen, sondern zu verhindern, daß Sie von der Klippe stürzen. Die Regel „Nicht auf das Geländer stützen!“ läßt sich aber leichter befolgen, als „Nicht von der Klippe stürzen!“

Übungsaufgabe 37: (2) Ändern Sie das Beispiel *CarBuilder.java* so, daß der Fertigungsprozeß einen weiteren Schritt beinhaltet. Bauen Sie die Abgasanlage (*exhaust system*), die Sitze und Kotflügel (*fenders*) ein. Wie beim zweiten Fertigungsschritt können die Vorgänge simultan durch Roboter verarbeitet werden. ■

Übungsaufgabe 38: (3) Modellieren Sie das Bauen eines Hauses (siehe Anfang von Abschnitt 22.5) nach dem Muster von Beispiel *CarBuilder.java*. ■

22.9 Leistungssteigerung

[302] Die seit der SE 5 vorhandene Bibliothek im Package `java.util.concurrent` beinhaltet eine Reihe von Klassen, mit denen sich die Performanz eines Programms verbessern läßt. Es ist beim Durchsehen der Bibliothek schwierig, zu erkennen, welche Klassen für den regulären Gebrauch (zum Beispiel *`BlockingQueue`*) und welche zur Verbesserung der Performanz gedacht sind. In diesem Abschnitt besprechen wir einige Aspekte der Performanzsteigerung und die damit verbundenen Klassen.

22.9.1 Verschiedene Sperrmechanismen im Vergleich

[303] Java bietet sowohl die traditionelle Synchronisierung (Schlüsselwort `synchronized`) als auch die neue Klasse *`Lock`* und die atomaren Klassen im Package `java.util.concurrent.atomic` (beides ab der SE 5). Wir wollen die verschiedenen Konzepte in diesem Unterschied miteinander vergleichen, um ihren jeweiligen Wert und ihre bestimmungsgemäße Verwendung zu verstehen.

[304] The naive approach is to try a simple test on each approach, like this:

```
//: concurrency/SimpleMicroBenchmark.java
// The dangers of microbenchmarking.
import java.util.concurrent.locks.*;

abstract class Incrementable {
    protected long counter = 0;
    public abstract void increment();
}

class SynchronizingTest extends Incrementable {
    public synchronized void increment() { ++counter; }
}

class LockingTest extends Incrementable {
    private Lock lock = new ReentrantLock();
    public void increment() {
        lock.lock();
        try {
            ++counter;
        } finally {
            lock.unlock();
        }
    }
}

public class SimpleMicroBenchmark {
    static long test(Incrementable incr) {
        long start = System.nanoTime();
        for(long i = 0; i < 10000000L; i++)
            incr.increment();
        return System.nanoTime() - start;
    }
    public static void main(String[] args) {
        long synchTime = test(new SynchronizingTest());
        long lockTime = test(new LockingTest());
        System.out.printf("synchronized: %1$10d\n", synchTime);
        System.out.printf("Lock: %1$10d\n", lockTime);
        System.out.printf("Lock/synchronized = %1$.3f",
                          (double)lockTime/(double)synchTime);
    }
} /* Output: (75% match)
      synchronized: 244919117
      Lock: 939098964
      Lock/synchronized = 3.834
   *///:~
```

Die Ausgabe dokumentiert, daß eine traditionell synchronisierte Methode scheinbar schneller verarbeitet wird, als eine Methode, die mittels eines Sperrobjektes vom Typ (`ReentrantLock`) synchronisiert ist. Wie ist das möglich?

[305] Das obige Beispiel veranschaulicht die Gefahren des sogenannten „Microbenchmarking“.[25] Dieser Begriff bezeichnet das isolierte Testen der Performanz einer Eigenschaft außerhalb ihres Kontextes. Selbstverständlich müssen Sie Testprogramme schreiben, um Aussagen wie „die Synchronisierung per Sperrobjekt ist schneller als die traditionelle Synchronisierung per `synchronized`“ verifizieren zu können. Sie brauchen aber ein Bewußtsein für die Dinge, die während der Übersetzungs- beziehungsweise Laufzeit eines Programms geschehen, wenn Sie einen solchen Test schreiben.

[306] Das Beispiel *SimpleMicroBenchmark.java* hat mehrere Schwachstellen. Erstens zeigt sich der tatsächliche Performanzunterschied beider Ansätze nur, wenn die synchronisierten Stellen *unter Wettbewerbsbedingungen* getestet werden, das heißt, wenn mehrere Aufgaben versuchen, Zugriff auf den kritischen Abschnitt des Quelltextes zu erhalten. Beim obigen Beispiel wird jeder Sperrmechanismus isoliert und mit nur einem einzigen Thread (`main`-Thread) getestet.

[307] Zweitens kann der Compiler spezielle Optimierungen vornehmen, wenn das Schlüsselwort `synchronized` vorhanden ist, und erkennt eventuell auch, daß das Programm nur einen einzigen Thread verwendet. Der Compiler erkennt möglicherweise sogar, daß das Feld `counter` eine feste Anzahl mal inkrementiert wird und berechnet das Ergebnis im Voraus. Verschiedene Compiler und Laufzeitumgebungen können sich unterschiedlich verhalten, so daß schwer vorherzusehen ist, was passieren wird. Wir müssen aber dafür sorgen, daß der Compiler das Ergebnis nicht vorherbestimmen kann.

[308] Das Programm muß komplizierter sein, um ein aussagefähiges Testergebnis zu erhalten. Zunächst brauchen wir mehrere Aufgaben und zwar solche, die interne Werte nicht nur ändern, sondern auch solche, die sie lesen (andernfalls bemerkt der Compiler im Rahmen der Optimierung eventuell, daß die Werte nirgends gebraucht werden). Außerdem muß die Berechnung kompliziert und hinreichend schwer vorhersehbar sein, damit der Compiler keine aggressiven Optimierungen durchführen kann. Dies wird gewährleistet, in dem wir ein großes Array zufällig bestimmter `int`-Werte vordefinieren (und dadurch den Einfluß der `Random`-Methode `nextInt()` in den Hauptschleifen vermeiden) und diese Werte anschließend aufaddieren:

```
//: concurrency/SynchronizationComparisons.java
// Comparing the performance of explicit Locks
// and Atomics versus the synchronized keyword.
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

abstract class Accumulator {
    public static long cycles = 50000L;
    // Number of Modifiers and Readers during each test:
    private static final int N = 4;
    public static ExecutorService exec = Executors.newFixedThreadPool(N*2);
    private static CyclicBarrier barrier = new CyclicBarrier(N*2 + 1);
    protected volatile int index = 0;
    protected volatile long value = 0;
    protected long duration = 0;
    protected String id = ''error'';
    protected final static int SIZE = 100000;
    protected static int[] preLoaded = new int[SIZE];
    static {
```

[25] Brian Goetz' Erläuterungen haben mir sehr geholfen, diese Dinge zu verstehen. Mehr über Performanzmessungen finden Sie in Brians Artikel „Java theory and practice: Dynamic compilation and performance measurement. The perils of benchmarking under dynamic compilation.“ unter der Webadresse *https://www.ibm.com/developerworks/library/j-jtp12214/*.

```
        // Load the array of random numbers:
        Random rand = new Random(47);
        for(int i = 0; i < SIZE; i++)
            preLoaded[i] = rand.nextInt();
    }
    public abstract void accumulate();
    public abstract long read();
    private class Modifier implements Runnable {
        public void run() {
            for(long i = 0; i < cycles; i++)
                accumulate();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    private class Reader implements Runnable {
        private volatile long value;
        public void run() {
            for(long i = 0; i < cycles; i++)
                value = read();
            try {
                barrier.await();
            } catch(Exception e) {
                throw new RuntimeException(e);
            }
        }
    }
    public void timedTest() {
        long start = System.nanoTime();
        for(int i = 0; i < N; i++) {
            exec.execute(new Modifier());
            exec.execute(new Reader());
        }
        try {
            barrier.await();
        } catch(Exception e) {
            throw new RuntimeException(e);
        }
        duration = System.nanoTime() - start;
        printf("%-13s: %13d\n", id, duration);
    }
    public static void
        report(Accumulator acc1, Accumulator acc2) {
        printf("%-22s: %.2f\n", acc1.id + "/" + acc2.id,
               (double)acc1.duration/(double)acc2.duration);
    }
}

class BaseLine extends Accumulator {
    { id = "BaseLine"; }
    public void accumulate() {
        value += preLoaded[index++];
        if (index >= SIZE) index = 0;
    }
    public long read() { return value; }
```

```
}

class SynchronizedTest extends Accumulator {
    { id = ''synchronized''; }
    public synchronized void accumulate() {
        value += preLoaded[index++];
        if(index >= SIZE) index = 0;
    }
    public synchronized long read() {
        return value;
    }
}

class LockTest extends Accumulator {
    { id = ''Lock''; }
    private Lock lock = new ReentrantLock();
    public void accumulate() {
        lock.lock();
        try {
            value += preLoaded[index++];
            if(index >= SIZE) index = 0;
        } finally {
            lock.unlock();
        }
    }
    public long read() {
        lock.lock();
        try {
            return value;
        } finally {
            lock.unlock();
        }
    }
}

class AtomicTest extends Accumulator {
    { id = ''Atomic''; }
    private AtomicInteger index = new AtomicInteger(0);
    private AtomicLong value = new AtomicLong(0);
    public void accumulate() {
        // Oops! Relying on more than one Atomic at
        // a time doesn't work. But it still gives us
        // a performance indicator:
        int i = index.getAndIncrement();
        value.getAndAdd(preLoaded[i]);
        if(++i >= SIZE)
            index.set(0);
    }
    public long read() { return value.get(); }
}

public class SynchronizationComparisons {
    static BaseLine baseLine = new BaseLine();
    static SynchronizedTest synch = new SynchronizedTest();
    static LockTest lock = new LockTest();
    static AtomicTest atomic = new AtomicTest();
    static void test() {
        print(''============================'');
        printf(''%-12s : %13d\n'', ''Cycles'', Accumulator.cycles);
```

```
        baseLine.timedTest();
        synch.timedTest();
        lock.timedTest();
        atomic.timedTest();
        Accumulator.report(synch, baseLine);
        Accumulator.report(lock, baseLine);
        Accumulator.report(atomic, baseLine);
        Accumulator.report(synch, lock);
        Accumulator.report(synch, atomic);
        Accumulator.report(lock, atomic);
    }
    public static void main(String[] args) {
        int iterations = 5; // Default
        if(args.length > 0) // Optionally change iterations
            iterations = new Integer(args[0]);
        // The first time fills the thread pool:
        print("Warmup");
        baseLine.timedTest();
        // Now the initial test doesn't include the cost
        // of starting the threads for the first time.
        // Produce multiple data points:
        for(int i = 0; i < iterations; i++) {
            test();
            Accumulator.cycles *= 2;
        }
        Accumulator.exec.shutdown();
    }
} /* Output: (Sample)
      Warmup
      BaseLine : 34237033
      ============================
      Cycles : 50000
      BaseLine : 20966632
      synchronized : 24326555
      Lock : 53669950
      Atomic : 30552487
      synchronized/BaseLine : 1.16
      Lock/BaseLine : 2.56
      Atomic/BaseLine : 1.46
      synchronized/Lock : 0.45
      synchronized/Atomic : 0.79
      Lock/Atomic : 1.76
      ============================
      Cycles : 100000
      BaseLine : 41512818
      synchronized : 43843003
      Lock : 87430386
      Atomic : 51892350
      synchronized/BaseLine : 1.06
      Lock/BaseLine : 2.11
      Atomic/BaseLine : 1.25
      synchronized/Lock : 0.50
      synchronized/Atomic : 0.84
      Lock/Atomic : 1.68
      ============================
      Cycles : 200000
      BaseLine : 80176670
```

```
synchronized : 5455046661
Lock : 177686829
Atomic : 101789194
synchronized/BaseLine : 68.04
Lock/BaseLine : 2.22
Atomic/BaseLine : 1.27
synchronized/Lock : 30.70
synchronized/Atomic : 53.59
Lock/Atomic : 1.75
============================
Cycles : 400000
BaseLine : 160383513
synchronized : 780052493
Lock : 362187652
Atomic : 202030984
synchronized/BaseLine : 4.86
Lock/BaseLine : 2.26
Atomic/BaseLine : 1.26
synchronized/Lock : 2.15
synchronized/Atomic : 3.86
Lock/Atomic : 1.79
============================
Cycles : 800000
BaseLine : 322064955
synchronized : 336155014
Lock : 704615531
Atomic : 393231542
synchronized/BaseLine : 1.04
Lock/BaseLine : 2.19
Atomic/BaseLine : 1.22
synchronized/Lock : 0.47
synchronized/Atomic : 0.85
Lock/Atomic : 1.79
============================
Cycles : 1600000
BaseLine : 650004120
synchronized : 52235762925
Lock : 1419602771
Atomic : 796950171
synchronized/BaseLine : 80.36
Lock/BaseLine : 2.18
Atomic/BaseLine : 1.23
synchronized/Lock : 36.80
synchronized/Atomic : 65.54
Lock/Atomic : 1.78
============================
Cycles : 3200000
BaseLine : 1285664519
synchronized : 96336767661
Lock : 2846988654
Atomic : 1590545726
synchronized/BaseLine : 74.93
Lock/BaseLine : 2.21
Atomic/BaseLine : 1.24
synchronized/Lock : 33.84
synchronized/Atomic : 60.57
Lock/Atomic : 1.79
```

```
*///:~
```

[309] Dieses Programm implementiert das Entwurfsmuster *Templatemethod*,[26] um die allgemeingültigen Teile des Quelltextes in einer Basisklasse zu platzieren, während die variablen Komponenten in den Methoden `accumulate()` und `read()` der abgeleiteten Klassen untergebracht sind. Sie sehen in jeder der drei abgeleiteten Klassen `SynchronizedTest`, `LockTest` und `AtomicTest`, wie die verschiedenen Verfahren für den gegenseitigen Ausschluß von Threads in den Methoden `accumulate()` und `read()` implementiert sind.

[310] Die Aufgaben werden in diesem Programm von Threads verarbeitet, die aus einem „fixed Threadpool“ (`newFixedThreadPool()`) stammen, damit das Erzeugen der Threads zu Beginn stattfindet und zusätzliche Unkosten während der Tests vermieden werden. Aus diesem Grund wird der erste Test zweimal ausgeführt und das erste Ergebnis verworfen, da es die Erzeugung der Threads beinhaltet.

[311] Das `CyclicBarrier`-Objekt ist notwendig, um zu gewährleisten, daß alle Aufgaben vollständig verarbeitet sind, bevor ein Test als beendet erklärt wird.

[312] Der statische Initialisierungsblock in der Klasse `Accumulator` dient dazu, vor dem Testbeginn ein Array mit Zufallszahlen zu initialisieren. Auf diese Weise schlagen eventuelle Unkosten beim Erzeugen der Zufallszahlen während des Tests nicht zu Buche.

[313] Die `accumulate()`-Methode rückt bei jedem Aufruf ein Arrayelement vorwärts und addiert eine weitere zufällig generierte Zahl zum Inhalt des `value`-Feldes. Die `Modifier`- und `Reader`-Aufgaben sorgen für „Wettbewerbsbedingungen“ beim Zugriff auf das `Accumulator`-Objekt.

[314] Beachten Sie in der Klasse `AtomicTest` den Hinweis, daß die Situation zu komplex für atomare Objekte ist. Umfaßt eine Operation mehr als ein atomares Objekt, so sind Sie wahrscheinlich gezwungen, aufzugeben und den gegenseitigen Ausschluß mit Hilfe eines anderen Verfahrens zu erwirken (die Dokumentation des Java Development Kits sagt aus, daß die Verwendung atomarer Objekte nur dann im Sinne des Erfinders funktioniert, wenn sich die kritische Aktualisierungsoperation auf ein einziges Feld beschränkt). Ich habe mich entschieden, den Test trotzdem stehen zu lassen, damit Sie ein Gefühl für den Performanzgewinn durch atomare Objekte bekommen.

[315] Die `main()`-Methode führt den Test mehrmals durch, wobei Sie die Voreinstellung von fünf Iterationen per Kommandozeilenargument überschreiben können. Die Anzahl der Testzyklen wird von Iteration zu Iteration verdoppelt, so daß Sie das Verhalten der verschiedenen Sperrmechanismen bei wachsender Laufzeit beobachten können. Wie Sie an den Ausgaben sehen, ist das Ergebnis recht überraschend. Bei den ersten vier Iterationen scheint die traditionelle Synchronisierung per `synchronized` effizienter zu sein als Sperr- oder atomare Objekte. Aber plötzlich wird ein Grenzwert überschritten und die traditionelle Sychronisierung wird ineffizient, während das Verhältnis des Sperr- beziehungsweise des atomaren Objektes zum Referenztest (`BaseLine`) ungefähr gleich bleibt und somit erheblich effizienter wird, als die traditionelle Synchronisierung.

[316] Bedenken Sie, daß dieses Programm nur einen Anhaltspunkt hinsichtlich der Unterschiede zwischen den verschiedenen Sperrmechanismen liefert und die obigen Ausgaben spezifisch für meinen Rechner und die Bedingungen sind, unter denen ich den Test durchgeführt habe. Wenn Sie mit dem Programm experimentieren, werden Sie feststellen, daß sich das Verhalten abhängig von der Anzahl von Threads und der Laufzeit deutlich verändern kann. Manche Hotspot-Optimierungen zur Laufzeit treten erst in Kraft, wenn das Programm eine Zeit lang gelaufen ist, bei Serverprogrammen sogar mehrere Stunden.

[317] Dennoch zeigt sich recht deutlich, daß Sperrobjekte in der Regel erheblich effizienter sind, als traditionelle Synchronisierung per `synchronized`. Außerdem scheinen die Unkosten durch tra-

[26] Siehe *Thinking in Patterns* (*http://www.mindview.net*).

ditionelle Synchronisierung erheblich zu schwanken, während sie bei Sperrobjekten etwa konstant ausfallen.

[318] Bedeutet das, daß Sie die traditionelle Synchronisierung per `synchronized` nicht benutzen sollten? Es gibt zwei Faktoren zu berücksichtigen: Erstens sind die synchronisierten Methoden im obigen Beispiel sehr klein. Es ist im allgemeinen sinnvoll, kritische Abschnitte auf das absolut notwendige zu beschränken. Andererseits kann ein kritischer Abschnitt aber auch umfangreicher sein, als im obigen Beispiel, so daß die Verarbeitung der dortigen Anweisungen signifikant mehr Zeit beansprucht, als die Unkosten durch den Ein- und Austritt in diesen Bereich, der Vorteil durch den schnelleren Sperrmechanismus also nebensächlich wird. Die einzige Möglichkeit, dies herauszufinden, besteht natürlich darin, die verschiedenen Verfahren und ihre Auswirkungen auszuprobieren, wenn (und nur dann) Sie an der Performanz Ihrer Anwendung arbeiten.

[319] Zweitens wird beim Lesen des Quelltextes in diesem Kapitel ersichtlich, daß die tradionelle Synchronisierung per `synchronized` zu einem erheblich leichter zu lesenden Quelltext führt, als die `lock()`/`try`/`finally`/`unlock()`-Kombinationen, die beim Einsatz von Sperrobjekten benötigt werden. Aus diesem Grund verwende ich in diesem Kapitel `synchronized`. Wie ich schon mehrmals in diesem Buch erwähnt habe, wird Quelltext viel häufiger gelesen als geschrieben. Es ist beim Programmieren wichtiger, mit anderen Menschen zu kommunizieren als mit dem Computer, kurz, die Lesbarkeit des Quelltextes ist ausschlaggebend. Es ist daher sinnvoll, mit traditioneller Synchronisierung zu beginnen und nur dann auf Sperrobjekte umzusteigen, wenn Sie die Performanz Ihrer Anwendung verbessern möchten.

[320] Es ist nett, wenn Sie in einer threadbasierten Anwendung atomare Klassen verwenden können. Denken Sie aber daran, daß, wie wir im Beispiel *SynchronizationComparisons.java* gesehen haben, Objekte atomarer Klassen nur in sehr einfachen Fällen nützlich sind, in der Regel dann, wenn nur ein solches Objekt modifiziert wird, welches außerdem von allen anderen Objekt unabhängig ist. Es ist sicherer, mit einem traditionelleren Sperrmechanismus zu beginnen und nur dann atomare Klassen zu verwenden, wenn Sie die Performanz Ihrer Anwendung verbessern möchten.

22.9.2 Nicht-blockierende Container

[321] In Kapitel 12 wurde betont, daß Container in allen Bereichen der Programmierung ein fundamentales Werkzeug sind. Dies gilt auch für die Threadprogrammierung. Daher hatten die frühen Container, wie `Vector` und `Hashtable`, viele synchronisierte Methoden, die allerdings inakzeptable Unkosten bewirkten, wenn diese Klassen außerhalb des Rahmens der Threadprogrammierung verwendet wurden. In Java 1.2 war die neue Containerbibliothek unsynchronisiert und die Klasse `Collections` erhielt verschiedene statische Dekoratormethoden, mit denen die einzelnen Containertypen synchronisiert werden konnten. Obwohl eine Verbesserung, da Sie nun die Wahl hatten, ob Sie Ihren Container mit oder ohne Synchronisierung verwenden wollten, war noch immer das synchronisierungsbedingte Sperren für die Unkosten verantwortlich. Die SE 5 ergänzte die Bibliothek um einige speziell auf verbesserte threadsichere Performanz ausgerichtete Container, bei denen aufgrund ausgeklügelter Technik keine Sperrung mehr erforderlich ist.

[322] Die Funktionsweise dieser nicht-blockierenden Container besteht darin, daß Änderungen am Inhalt des Containers gleichzeitig mit Leseoperationen stattfinden können, solange die Leseoperationen nur die Ergebnisse *abgeschlossener* Änderungsoperationen „sehen“ können. Die Änderung wird an einer separaten Kopie eines Teils der Datenstruktur (in manchen Fällen auch der gesamten Struktur) durchgeführt, die während des Änderungsvorgangs unsichtbar ist. Erst wenn die Änderung vollständig ist, wird die modifizierte Datenstruktur atomar gegen den entsprechenden Teil der Hauptdatenstruktor ausgetauscht und anschließend ist die Änderung für Leseoperationen sichtbar.

[323] Bei der Klasse `CopyOnWriteArrayList` bewirkt eine Schreiboperation, daß eine Kopie des gesamten unterliegenden Arrays erzeugt wird. Das ursprüngliche Array bleibt an Ort und Stelle, so daß Leseoperationen sicher bearbeitet werden können, während die Kopie des Arrays modifiziert wird. Ist die Änderungsoperation abgeschlossen, so setzt eine atomare Operation das neue Array an die Stelle des alten, so daß Leseoperationen von nun an die aktualisierte Information erhalten. Zu den Vorteilen der Klasse `CopyOnWriteArrayList` gehört, daß sie keine Ausnahme vom Typ `java.util.ConcurrentModificationException` auswerfen, wenn mehrere Iteratoren die Liste durchlaufen, während ihr Inhalt modifiziert wird, das heißt, es sind, im Gegensatz zu früher, keine speziellen Anweisungen notwendig, um sich gegen solche Ausnahmen zu schützen. Die Klasse `CopyOnWriteArraySet` stützt sich auf `CopyOnWriteArrayList`, um ihr nicht-blockierendes Verhalten zu implementieren.

[324] Die Klassen `ConcurrentHashMap` und `ConcurrentLinkedQueue` verwenden einen ähnlichen Mechanismus, um gleichzeitiges Lesen und Schreiben zu ermöglichen, wobei anstelle des gesamten Containers aber nur echte Teile kopiert und modifiziert werden. Aber auch hier ist eine Änderungsoperation erst nach ihrem Abschluß für Leseoperationen sichtbar. Die Klasse `ConcurrentHashMap` wirft keine Ausnahmen vom Typ `ConcurrentModificationException` aus.

22.9.2.1 Synchronisierte ArrayList und CopyOnWriteArrayList im Vergleich

[325] Solange Sie den Inhalt eines nicht-blockierenden Containers nur abfragen, sind die Operationen erheblich schneller als beim entsprechenden synchronizierten Typ, da die Unkosten durch das Akquirieren und Aufheben der Sperren wegfallen. Dies gilt auch noch für eine kleine Anzahl von Änderungsoperationen bei einem nicht-blockierenden Container, wobei es interessant wäre, ein Gefühl dafür zu bekommen, was „klein“ bedeutet. Wir erarbeiten in diesem Unterunterabschnitt einen groben Eindruck von den Performanzunterschieden zwischen diesen Containern unter verschiedenen Bedingungen.

[326] Wir beginnen mit einem generischen Framework für Leistungstests an einem Container beliebigen Typs, insbesondere vom Typ *`Map`*. Der generische Typparameter `C` repräsentiert den Typ des Containers:

```
//: concurrency/Tester.java
// Framework to test performance of concurrency containers.
import java.util.concurrent.*;
import net.mindview.util.*;

public abstract class Tester<C> {
    static int testReps = 10;
    static int testCycles = 1000;
    static int containerSize = 1000;
    abstract C containerInitializer();
    abstract void startReadersAndWriters();
    C testContainer;
    String testId;
    int nReaders;
    int nWriters;
    volatile long readResult = 0;
    volatile long readTime = 0;
    volatile long writeTime = 0;
    CountDownLatch endLatch;
    static ExecutorService exec = Executors.newCachedThreadPool();
    Integer[] writeData;
    Tester(String testId, int nReaders, int nWriters) {
```

```
        this.testId = testId + '' '' + nReaders + ''r '' + nWriters + ''w'';
        this.nReaders = nReaders;
        this.nWriters = nWriters;
        writeData = Generated.array(Integer.class,
            new RandomGenerator.Integer(), containerSize);
        for(int i = 0; i < testReps; i++) {
            runTest();
            readTime = 0;
            writeTime = 0;
        }
    }
    void runTest() {
        endLatch = new CountDownLatch(nReaders + nWriters);
        testContainer = containerInitializer();
        startReadersAndWriters();
        try {
            endLatch.await();
        } catch(InterruptedException ex) {
            System.out.println(''endLatch interrupted'');
        }
        System.out.printf(''%-27s %14d %14d\n'', testId, readTime, writeTime);
        if(readTime != 0 && writeTime != 0)
            System.out.printf(''%-27s %14d\n'',
                ''readTime + writeTime ='', readTime + writeTime);
    }
    abstract class TestTask implements Runnable {
        abstract void test();
        abstract void putResults();
        long duration;
        public void run() {
            long startTime = System.nanoTime();
            test();
            duration = System.nanoTime() - startTime;
            synchronized(Tester.this) {
                putResults();
            }
            endLatch.countDown();
        }
    }
    public static void initMain(String[] args) {
        if(args.length > 0)
            testReps = new Integer(args[0]);
        if(args.length > 1)
            testCycles = new Integer(args[1]);
        if(args.length > 2)
            containerSize = new Integer(args[2]);
        System.out.printf(''%-27s %14s %14s\n'', ''Type'', ''Read time'', ''Write time'');
    }
} ///:~
```

[327] Die abstrakte Methode `containerInitializer()` gibt eine Referenz auf den initialisierten zu testenden Container zurück, die im `testContainer`-Feld gespeichert wird. Die abstrakte Methode `startReadersAndWriters()` startet die lesenden und schreibenden Aufgaben, die den Containerinhalt während des Tests abfragen und ändern. Es werden verschiedene Tests mit variierenden Anzahlen von lesenden und schreibenden Aufgaben verarbeitet, um die Auswirkungen des Wettbewerbs um die Sperre (bei synchronisierten Containern) beziehungsweise der Schreiboperationen (bei nicht-blockierenden Containern) zu beobachten.

[328] Der Konstruktor erwartet verschiedene Parameter des angestrebten Tests (die Namen der Bezeichner sollten als Erklärung genügen) und ruft `testReps`-mal die `runTest()`-Methode auf. Die `runTest()`-Methode erzeugt ein `CountDownLatch`-Objekt (damit der Test feststellen kann, wann die Verarbeitung aller Aufgaben beendet ist), initialisiert den Container, ruft `startReadersAndWriters()` auf und wartet bis alle Aufgaben fertig verarbeitet sind.

[329] Alle Klassen für lesende beziehungsweise schreibende Aufgaben sind von der abstrakten Klasse `TestTask` abgeleitet, welche die Verarbeitungsdauer ihrer abstrakten `test()`-Methode mißt und anschließend in einem synchronisierten Block die `putResults()`-Methode aufruft, um die Testergebnisse zu speichern.

[330] Um dieses Gerüst (in dem Sie das Entwurfsmuster *Templatemethod* erkennen können) zu verwenden, müssen wir für den zu testenden Containertyp eine Klasse von `Tester` und entsprechende lesende beziehungsweise schreibende Aufgabenklassen von `TestTask` ableiten:

```
//: concurrency/ListComparisons.java
// {Args: 1 10 10} (Fast verification check during build)
// Rough comparison of thread-safe List performance.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class ListTest extends Tester<List<Integer>> {
    ListTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }
    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
    class Writer extends TestTask {
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    testContainer.set(index, writeData[index]);
        }
        void putResults() {
            writeTime += duration;
        }
    }
    void startReadersAndWriters() {
        for(int i = 0; i < nReaders; i++)
            exec.execute(new Reader());
        for(int i = 0; i < nWriters; i++)
            exec.execute(new Writer());
    }
}

class SynchronizedArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
```

```
        return Collections.synchronizedList(
            new ArrayList<Integer>(new CountingIntegerList(containerSize)));
    }
    SynchronizedArrayListTest(int nReaders, int nWriters) {
        super("Synched ArrayList", nReaders, nWriters);
    }
}

class CopyOnWriteArrayListTest extends ListTest {
    List<Integer> containerInitializer() {
        return new CopyOnWriteArrayList<Integer>(
            new CountingIntegerList(containerSize));
    }
    CopyOnWriteArrayListTest(int nReaders, int nWriters) {
        super("CopyOnWriteArrayList", nReaders, nWriters);
    }
}

public class ListComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedArrayListTest(10, 0);
        new SynchronizedArrayListTest(9, 1);
        new SynchronizedArrayListTest(5, 5);
        new CopyOnWriteArrayListTest(10, 0);
        new CopyOnWriteArrayListTest(9, 1);
        new CopyOnWriteArrayListTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Sample)
      Type Read time Write time
      Synched ArrayList 10r 0w 232158294700 0
      Synched ArrayList 9r 1w 198947618203 24918613399
      readTime + writeTime = 223866231602
      Synched ArrayList 5r 5w 117367305062 132176613508
      readTime + writeTime = 249543918570
      CopyOnWriteArrayList 10r 0w 758386889 0
      CopyOnWriteArrayList 9r 1w 741305671 136145237
      readTime + writeTime = 877450908
      CopyOnWriteArrayList 5r 5w 212763075 67967464300
      readTime + writeTime = 68180227375
   *///:~
```

[331] Die konkreten inneren Klassen `Reader` und `Writer` in der abstrakten Klasse `ListTest` führen die Lese- und Schreibzugriffe an einem *`List<Integer>`*-Objekt aus. Die `Reader`-Methode `putResults()` speichert die Felder `duration` und `result`, um zu verhindern, daß die Berechnungen optimiert werden. Anschließend definiert `ListTest` die Methode `startReadersAndWriters()`, welche die `Reader`- und `Writer`-Aufgaben erzeugt und startet.

[332] Wir leiten zwei konkrete Klassen von `ListTest` ab (`SynchronizedArrayListTest` und `CopyOnWriteArrayListTest`), in denen die `containerInitializer()`-Methode definiert ist und die spezifischen Testcontainer initialisiert.

[333] Die `main()`-Methode startet verschiedene Versionen des Tests mit unterschiedlich vielen Lese- und Schreibzugriffen. Die statische `Tester`-Methode `initMain(args)` gestattet, die Testparameter über Kommandozeilenargumente selbst zu bewerten.

[334] Das Standardverhalten führt jeden Test zehnmal aus, wodurch die Ausgabe stabilisiert wird, die

durch Eigenschaften der Laufzeitumgebung wie Hotspot-Optimierung und automatische Speicherbereinigung beeinflußt werden kann (siehe Fußnote 25 auf Seite 982). Die obige Ausgabe wurde geändert und zeigt nur die letzte Iteration jedes Tests. Der Ausgabe zufolge ist die Performanz eines synchronisierten `ArrayList`-Objektes praktisch unabhängig von der Anzahl der Lese- und Schreibaufgaben (Leseaufgaben wetteifern mit anderen Leseaufgaben gleichermaßen um die Sperre wie Schreibaufgaben). Das `CopyOnWriteArrayListTest`-Objekt ist dagegen bei Abwesenheit von Schreibaufgaben dramatisch schneller, selbst noch bei fünf Schreibaufgaben. Sie können `CopyOnWriteArrayListTest`-Objekte scheinbar freizügig verwenden: Die Auswirkungen von Schreiboperationen scheinen ~~for a while~~ nicht über die Unkosten durch Synchronisierung der gesamten Liste hinauszuwachsen. Natürlich müssen Sie beide Möglichkeiten in Ihrer Anwendung ausprobieren, um festzustellen, welche die bessere ist.

[335] Beachten Sie wiederum, daß dieser Test keinen guten Vergleich in absoluten Zahlen liefert. Ihre Zahlen werden fast sicher abweichen. Das Ziel besteht darin, einen Eindruck des relativen Verhaltens der beiden Containertypen zu vermitteln.

[336] Da sich die Implementierung der Klasse `CopyOnWriteArraySet` auf `CopyOnWriteArrayList` stützt, verhält sie sich ähnlich und wir können auf einen separaten Test verzichten.

22.9.2.2 Synchronisierte HashMap und ConcurrentHashMap im Vergleich

[337] Wir verwenden dasselbe Gerüst, um einen Eindruck vom Leistungsverhältnis zwischen einem synchronisierten `HashMap`- und einem `ConcurrentHashMap`-Objekt zu bekommen:

```
//: concurrency/MapComparisons.java
// {Args: 1 10 10} (Fast verification check during build)
// Rough comparison of thread-safe Map performance.
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

abstract class MapTest extends Tester<Map<Integer,Integer>> {
    MapTest(String testId, int nReaders, int nWriters) {
        super(testId, nReaders, nWriters);
    }
    class Reader extends TestTask {
        long result = 0;
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    result += testContainer.get(index);
        }
        void putResults() {
            readResult += result;
            readTime += duration;
        }
    }
    class Writer extends TestTask {
        void test() {
            for(long i = 0; i < testCycles; i++)
                for(int index = 0; index < containerSize; index++)
                    testContainer.put(index, writeData[index]);
        }
        void putResults() {
            writeTime += duration;
        }
```

```
    }
    void startReadersAndWriters() {
        for(int i = 0; i < nReaders; i++)
            exec.execute(new Reader());
        for(int i = 0; i < nWriters; i++)
            exec.execute(new Writer());
    }
}

class SynchronizedHashMapTest extends MapTest {
    Map<Integer,Integer> containerInitializer() {
        return Collections.synchronizedMap(
            new HashMap<Integer,Integer>(
                MapData.map(
                    new CountingGenerator.Integer(),
                    new CountingGenerator.Integer(),
                    containerSize)));
    }
    SynchronizedHashMapTest(int nReaders, int nWriters) {
        super(''Synched HashMap'', nReaders, nWriters);
    }
}

class ConcurrentHashMapTest extends MapTest {
    Map<Integer,Integer> containerInitializer() {
        return new ConcurrentHashMap<Integer,Integer>(
            MapData.map(
                new CountingGenerator.Integer(),
                new CountingGenerator.Integer(), containerSize));
    }
    ConcurrentHashMapTest(int nReaders, int nWriters) {
        super(''ConcurrentHashMap'', nReaders, nWriters);
    }
}

public class MapComparisons {
    public static void main(String[] args) {
        Tester.initMain(args);
        new SynchronizedHashMapTest(10, 0);
        new SynchronizedHashMapTest(9, 1);
        new SynchronizedHashMapTest(5, 5);
        new ConcurrentHashMapTest(10, 0);
        new ConcurrentHashMapTest(9, 1);
        new ConcurrentHashMapTest(5, 5);
        Tester.exec.shutdown();
    }
} /* Output: (Sample)
      Type Read time Write time
      Synched HashMap 10r 0w 306052025049 0
      Synched HashMap 9r 1w 428319156207 47697347568
      readTime + writeTime = 476016503775
      Synched HashMap 5r 5w 243956877760 244012003202
      readTime + writeTime = 487968880962
      ConcurrentHashMap 10r 0w 23352654318 0
      ConcurrentHashMap 9r 1w 18833089400 1541853224
      readTime + writeTime = 20374942624
      ConcurrentHashMap 5r 5w 12037625732 11850489099
      readTime + writeTime = 23888114831
   *///:~
```

Die Auswirkungen zusätzlicher Schreibaufgaben fällt bei `ConcurrentHashMap` noch weniger ins Gewicht, als bei `CopyOnWriteArrayList`. Allerdings basiert die Funktionalität von `ConcurrentHashMap` auf einer anderen Technik, welche die Unkosten durch Schreibzugriffe klar erkennbar minimiert.

22.9.3 Optimistisches Sperren

[338] Obwohl atomare Objekte atomare Operationen ausführen, zum Beispiel `decrementAndGet()`, gestatten einige atomare Klassen sogenanntes „optimistisches Sperren“ (*optimistic locking*). Bei diesem „Verfahren“ verwenden Sie keine echte Sperre, wenn Sie eine Berechnung durchführen, sondern rufen nach dem Abschluß der Berechnung, wenn Sie das atomare Objekt aktualisieren wollen, dessen `compareAndSet()`-Methode auf. Die Methode erwartet den alten und den neuen Wert. Stimmen der alte Wert und der Inhalt des atomaren Objektes nicht überein, so scheitert der Methodenaufruf (in diesem Fall wurde der Inhalt des atomaren Objektes zwischenzeitlich von einer anderen Aufgabe verändert). In der Regel würden wir eine Sperre verwenden (traditionelle Synchronisierung oder ein Sperrobjekt), um zu verhindern, daß mehr als eine Aufgabe das Objekt zur selben Zeit modifizieren kann, aber wir sind „optimistisch“, daß keine andere Aufgabe die Daten ändert und verzichten auf eine Sperre. Es sei nochmals darauf hingewiesen, daß all dies der Performanz dienen soll. Die Verwendung atomarer Objekte anstelle traditioneller Synchronisierung oder expliziter Sperrobjekte kann einen Performanzgewinn bewirken.

[339] Was geschieht, wenn die Methode `compareAndSet()` scheitert? Hier beginnen die Schwierigkeiten. Sie können „optimistisches Sperren“ nur bei Problemen anwenden, die sich an die Erfordernisse dieses „Verfahrens“ anpassen lassen. Scheitert `compareAndSet()`, so müssen Sie entscheiden *können*, wie es weitergeht. Dieser Gesichtspunkt ist sehr wichtig, denn wenn Sie nichts tun können, damit sich Ihr Programm in dieser Situation wieder aufrappelt, dann eignet sich das „optimistische Sperren“ nicht und Sie müssen einen der herkömmlichen Sperrmechanismen implementieren. Eventuell können Sie `compareAndSet()` ein zweites mal aufrufen und es genügt, wenn die Operation diesesmal ausgeführt wird. Es kann auch vertretbar sein, das Scheitern einfach nicht zu beachten. Bei manchen Simulationen ergibt sich ein verloren gegangener Datenpunkt letztendlich aus dem Gesamtzusammenhang. (Sie müssen natürlich Ihr Modell gut genug verstehen, um zu wissen, ob eine solche Folgerung zutrifft.)

[340] Wir betrachten eine fiktive Simulation, die aus 100000 „Genen“ der Länge 30 besteht, vielleicht der Anfang eines genetischen Algorithmus'. Angenommen, jeder Evolutionsschritt des genetischen Algorithmus' erfordert eine sehr aufwändige Berechnung und Sie haben sich entschieden, ein Mehrprozessorsystem zu nutzen, um die Aufgaben verteilen und damit die Performanz verbessern zu können. Außerdem verwenden Sie atomare Objekte anstelle von Sperrobjekten, um die Unkosten durch das Akquirieren und Aufheben der Sperren zu vermeiden. (Selbstverständlich sind Sie zu Ihrer jetzigen Lösung gekommen, nachdem Sie zunächst den einfachsten Weg gegangen sind und traditionelle Synchronisierung per `synchronized` gewählt haben. Als das Programm lief, entdeckten Sie wie langsam es war und begannen mit Optimierung der Performanz.) Modellbedingt können Kollisionen während der Berechnung einfach ignoriert werden und es genügt, wenn die entsprechende Aufgabe die Kollision anzeigt und auf die Aktualisierung des Wertes verzichtet:

```
//: concurrency/FastSimulation.java
import java.util.concurrent.*;
import java.util.concurrent.atomic.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class FastSimulation {
    static final int N__ELEMENTS = 100000;
    static final int N__GENES = 30;
```

```
    static final int N__EVOLVERS = 50;
    static final AtomicInteger[][] GRID =
        new AtomicInteger[N__ELEMENTS][N__GENES];
    static Random rand = new Random(47);
    static class Evolver implements Runnable {
        public void run() {
            while(!Thread.interrupted()) {
                // Randomly select an element to work on:
                int element = rand.nextInt(N__ELEMENTS);
                for(int i = 0; i < N__GENES; i++) {
                    int previous = element - 1;
                    if(previous < 0) previous = N__ELEMENTS - 1;
                    int next = element + 1;
                    if(next >= N__ELEMENTS) next = 0;
                    int oldvalue = GRID[element][i].get();
                    // Perform some kind of modeling calculation:
                    int newvalue = oldvalue +
                        GRID[previous][i].get() + GRID[next][i].get();
                    newvalue /= 3; // Average the three values
                    if(!GRID[element][i]
                       .compareAndSet(oldvalue, newvalue)) {
                        // Policy here to deal with failure. Here, we
                        // just report it and ignore it; our model
                        // will eventually deal with it.
                        print("Old value changed from " + oldvalue);
                    }
                }
            }
        }
    }
    public static void main(String[] args) throws Exception {
        ExecutorService exec = Executors.newCachedThreadPool();
        for(int i = 0; i < N__ELEMENTS; i++)
            for(int j = 0; j < N__GENES; j++)
                GRID[i][j] = new AtomicInteger(rand.nextInt(1000));
        for(int i = 0; i < N__EVOLVERS; i++)
            exec.execute(new Evolver());
        TimeUnit.SECONDS.sleep(5);
        exec.shutdownNow();
    }
} /* (Execute to see output) *////:~
```

Die Elemente befinden sich sämtlich in einem Array, wodurch die Performanz verbessert werden soll (diese Annahme wird in Übungsaufgabe 39 überprüft). Jede `Evolver`-Aufgabe berechnet den Durchschnittswert eines Elementes und seiner beiden Nachbarn. Tritt bei der Aktualisierung des Elementes eine Kollision auf, so gibt die Aufgabe den Wert einfach aus und arbeitet weiter. Beachten Sie, daß das Programm keinerlei Sperrmechanismus verwendet.

Übungsaufgabe 39: (6) Sind die Annahmen im Beispiel *FastSimulation.java* vernünftig? Ersetzen Sie versuchshalber im Array `GRID` die `AtomicInteger`-Objekte durch gewöhnliche `int`-Werte und verwenden Sie Sperrobjekte. Vergleichen Sie die Performanz zwischen beiden Versionen des Programms. ■

22.9.4 Das Interface ReadWriteLock und die Klasse ReentrantReadWriteLock

[341] Das Interface *java.util.concurrent.locks.ReadWriteLock* optimiert die Situation, in der Sie relativ selten in eine Datenstruktur schreiben, aber viele Aufgaben diese Struktur abfragen. Ein Sperrobjekt vom Typ *ReadWriteLock* gestattet viele gleichzeitige Lesezugriffe, solange kein Schreibzugriff stattfindet. Bei akquirierter Schreibsperre ist kein lesender Zugriff möglich, bis die Schreibsperre wieder aufgehoben wird.

[342] Es ist völlig ungewiß, ob ein Sperrobjekt vom Typ *ReadWriteLock* die Performanz Ihres Programms positiv beeinflußt und hängt vom Verhältnis der Lese- und Schreibzugriffe, der Geschwindigkeit der einzelnen Lese- und Schreiboperationen (der Sperrmechanismus ist komplex und zahlt sich erst bei zeitaufwändigen Operationen aus), der Wettbewerbssituation unter den vorhandenen Threads und davon ab, ob das Programm auf einem Mehrprozessorsystem läuft oder nicht. Letztendlich haben Sie keine andere Wahl, als auszuprobieren, ob sich ein Sperrobjekt vom Typ *ReadWriteLock* vorteilhaft auswirkt oder nicht.

[343] Das folgende Beispiel zeigt die grundsätzliche Anwendung eines Sperrobjektes vom Typ *ReadWriteLock*:

```
//: concurrency/ReaderWriterList.java
import java.util.concurrent.*;
import java.util.concurrent.locks.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ReaderWriterList<T> {
    private ArrayList<T> lockedList;
    // Make the ordering fair:
    private ReentrantReadWriteLock lock = new ReentrantReadWriteLock(true);
    public ReaderWriterList(int size, T initialValue) {
        lockedList = new ArrayList<T>(Collections.nCopies(size, initialValue));
    }
    public T set(int index, T element) {
        Lock wlock = lock.writeLock();
        wlock.lock();
        try {
            return lockedList.set(index, element);
        } finally {
            wlock.unlock();
        }
    }
    public T get(int index) {
        Lock rlock = lock.readLock();
        rlock.lock();
        try {
            // Show that multiple readers
            // may acquire the read lock:
            if(lock.getReadLockCount() > 1)
                print(lock.getReadLockCount());
            return lockedList.get(index);
        } finally {
            rlock.unlock();
        }
    }
    public static void main(String[] args) throws Exception {
        new ReaderWriterListTest(30, 1);
    }
```

```
}

class ReaderWriterListTest {
    ExecutorService exec = Executors.newCachedThreadPool();
    private final static int SIZE = 100;
    private static Random rand = new Random(47);
    private ReaderWriterList<Integer> list =
        new ReaderWriterList<Integer>(SIZE, 0);
    private class Writer implements Runnable {
        public void run() {
            try {
                for(int i = 0; i < 20; i++) { // 2 second test
                    list.set(i, rand.nextInt());
                    TimeUnit.MILLISECONDS.sleep(100);
                }
            } catch(InterruptedException e) {
                // Acceptable way to exit
            }
            print("Writer finished, shutting down");
            exec.shutdownNow();
        }
    }
    private class Reader implements Runnable {
        public void run() {
            try {
                while(!Thread.interrupted()) {
                    for(int i = 0; i < SIZE; i++) {
                        list.get(i);
                        TimeUnit.MILLISECONDS.sleep(1);
                    }
                }
            } catch(InterruptedException e) {
                // Acceptable way to exit
            }
        }
    }
    public ReaderWriterListTest(int readers, int writers) {
        for(int i = 0; i < readers; i++)
            exec.execute(new Reader());
        for(int i = 0; i < writers; i++)
            exec.execute(new Writer());
    }
} /* (Execute to see output) *////:~
```

Ein `ReaderWriterList`-Objekt kann eine feste Anzahl von Elementen eines frei wählbaren Typs aufnehmen. Der Konstruktor erwartet die gewünschte Größe der Liste sowie ein „Anfangsobjekt“, mit dessen Kopien die Liste gefüllt wird. Die `set()`-Methode akquiriert die Schreibsperre, um die `set()`-Methode des unterliegenden `ArrayList`-Objektes aufrufen zu können, die `get()`-Methode akquiriert die Lesesperre, um die `get()`-Methode aufrufen zu können. Die `get()`-Methode der Klasse `ReaderWriterList` prüft außerdem, ob mehr als eine `Reader`-Aufgabe die Lesesperre akquiriert hat und gibt in diesem Fall deren Anzahl aus, um zu demonstrieren, daß die Lesesperre mehrfach vergeben werden kann.

[344] Zum Testen der Klasse `ReaderWriterList` erzeugt die Hilfsklasse `ReaderWriterListTest` sowohl `Reader`- als auch `Writer`-Aufgaben für ein `ReaderWriterList<Integer>`-Objekt. Beachten Sie, daß es deutlich weniger `Writer`- als `Reader`-Aufgaben gibt.

[345] Die API-Dokumentation der Klasse `ReentrantReadWriteLock` beschreibt eine Reihe weiterer Methoden sowie die Konzepte „Fairness“ und „Policy Decisions“. `ReentrantReadWriteLock` ist eine anspruchsvolle Klasse und nur interessant, wenn Sie sich nach Möglichkeiten umsehen, um die Performanz zu verbessern. Beim ersten Entwurf Ihres Programms sollten Sie einfache Synchronisierung wählen und nur bei Bedarf über Sperrobjekte vom Typ *`ReadWriteLock`* nachdenken.

Übungsaufgabe 40: (6) Schreiben Sie eine Klasse `ReaderWriterMap` mit darin enthaltendem `HashMap`-Objekt (orientieren Sie sich dabei am Beispiel *ReaderWriterList.java*). Passen Sie das Beispiel *MapComparisons.java* so an, daß Sie die Performanz Ihrer Klasse `ReaderWriterMap` untersuchen können. Wie performant ist `ReaderWriterMap` verglichen mit einem synchronisierten `HashMap`- beziehungsweise einem `ConcurrentHashMap`-Objekt? ■

22.10 Aktive Objekte

[346] Nachdem Sie dieses Kapitel durchgearbeitet haben, haben Sie vermutlich den Eindruck, daß die Threadprogrammierung in Java kompliziert und ihre korrekte Verwendung schwierig zu durchzusetzen ist. Nicht zuletzt wirkt sich die Threadprogrammierung anscheinend sogar ein wenig kontraproduktiv aus. Einerseits können Aufgaben zwar parallel verarbeitet werden, andererseits müssen Sie viel Mühe in technische Tricks investieren, um zu verhindern, daß sich diese Aufgaben nicht gegenseitig störend beeinflussen.

[347] Wenn Sie einmal in Assembler programmiert haben, dann haben Sie bei der Threadprogrammierung ein ähnliches Gefühl: Es kommt auf jede Einzelheit an, Sie sind für alles verantwortlich und es gibt kein Sicherheitsnetz in Form von Prüfungen zur Übersetzungszeit.

[348] Wohnt das Problem dem Threadmodell selbst inne? Letztenendes stammt das Konzept relativ unverändert aus der prozeduralen Programmierung. Vielleicht gibt es ein anderes Modell für parallele Programmierung, das besser zur objektorientierten Programmierung paßt.

[349] Ein alternativer Ansatz sind die sogenannten *aktiven Objekte* (*active objects*) oder *Aktoren* (*actors*).[27] Die Objekte werden als „aktiv“ bezeichnet, da jedes Objekt einen eigenen ~~worker thread~~ und eine Warteschlange für Nachrichten unterhält, wobei sämtliche Anfragen an das Objekt in der Warteschlange aneinandergereiht werden und stets höchstens eine Anfrage verarbeitet wird. Aktive Objekte *sequentialisieren also Nachrichten anstelle von Methodenaufrufen*, das heißt, wir müssen Resourcen nicht mehr dagegen schützen, daß Aufgaben während ihrer Verarbeitung suspendiert werden.

[350] Wenn Sie einem aktiven Objekt eine Nachricht senden, wird diese in eine Aufgabe umgewandelt und in die Warteschlange des Objektes gestellt, um zu einem späteren Zeitpunkt verarbeitet zu werden. Der seit der SE 5 vorhandene Typ *`java.util.concurrent.Future`* paßt in diesen Zusammenhang. Das folgende Beispiel hat zwei Methoden, die je einen Methodenaufruf in die Warteschlange stellen:

```
//: concurrency/ActiveObjectDemo.java
// Can only pass constants, immutables, ''disconnected
// objects,'' or other active objects as arguments
// to asynch methods.
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.Print.*;

public class ActiveObjectDemo {
```

[27] Danke an Allen Holub, der sich die Zeit genommen hat, mit dieses Modell zu erklären.

```
    private ExecutorService ex = Executors.newSingleThreadExecutor();
    private Random rand = new Random(47);
    // Insert a random delay to produce the effect
    // of a calculation time:
    private void pause(int factor) {
        try {
            TimeUnit.MILLISECONDS.sleep(100 + rand.nextInt(factor));
        } catch(InterruptedException e) {
            print("sleep() interrupted");
        }
    }
    public Future<Integer> calculateInt(final int x, final int y) {
        return ex.submit(new Callable<Integer>() {
            public Integer call() {
                print("starting " + x + " + " + y);
                pause(500);
                return x + y;
            }
        });
    }
    public Future<Float> calculateFloat(final float x, final float y) {
        return ex.submit(new Callable<Float>() {
            public Float call() {
                print("starting " + x + " + " + y);
                pause(2000);
                return x + y;
            }
        });
    }
    public void shutdown() { ex.shutdown(); }
    public static void main(String[] args) {
        ActiveObjectDemo d1 = new ActiveObjectDemo();
        // Prevents ConcurrentModificationException:
        List<Future<?>> results = new CopyOnWriteArrayList<Future<?>>();
        for(float f = 0.0f; f < 1.0f; f += 0.2f)
            results.add(d1.calculateFloat(f, f));
        for(int i = 0; i < 5; i++)
            results.add(d1.calculateInt(i, i));
        print("All asynch calls made");
        while(results.size() > 0) {
            for(Future<?> f : results)
                if(f.isDone()) {
                    try {
                        print(f.get());
                    } catch(Exception e) {
                        throw new RuntimeException(e);
                    }
                    results.remove(f);
                }
        }
        d1.shutdown();
    }
} /* Output: (85% match)
      All asynch calls made
      starting 0.0 + 0.0
      starting 0.2 + 0.2
      0.0
```

```
    starting 0.4 + 0.4
    0.4
    starting 0.6 + 0.6
    0.8
    starting 0.8 + 0.8
    1.2
    starting 0 + 0
    1.6
    starting 1 + 1
    0
    starting 2 + 2
    2
    starting 3 + 3
    4
    starting 4 + 4
    6
    8
*///:~
```

[351] Der durch die statische `Executors`-Methode `newSingleThreadExecutor()` erzeugte „single Threadexekutor“ pflegt seine eigene unbeschränkte blockierende Warteschlange und verfügt über nur einen einzigen Thread, um Aufgaben aus seiner Warteschlange zu entnehmen und vollständig zu verarbeiten. Es genügt, in den Methoden `calculateInt()` und `calculateFloat()` die `submit()`-Methode mit einem *`Callable`*-Objekt aufzurufen, den Aufruf der `calculateXXX()`-Methode also in eine Nachricht umzuwandeln. Der Methodenkörper befindet sich in der `call()`-Methode in der anonymen inneren Klasse. Beachten Sie, daß jede Methode des aktiven Objektes ein *`Future<T>`*-Objekt mit generischem Typparameter `T` zurückgibt, welcher der eigentliche Rückgabetyp der `call()`-Methode ist. Dadurch kehrt der Methodenaufruf fast unmittelbar zurück und der Aufrufer kann anhand des *`Future`*-Objektes feststellen, wann die Verarbeitung der Aufgabe beendet ist und den tatsächlichen Rückgabewert abfragen. Dies ist der kompliziertesten Fall. Der Prozeß wird vereinfacht, wenn die aufgerufene Methode keinen Rückgabewert hat.

[352] In der `main()`-Methode wird ein *`List<Future<?>>`*-Objekt erzeugt, um die von den Nachrichten an das aktive Objekt (`calculateInt()` und `calculateFloat()`) zurückgegebenen *`Future`*-Objekte zu speichern. In der `for`-Schleife wird der Status jedes *`Future`*-Objektes mittels `isDone()` abgefragt und verarbeitete Objekte werden aus der Liste entfernt. Beachten Sie, daß die Notwendigkeit, mit einer Kopie der Liste zu arbeiten, um Ausnahmen vom Typ `java.util.ConcurrentModificationException` zu vermeiden, durch die Wahl der Klasse `java.util.concurrent.CopyOnWriteArrayList` nicht mehr besteht.

[353] Um unbeabsichtigte Kopplungen zwischen Threads zu verhindern, müssen die Argumente einer Methode eines aktiven Objektes entweder nur-lesbar, wiederum aktive Objekte oder nicht verknüpfte Objekte (*disconnected objects*) sein, das heißt Objekte, die keine Verbindung mit einer anderen Aufgabe haben (schwierig zu bewerkstelligen, da die Sprache keine Unterstützung bietet).

[354] Eigenschaften und Fähigkeiten aktiver Objekte:

- Jedes aktive Objekt unterhält seinen eigenen ~~worker thread~~.
- Jedes aktive Objekt hat die uneingeschränkte Kontrolle über seine Felder (eine etwas strengere Eigenschaft als bei gewöhnlichen Klassen, die nur die *Möglichkeit* haben, ihre Felder zu schützen).
- Sämtliche Kommunikation zwischen aktiven Objekten wird über zwischen diesen ausgetauschte Nachrichten abgewickelt.

- Sämtliche zwischen aktiven Objekten ausgetauschten Nachrichten werden in einer Warteschlange aufgereiht.

[355] Die Ergebnisse sind unwiderstehlich. Da eine Nachricht von einem aktiven Objekt an ein anderes lediglich durch die kurze Verzögerung beim Eintrag in die Warteschlange blockiert werden kann und diese Verzögerung stets sehr gering und von anderen Objekten unabhängig ist, ist das Senden einer Nachricht effektiv nicht blockierbar (im schlimmsten Fall kommt es zu einer geringfügigen Verzögerung). Da ein System aktiver Objekte ausschließlich über Nachrichtenaustausch kommuniziert, können zwei Objekte beim Wettstreit um das Aufrufen einer Methode eines aktiven Objektes nicht blockiert werden, so daß Verklemmungen ausgeschlossen sind: ein großer Schritt vorwärts. Da der worker thread eines aktiven Objektes stets höchstens eine Nachricht verarbeitet, gibt es keinen Wettstreit um Resourcen und Sie müssen sich den Kopf nicht mit dem Synchronisieren von Methoden zerbrechen. Synchronisierung findet noch immer statt; allerdings auf der Nachrichtenebene, durch Anordnung der Methodenaufrufe in der Warteschlange, so daß stets höchstens ein Methodenaufruf verarbeitet werden kann.

[356] Aufgrund der fehlenden direkten Unterstützung durch den Compiler ist der obige Ansatz leider recht umständlich. Allerdings sind auf dem Gebiet der aktiven Objekte und Aktoren Fortschritte zu verzeichnen; interessanter noch, auch in der sogenannten agentenbasierten (*agent-based*) Programmierung. Agenten sind effektiv aktive Objekte, aber Agentensysteme gestatten auch Transparenz über Netzwerke und Maschinen hinweg. Ich wäre nicht überrascht, wenn sich die agentenbasierte Programmierung als eventueller Nachfolger der objektorientierten Programmierung herausstellt, da dieser Ansatz Objekte mit einem vergleichsweise einfachen Ansatz für die parallele Programmierung vereint.

[357] Sie finden im Internet weiterführende Informationen über aktive Objekte, Aktoren und Agenten. Einige Ideen hinter dem Konzept des aktiven Objektes stammen von Charles Antony Richard Hoares *Theory of Communicating Sequential Processes (CSP)*.

Übungsaufgabe 41: (6) Legen Sie im Beispiel *ActiveObjectDemo.java* einen Nachrichtenbehandler ohne Rückgabewert an und rufen Sie in in der `main()`-Methode auf. ■

Übungsaufgabe 42: (7) Ändern Sie das Beispiel *WaxOMatic.java* so, daß es aktive Objekte verwendet. ■

Projekt[28]**:** Entwicklen Sie mit Hilfe von Annotationen und Javassist eine Klassenannotation `@Active`, welche die ausgezeichnete Klasse in ein aktives Objekt transformiert. ■

22.11 Zusammenfassung

[358] Das Ziel dieses Kapitels bestand darin, Ihnen die Grundzüge der Threadprogrammierung mit Java zu vermitteln. Sie verstehen nun:

- Sie können viele voneinander unabhängige Aufgaben parallel verarbeiten.
- Sie müssen beim Beenden dieser Aufgaben alle möglichen Probleme in Betracht ziehen.
- Aufgaben können sich über gemeinsam verwendete Resourcen gegenseitig störend beeinflussen. Mutex-Objekte beziehungsweise Sperrobjekte sind die grundlegende Vorgehensweise, um solche Kollisionen zu verhindern.

[28] Projekte sind Vorschläge für Semester- oder Halbjahresarbeiten. Der *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können, enthält *keine* Lösungsvorschläge für Projekte.

- Ohne gewissenhaften Entwurf können sich Aufgaben gegenseitig verklemmen.

[359] Es ist wesentlich, zu lernen, wo Threadprogrammierung angebracht beziehungsweise nicht angebracht ist. Die wichtigsten Argumente *für* Threadprogrammierung sind:

- Das Vorhandensein mehrerer Aufgaben gestattet den Rechner effizienter zu nutzen (insbesondere, die Aufgaben transparent auf mehrere Prozessoren zu verteilen).
- Bessere Organisation des Quelltextes.
- Bessere Bedienbarkeit einer Anwendung durch die Benutzer.

Die Nutzung des Prozessors während der Wartezeit auf eine Ein-/Ausgabeoperation ist ein klassisches Beispiel für bessere Resourcenauslastung. Die verbesserte Organisation des Quelltextes tritt typischerweise bei Simulationen auf. Das klassische Beispiel für die verbesserte Bedienbarkeit ist die Schaltfläche „Cancel“ beim Herunterladen großer Dateien oder Datenvolumina.

[360] Ein weiterer Vorteil von Threads ist, daß die Umschaltung zwischen „leichten Ausführungskontexten“ (Größenordnung: 100 Anweisungen), im Gegensatz zu den „schwergewichtigen Prozeßkontexten“ (Größenordnung: 1000 Anweisungen). Da sich sämtliche Threads innerhalb eines Prozesses denselben Speicherbereich teilen, werden beim Umschalten zwischen zwei leichten Kontexten nur die Programmausführung und die lokalen Variablen ausgetauscht. Beim Umschalten zwischen zwei schwergewichtigen Prozessen muß dagegen der gesamte Speicherbereich ausgewechselt werden.

[361] Die wichtigsten Argumente *gegen* Threadprogrammierung sind:

- ~~Slowdown occurs~~, während Threads auf gemeinsam verwendete Resourcen warten.
- Die Verwaltung von Threads verursacht zusätzliche Unkosten (verbraucht Prozessorzyklen).
- Undurchdachte Entwürfe bringen unbelohnte Komplexität mit sich.
- Es besteht Gelegenheit für Symptome wie das Verhungern von Threads, Wettlaufsituationen (*race conditions*), Verklemmungen (*deadlocks*) und Livelocks (*livelocks*), wobei mehrere Threads individuelle Aufgaben verarbeiten, das Programm aber nicht beendet werden kann.
- Inkonsistentes Verhalten unter verschiedenen Plattformen. Ich habe beispielsweise bei der Arbeit an einigen Beispielen für dieses Buch festgestellt, daß sich auf einem meiner Rechner schnell Wettlaufsituationen ergaben, die sich auf einem anderen Rechner aber nicht zeigten. Wenn Sie auf dem letzteren Rechner ein Programm entwickeln, erleben Sie bei der späteren Verteilung vielleicht eine böse Überraschung.

[362] Eine der größten Schwierigkeiten in der Threadprogrammierung besteht darin, daß mehr als eine Aufgabe Zugriff auf eine Resource (zum Beispiel ein Objektfeld) haben kann und Sie dafür sorgen müssen, daß nicht mehrere Threads zugleich versuchen, lesend oder schreibend auf die Resource zuzugreifen. Die Lösung dieses Problems setzt verständigen Umgang mit den vorhandenen Sperrmechanismen voraus, wie beispielsweise der Synchronisierung. Sie sind einerseits grundsätzliche Werkzeuge, erfordern aber andererseits gewissenhaftes Verständnis, da sie stillschweigend Verklemmungen herbeiführen können.

[363] ~~An addition, there's an art to the application of threads.~~ Das Design von Java gestattet Ihnen, zumindest theoretisch, soviele Objekte zu erzeugen, als Sie zur Lösung Ihres Problems benötigen. (Das Erzeugen von Millionen von Objekten zur Lösung eine Ingenieuraufgabe mit Hilfe Finiter Elemente, ist zum Beispiel in Java nicht praktikabel, sofern Sie nicht das *Flyweight*-Entwurfsmuster zugrundelegen.) Andererseits gibt es eine Obergrenze für die Anzahl erzeugter Threads, da Threads ab einer gewissen Anzahl störrisch werden. Dieser kritische Punkt ist unter Umständen schwierig auszumachen und hängt häufig vom Betriebssysten und der Laufzeitumgebung ab; es können einige

hundert oder auch einige tausend Threads sein. Da Sie in der Regel aber nur mit einer Handvoll Threads arbeiten, um ein Problem zu lösen, ist dies typischerweise keine echte Beschränkung. Bei einem allgemeineren Entwurf wird die Höchstzahl von Threads aber zu einer Randbedingung, die Sie zwingen kann, auf kooperative Threadunterstützung auszuweichen.

[364] Threadprogrammierung ist eine schwarze Kunst, gleichgültig wie einfach sie sich im Gewand einer bestimmten Programmiersprache oder Bibliothek darstellt. Es gibt immer etwas, das Sie beißt, wenn Sie es am wenigstens erwarten. Das Interessante am Problem der speisenden Philosophen ist nämlich, daß es sich so einstellen läßt, daß Verklemmungen nur selten auftreten, wodurch Sie den Eindruck haben, daß alles in Ordnung ist.

[365] Verwenden Sie Threads generell sorgfältig und sparsam. Wenn das Problem, das Sie per Threadprogrammierung lösen möchten, wächst und kompliziert wird, ziehen Sie in Betracht, auf eine Sprache wie Erlang auszuweichen. Wenn Sie öfter auf derartige Schwierigkeiten stoßen und die Probleme kompliziert genug sind, ist das Ausweichen auf eine andere Programmiersprache für einen Teil Ihres Programms eine legitime Vorgehensweise.

22.11.1 Literaturempfehlungen

[366] Leider kursieren viele irreführende Informationen über das Thema Threadprogrammierung. Dies unterstreicht, wie unübersichtlich das Gebiet ist und wie leicht man glaubt, es zu verstehen. (Ich spreche aus eigener Erfahrung. Ich hatte in der Vergangenheit schon viele Male den Eindruck, die Threadprogrammierung verstanden zu haben und zweifle nicht daran, daß mir noch die eine oder andere Erleuchtung bevorsteht.) Jedesmal wenn Sie einen neuen Text über Threadprogrammierung in die Hand nehmen, ist einige Detektivarbeit erforderlich, um zu versuchen zu verstehen, was der Autor verstanden hat und was nicht. Ich bin mir dagegen sicher, daß die folgenden drei Bücher vertrauenswürdig sind.

[367] *Java Concurrency in Practice* von Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes und Doug Lea, erschienen bei Addison-Wesley (2006). Im Grunde genommen die Hautevolee auf dem Gebiet der Threadprogrammierung unter Java.

[368] *Concurrent Programming in Java*, 2nd Edition, von Doug Lea, erschienen bei Addison-Wesley (2000). Obwohl dieses Buch deutlich vor der SE 5 liegt, schlägt sich ein großer Teil der Arbeit des Autors in den Bibliotheken im Package `java.util.concurrent` nieder. Das Buch ist wesentlich für ein umfassendes Verständnis der Threadprogrammierung. Es geht über die Threadprogrammierung unter Java hinaus und diskutiert aktuelle Sicht- und Denkweisen über Sprachen und Technologien hinweg. Obwohl hie und da abgestumpft, verdient das Buch mehrfaches Lesen (vorzugsweise mit jeweils einigen Monaten Abstand, um das Gelernte zu verinnerlichen). Doug Lea gehört zu den wenigen Menschen weltweit, die Threadprogrammierung tatsächlich verstehen, das heißt die Mühe lohnt sich.

[369] *The Java Language Specification*, 3rd Edition, (Kapitel 17), von James Gosling, Bill Joy, Guy Steele und Gilad Bracha, erschienen bei Addison-Wesley (2005). Die technische Spezifikation ist auch in elektronischer Form verfügt unter der Adresse: *http://java.sun.com/docs/books/jls*.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Kapitel 23

Graphische Benutzerschnittstellen

Inhaltsübersicht

[0] Eine fundamentale Richtlinie beim Design einer Bibliothek lautet: „Einfaches bleibt einfach, kompliziertes wird möglich.“[1]

[1] Das ursprüngliche Ziel beim Design der Bibliothek für graphische Benutzeroberflächen (*graphical user interface*, GUI) in Java 1.0 bestand darin, dem Entwickler das Programmieren einer GUI zu ermöglichen, die auf allen Plattformen ästhetisch aussieht. Dieses Ziel wurde nicht erreicht. Das Abstract Windowing Toolkit (AWT) von Java 1.0 taugte lediglich für GUIs, die auf allen Plattformen gleich mittelmäßig wirkten. Das AWT war darüber hinaus restriktiv: Sie konnten nur vier Zeichensätze verwenden und hatten keine Möglichkeit, die anspruchsvolleren GUI-Elemente Ihres Betriebsssystems zu nutzen. Das AWT-Modell unter Java 1.0 war ungelenk und nicht objektorientiert. Ein Teilnehmer in einer meiner Schulungen (der während der Entwicklung von Java bei Sun Microsystems arbeitete) erläuterte mir warum: Das ursprüngliche AWT wurde innerhalb eines Monats erdacht, entworfen und implementiert. Zweifellos ein Wunder an Produktivität, aber auch ein Schulbeispiel für die Wichtigkeit des Entwurfs.

[2] Die Situation verbesserte sich mit Erscheinen des AWT-Ereignismodells unter Java 1.1, welches einen viel klareren, objektorientierten Entwurf hatte sowie den sogenannten JavaBeans, einem Modell für die Programmierung von Komponenten, mit dem Ziel die Entwicklung visueller Programmierumgebungen zu erleichtern. Java 2 (Version 1.2 des Java Development Kits) beendete den Transformationsprozeß vom alten AWT unter Java 1.0 im wesentlichen dadurch, daß alles durch die sogenannten *Java Foundation Classes* (JFC) ersetzt wurde, deren GUI-Komponenten unter der Bezeichnung „Swing“ zusammengefaßt wurden. Die Swing-Bibliothek besteht aus einer Vielzahl leicht benutzbarer und einfach verständlicher JavaBeans, aus denen sich sowohl per Drag-and-Drop als auch mittels manueller Programmierung annehmbare GUIs konstruieren lassen. Die „Drei-Versionen-Regel“ der Softwareindustrie (kein Produkt ist gut, bevor es Version 3 erreicht), scheint auch für die Entwicklung von Programmiersprachen zu gelten.

[3] Dieses Kapitel stellt die moderne Swing-Bibliothek von Java vor und basiert auf der vernünftigen Annahme, daß sich Sun Microsystems mit Swing endgültig für eine Richtung in der GUI-Bibliothek von Java entschieden hat.[2] Falls Sie die ursprüngliche „alte“ AWT-Version brauchen, zum Beispiel weil Sie einen älteren Quelltext pflegen müssen oder hinsichtlich des Browsers Einschränkungen bestehen, finden Sie in der ersten Auflage dieses Buches eine Einführung (siehe *http://www.mindview.net*). Beachten Sie, daß einige AWT-Komponenten bis in die heutige Java-Version überdauert haben und in manchen Situationen verwendet werden müssen.

[4] Denken Sie bitte daran, daß dieses Kapitel weder eine umfassende Dokumentation aller Swing-Komponenten, noch aller Methoden der in diesem Kapitel auftretenden Klassen ist. Die Swing-Bibliothek ist gewaltig und das Ziel dieses Kapitels besteht nur darin, Sie hinsichtlich der Grundlagen und Konzepte auf den richtigen Weg zu bringen. Wenn Sie mehr benötigen, als Sie in diesem Kapitel vorfinden, stehen die Chancen gut, daß Swing Ihnen zur Verfügung stellt, was Sie brauchen, wenn Sie bereit sind, die erforderliche Recherche auf sich zu nehmen.

[5] Ich setze voraus, daß Sie die Dokumentation des Java Development Kits von *http://java.sun.com* heruntergeladen und installiert haben und die Einzelheiten und Methoden in der API-Dokumentation der Klassen im Package `javax.swing` nachschlagen. Sie können natürlich auch im Internet suchen, aber der beste Einstiegspunkt ist das Swing-Tutorial von Sun Microsystems unter der Webadresse *http://java.sun.com/docs/books/tutorial/uiswing*.

[1] Eine Variation des „Prinzips des geringen Erstaunens“: „Überraschen Sie den Benutzer nicht.“

[2] Beachten Sie die von IBM ins Lebens gerufene neue quelloffene GUI-Bibliothek für die Entwicklungsumgebung „Eclipse“ (*www.eclipse.org*), die Sie als Alternative zu Swing in Betracht ziehen können. Diese Bibliothek wird in Abschnitt 23.14 behandelt.

[6] Es gibt zahlreiche (ziemlich dicke) Bücher, die sich ausschließlich Swing widmen. Greifen Sie zu diesen Büchern, wenn Sie tieferes Wissen benötigen oder das Standardverhalten von Swing ändern möchten.

[7] Während Sie sich mit Swing auseinandersetzen, werden Sie entdecken:

- Verglichen mit vielen anderen Programmiersprachen und Entwicklungsumgebungen hat Swing ein deutlich besseres Programmiermodell (was nicht heißen soll, daß Swing perfekt ist, wohl aber ein Schritt vorwärts).
- Sogenannte „GUI-Builder“ (visuelle Entwicklungsumgebungen) sind ein unerläßlicher Bestandteil einer vollständigen Java-Entwicklungsumgebung. JavaBeans und Swing ermöglichen dem GUI-Builder, den Quelltext zu generieren, während Sie mit Hilfe eines graphischen Werkzeuges GUI-Komponenten auf einem Formular positionieren. Die GUI-Entwicklung wird erheblich beschleunigt, wodurch mehr Zeit zum Experimentieren übrig bleibt, so daß mehr Entwürfe probiert und eventuell ein besserer gefunden werden kann.
- Da Swing relativ unkompliziert ist, auch wenn Sie einen GUI-Builder verwenden, statt die Anweisungen manuell zu entwickeln, ist der resultierende Quelltext nachvollziehbar. Damit ist ein großes Problem mit GUI-Buildern aus der Vergangenheit gelöst, die mühelos unleserlichen Quelltext generieren konnten.

[8] Die Swing-Bibliothek enthält sämtliche Komponenten, die Sie in einer modernen Benutzerschnittstelle erwarten: von Schaltflächen mit kleinen Graphiken bis hin zu GUI-Komponenten für baum- oder tabellenartige Datenstrukturen. Die Swing-Bibliothek ist zwar groß, aber die Komplexität paßt sich an die Aufgabe an, das heißt bei einfachen Dingen sind nur wenige Anweisungen erforderlich, bei komplizierteren Aufgaben wächst die Komplexität proportional mit.

[9] Viele der Swing-Eigenschaften und -Fähigkeiten, die Ihnen gefallen werden, lassen sich unter dem Konzept orthogonality of use zusammenfassen: Wenn Sie die Grundideen der Swing-Bibliothek einmal verinnerlicht haben, können Sie sie in der Regel überall anwenden. Das liegt hauptsächlich an den standardisierten Bezeichnern. Bei der Arbeit an diesem Kapitel konnte ich die Methodennamen im allgemeinen mit Erfolg vorausahnen. Dies ist sicherlich ein Qualitätssiegel für gutes Bibliotheksdesign. Außerdem können Sie in der Regel Komponenten in andere Komponenten einsetzen und es funktioniert.

[10–11] Die Navigation durch eine GUI per Tastatur ist automatisch vorhanden: Sie können eine Swing-Anwendung ohne zusätzlichen Programmieraufwand ohne Maus steuern. Rollbalken werden mühelos unterstützt: Sie verpacken Ihre GUI-Komponente einfach in einer `JScrollPane`-Komponente, bevor Sie sie auf ihrem Formular positionieren. Eigenschaften oder Fähigkeiten wie Tooltips verlangen typischerweise nur eine einzige Zeile. Aus Portabilitätsgründen ist die Swing-Bibliothek komplett in Java geschrieben.

[12] Swing unterstützt außerdem eine grundlegende Fähigkeit, nämlich sogenanntes austauschbares (*pluggable*) Look-and-Feel, wodurch sich die Erscheinungsform der Benutzerschnittstelle dynamisch an die Erwartungen der Benutzer verschiedener Plattformen und Betriebssysteme anpassen läßt. Es ist sogar möglich (wenn auch kompliziert), Ihr eigenes Look-and-Feel zu entwickeln. Sie finden eine Reihe von Look-and-Feels im Internet.[3]

[13] Trotz aller positiven Aspekte ist Swing weder für jeden geeignet, noch hat die Bibliothek alle von den Designern betrachteten Probleme bei Benutzerschnittstellen gelöst. Gegen Ende dieses Kapitels werfen wir einen Blick auf zwei Alternativen zu Swing, nämlich das von IBM zur Verfügung gestellte

[3] Mein Lieblingsbeispiel ist das von Ken Arnold entwickelte „Napkin“ („Serviette“) Look-and-Feel, welches die Fenster so zeichnet, als seien sie schnell auf eine Papierserviette hingeworfen worden. Sie finden das „Napkin“ Look-and-Feel unter der Webadresse *http://napkinlaf.sourceforge.net*.

Standard Widget Toolkit (SWT), eine zur Entwicklung der Entwicklungsumgebung Eclipse geschriebene nun aber quelloffene eigenständige GUI-Bibliothek sowie das von Macromedia zur Verfügung gestellt Werkzeug Flex, zur Entwicklung clientseitiger Flash-Oberflächen für Webapplikationen.

23.1 Applets

[14] Nachdem Java veröffentlicht wurde, drängte sich der Aufruhr um diese neue Programmiersprache hauptsächlich um das *Applet*, ein über das Internet verteilbares Programm, welches innerhalb eines sogenannten Sandkastens (*sand box*) im Webbrowser ausgeführt werden konnte. Die Fachwelt sah das Applet als nächsten Schritt in der Entwicklung des Internets voraus und die Autoren vieler damaliger Java-Bücher gingen davon aus, daß ihre Leser an der Programmiersprache nur deshalb interessiert seien, um Applets schreiben zu können.

[15] Es kam jedoch aus verschiedenen Gründen nie zu dieser Revolution. Das Problem bestand zu einem großen Teil darin, daß die meisten Rechner nicht mit der erforderlichen Java-Software ausgestattet waren, nur um Applets ausführen zu können und die meisten Benutzer nicht bereit waren, ein Paket von 10MB herunterzuladen, um ein Programm ausführen zu können, das sie nebenbei im Internet aufgestöbert hatten. Der alleinige Gedanke schreckte viele Benutzer bereits ab. Java-Applets haben unter den Verfahren zur clientseitigen Verteilung von Anwendungen nie eine kritische Masse erreicht und liegen mittlerweile, obwohl Sie gelegentlich noch auf Applets stoßen können, auf dem Schuttabladeplatz der elektronischen Datenverarbeitung.

[16] Dies bedeutet aber keinesfalls, daß Applets keine interessante oder lohnende Technologie sind. Wenn Sie davon ausgehen können, daß die Benutzer eine Java-Laufzeitumgebung installiert haben (beispielsweise im Netzwerk eines Unternehmens), so können sich Applets (beziehungsweise das Java Network Launching Protocol (JNLP) und Java Web Start; siehe Abschnitt 23.9) durchaus anbieten, um ohne den üblichen Aufwand Programme zu verteilen und die auf den Einzelplatzrechnern installierte Software zu aktualisieren.

[17] Sie finden in den Online-Anhängen zu diesem Buch unter der Webadresse *http://www.mindview.net* eine Einführung in die Applet-Technologie.

23.2 Swing-Grundlagen

[18–20] Die meisten Swing-Anwendungen basieren auf einem Objekt der Klasse `JFrame`, das ein Fenster mit den betriebssystemabhängigen Eigenschaften und Fähigkeiten erzeugt. Die Titelzeile des Fensters kann über den Konstruktor der Klasse `JFrame` bewertet werden:

```
//: gui/HelloSwing.java
import javax.swing.*;

public class HelloSwing {
    public static void main(String[] args) {
        JFrame frame = new JFrame("Hello Swing");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
    }
} ///:~
```

Die Methode `setDefaultCloseOperation()` teilt dem `JFrame`-Objekt mit, was zu tun ist, wenn der Benutzer das Fenster beendet. Die Konstante `EXIT_ON_CLOSE` weist das `JFrame`-Objekt an, das

Programm zu beenden. Das Standardverhalten, ohne Aufruf der `setDefaultCloseOperation()`-Methode, ist nichts zu tun, das heißt die Anwendung würde nicht beendet werden. Die Methode `setSize()` definiert die Abmessungen des Fensters in Pixeln. Beachten Sie die letzte Zeile:

```
frame.setVisible(true);
```

Ohne diese Anweisung würden Sie keine Ausgabe erhalten.

[21] Im folgenden Beispiel legen wir im Darstellungsbereich des `JFrame`-Objektes ein `JLabel`-Objekt (eine Beschriftung) an:

```
//: gui/HelloLabel.java
import javax.swing.*;
import java.util.concurrent.*;

public class HelloLabel {
    public static void main(String[] args) throws Exception {
        JFrame frame = new JFrame("Hello Swing");
        JLabel label = new JLabel("A Label");
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
        TimeUnit.SECONDS.sleep(1);
        label.setText("Hey! This is Different!");
    }
} ///:~
```

Nach einer Sekunde ändert sich der Beschriftungstext. In einem so trivialen Programm ist es nett und sicher, eine Eigenschaft einer GUI-Komponente direkt über den `main`-Thread zu ändern. Swing verfügt über einen eigenen Thread, um Ereignisse in der Benutzerschnittstelle abzufangen und die Bildschirmausgabe zu aktualisieren. Wenn Sie den Bildschirminhalt mit eigenen Threads manipulieren, besteht die Gefahr der in Kapitel 22 beschriebenen Kollisionen und Verklemmungen.

[22] Threads wie `main` sollten Aufgaben stattdessen zur Verarbeitung an den Ereignisbehandlerthread[4] (*event dispatcher thread*) von Swing übergeben. Die Übergabe einer Aufgabe an den Ereignisbehandlerthread geschieht durch die statische `SwingUtilities`-Methode `invokeLater()`, welche die Aufgabe in die Ereigniswarteschlange einreiht, aus der sie (schließlich) vom Ereignisbehandlerthread entnommen und verarbeitet wird. Das vorige Beispiel lautet entsprechend umgeschrieben:

```
//: gui/SubmitLabelManipulationTask.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitLabelManipulationTask {
    public static void main(String[] args) throws Exception {
        JFrame frame = new JFrame("Hello Swing");
        final JLabel label = new JLabel("A Label");
        frame.add(label);
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setSize(300, 100);
        frame.setVisible(true);
        TimeUnit.SECONDS.sleep(1);
        SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    label.setText("Hey! This is Different!");
```

[4] Der Ereignisbehandlerthread gehört technisch zur AWT-Bibliothek.

```
                }
            });
        }
    } ///:~
```

[23] Nun wird das `JLabel`-Objekt nicht mehr direkt geändert. Sie übergeben statt dessen ein Objekt vom Typ *`Runnable`* (eine Aufgabe) und der Ereignisbehandlerthread führt die tatsächliche Änderung aus, wenn er die zugehörige Aufgabe aus der Ereigniswarteschlange entnimmt. Der Ereignisbehandlerthread verarbeitet das *`Runnable`*-Objekt separat, das heißt ohne zugleich weitere Aufgaben zu verarbeiten, so daß Kollisionen ausgeschlossen sind (vorausgesetzt, daß Ihr gesamtes Programm konsistent den Ansatz implementiert, daß Änderungen über die statische `SwingUtilities`-Methode `invokeLater()` in Auftrag gegeben werden). Dies schließt auch den Programmstart ein, das heißt die `main()`-Methode sollte keine Swing-Methoden aufrufen, wie im obigen Programm, sondern eine Aufgabe an die Warteschlange des Ereignisbehandlerthreads übergeben.[5] Das Beispiel liest sich in korrigierter Form etwa so:

```
//: gui/SubmitSwingProgram.java
import javax.swing.*;
import java.util.concurrent.*;

public class SubmitSwingProgram extends JFrame {
    JLabel label;
    public SubmitSwingProgram() {
        super("Hello Swing");
        label = new JLabel("A Label");
        add(label);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setSize(300, 100);
        setVisible(true);
    }
    static SubmitSwingProgram ssp;
    public static void main(String[] args) throws Exception {
        SwingUtilities.invokeLater(new Runnable() {
                public void run() { ssp = new SubmitSwingProgram(); }
            });
        TimeUnit.SECONDS.sleep(1);
        SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    ssp.label.setText("Hey! This is Different!");
                }
            });
    }
} ///:~
```

Beachten Sie, daß die `sleep()`-Methode nicht im Konstruktor aufgerufen wird. Wenn Sie die `sleep()`-Methode in den Konstruktor einsetzen, erscheint der ursprüngliche Beschriftungstext nicht auf dem Bildschirm, da der Konstruktor nach `sleep()` nicht mehr zuende verarbeitet wird und der neue Text eingesetzt wird, um nur einen Grund zu nennen. Das Aufrufen der `sleep()`-Methode im Konstruktor oder in einer Operation im Zusammenhang mit der Benutzerschnittstelle bedeutet, daß Sie den Ereignisbehandlerthread anhalten, solange die `sleep()`-Methode verarbeitet wird.

Übungsaufgabe 1: (1) Ändern Sie das Beispiel *HelloSwing.java*. Vergewissern Sie sich, daß die Anwendung ohne das Aufrufen der Methode `setDefaultCloseOperation()` nicht beendet wird. ■

[5] Diese Vorgehensweise existiert seit Version 5 der Java Standard Edition (SE 5) und ist daher bei älteren Programmen häufig nicht vorhanden. Es liegt nicht daran, daß die Programmierer ignorant sind. Das vorgeschlagene neue Verfahren scheint sich konstant zu verbreiten.

Übungsaufgabe 2: (2) Ändern Sie das Beispiel *HelloLabel.java*. Zeigen Sie, daß Beschriftungen dem Darstellungsbereich dynamisch hinzugefügt werden, in dem Sie eine zufällig bestimmte Anzahl von Beschriftungen anlegen. ■

23.2.1 Die Hilfsklasse SwingConsole

[24] Wir fassen nun die obigen Ideen zusammen und entfernen redundante Anweisungen, indem wir eine Hilfsklasse für die Swing-Beispiele in diesem Kapitel anlegen:

```
//: net/mindview/util/SwingConsole.java
// Tool for running Swing demos from the
// console, both applets and JFrames.
package net.mindview.util;
import javax.swing.*;

public class SwingConsole {
    public static void
        run(final JFrame f, final int width, final int height) {
        SwingUtilities.invokeLater(new Runnable() {
                public void run() {
                    f.setTitle(f.getClass().getSimpleName());
                    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    f.setSize(width, height);
                    f.setVisible(true);
                }
            });
    }
} ///:~
```

Eventuell möchten Sie diese Hilfklasse auch selbst verwenden. Ich habe sie daher in der Bibliothek im Package `net.mindview.util` untergebracht. Damit Sie die Hilfsklasse gebrauchen können, muß sich Ihre Anwendung in einem `JFrame`-Objekt befinden (trifft für alle Beispiele in diesem Kapitel zu). Die statische `run()`-Methode konfiguriert die Titelzeile des Fensters mit dem einfachen Namen Ihres `JFrame`-Objektes.

Übungsaufgabe 3: (3) Ändern Sie das Beispiel *SubmitSwingProgram.java*, so daß es die Hilfsklasse `SwingConsole` verwendet. ■

23.3 Anlegen einer Schaltfläche

[25–26] Das Anlegen einer Schaltfläche ist ein Kinderspiel: Sie rufen lediglich den Konstruktor der Klasse `JButton` auf, wobei Sie die Beschriftung der Schaltfläche übergeben. Sie lernen später in diesem Kapitel noch schickere Tricks, etwa wie Sie eine Miniaturgraphik auf der Schaltfläche anbringen können. In der Regel legen Sie ein Feld an, welches die Schaltfläche referenziert, um das `JButton`-Objekt auch später noch erreichen zu können.

[27] Ein `JButton`-Objekt ist eine Komponente, die beim Aktualisieren des Bildschirminhaltes automatisch neu gezeichnet wird. Das heißt, daß Sie Schaltflächen oder andere Steuerelemente nicht explizit zeichnen müssen, sondern daß es genügt, die Komponenten auf dem Formular zu positionieren und automatisch zeichnen zu lassen. In der Regel werden Schaltflächen im Konstruktor eines Formulars positioniert:

```
//: gui/Button1.java
// Putting buttons on a Swing application.
```

```
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Button1 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    public Button1() {
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new Button1(), 200, 100);
    }
} ///:~
```

Das Beispiel enthält noch ein anderes neues Element: Bevor Komponenten im Darstellungsbereich des `JFrame`-Objektes positioniert werden, erhält es einen sogenannten *Layoutmanager* vom Typ `FlowLayout`. Der Layoutmanager definiert, wie die Komponenten im Darstellungsbereich des Formulars angeordnet werden. `JFrame`-Objekte verwenden per Voreinstellung `BorderLayout` als Layoutmanager, bei dem allerdings jede neu hinzugefügte Komponente die bereits vorhandenen vollständig überdeckt (siehe später in diesem Kapitel). `FlowLayout` verteilt die Komponenten dagegen gleichmäßig auf dem Formular von links nach rechts und von oben nach unten.

Übungsaufgabe 4: (1) Verifizieren Sie, daß das Beispiel *Button1.java* nur eine Schaltfläche anzeigt, wenn die Methode `setLayout()` nicht aufgerufen wird. ■

23.4 Abfangen von Ereignissen

[28] Wenn Sie das obige Beispiel (*Button1.java*) übersetzen und ausführen, werden Sie feststellen, daß nichts geschieht, wenn Sie eine der Schaltflächen betätigen. An dieser Stelle müssen Sie eingreifen und Anweisungen hinterlegen, um zu definieren was geschieht. Die Grundaufgabe der ereignisgetriebenen Programmierung, welche die GUI-Programmierung in weiten Teilen umfaßt, besteht darin, Ereignisse mit Anweisungen zu verknüpfen, mit denen das Programm auf das Eintreten eines Ereignisses reagiert.

[29] Dieser Mechanismus ist bei Swing auf der Grundlage einer sauberen Trennung zwischen der Benutzerschnittstelle (der GUI-Komponente) und dem Ereignisbehandler (den Anweisungen, die bei Eintritt eines bestimmten Ereignisses an einer GUI-Komponente verarbeitet werden sollen) gegeben. Eine Swing-Komponente kann jeden möglichen Ereignistyp individuell berichten. Wenn Sie sich beispielsweise nicht dafür interessieren, ob der Mauszeiger über Ihre Schaltfläche bewegt wird, so registrieren Sie diesen Ereignistyp einfach nicht. Dies ist eine sehr einfache und elegante Umsetzung der ereignisgetriebenen Programmierung und wenn Sie die grundlegenden Konzepte einmal verstanden haben, können Sie mühelos mit Swing-Komponenten arbeiten, die Sie noch nie zuvor gesehen haben. Dieses Modell erstreckt sich sogar auf jeden Typ, der die Eigenschaften und Fähigkeiten einer JavaBean hat (siehe später in diesem Kapitel).

[30] Wir konzentrieren uns für's erste auf das Hauptereignis der Komponenten. Bei einer `JButton`-Komponente ist es die Betätigung der Schaltfläche. Sie registrieren Ihr Interesse an der Betätigung der Schaltfläche, indem Sie die `addActionListener()`-Methode des `JButton`-Objektes aufrufen. Diese Methode erwartet ein Argument vom Typ *`ActionListener`*. Das Interface *`ActionListen-`*

er deklariert eine einzige Methode namens `actionPerformed()`. Wenn Sie Anweisungen mit einer Schaltfläche verknüpfen möchten, implementieren Sie also das Interface *`ActionListener`* und registrieren ein Objekt dieser Implementierung per `addActionListener()` bei der `JButton`-Komponente. Die `actionPerformed()`-Methode wird bei Betätigung der Schaltfläche aufgerufen (eine solche Methode wird üblicherweise als *Rückruffunktion* (*callback*) bezeichnet).

[31] Was soll beim Betätigen der Schaltfläche geschehen? Wir wollen, daß sich auf dem Bildschirm etwas verändert und führen zu diesem Zweck eine neue Swing-Komponente ein: `JTextField` (ein Texteingabefeld). Hier können der Benutzer oder das Programm Textzeichen eingeben beziehungsweise darstellen. Es gibt mehrere Möglichkeiten, ein Texteingabefeld zu erzeugen. Die einfachste ist, dem Konstruktor die Breite des Feldes zu übergeben. Der Inhalt eines Texteingabefeldes auf dem Bildschirm kann mit Hilfe der Methode `setText()` geändert werden (die Klasse `JTextField` hat noch zahlreiche andere Methoden, die Sie in der API-Dokumentation nachlesen können):

```
//: gui/Button2.java
// Responding to button presses.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2 extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    class ButtonListener implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String name = ((JButton)e.getSource()).getText();
            txt.setText(name);
        }
    }
    private ButtonListener bl = new ButtonListener();
    public Button2() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2(), 200, 150);
    }
} ///:~
```

[32] Das Erzeugen eines `JTextField`-Objektes und seine Positionierung im Darstellungsbereich erfordern dieselben Schritte wie bei `JButton` oder jeder anderen Swing-Komponente. Der Unterschied zum vorigen Beispiel (*Button1.java*) ist die *`ActionListener`*-Klasse `ButtonListener`. Das Argument der `actionPerformed()`-Methode ist ein Objekt vom Typ `ActionEvent` und enthält sämtliche Informationen über das Ereignis und seine Herkunft. Die beiden Schaltflächen sollen einfach anzeigen, daß sie betätigt wurden. Die `getSource()`-Methode liefert das Objekt, von dem das Ereignis stammt und der Rückgabewert wird in den Typ `JButton` umgewandelt. Die `getText()`-Methode gibt die Beschriftung der Schaltfläche zurück, welche in das Texteingabefeld eingesetzt wird, um zu zeigen, daß die Anweisungen im Ereignisbehandler nach Betätigung der Schaltfläche tatsächlich aufgerufen wurden.

[33] Im Konstruktor der Klasse `Button2` wird mittels `addActionListener()` jeder der beiden Schaltflächen ein Ereignisbehandler vom Typ `ButtonListener` zugewiesen.

[34] Es ist häufig bequemer, den Ereignisbehandler als anonyme innere Klasse anzulegen, insbesondere dann, wenn Sie nur ein einziges Objekt der Ereignisbehandlerklasse brauchen. Das folgende Beispiel zeigt nochmals *Button2.java*, diesmal mit dem Ereignisbehandler als anonyme innere Klasse:

```
//: gui/Button2b.java
// Using anonymous inner classes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Button2b extends JFrame {
    private JButton
        b1 = new JButton("Button 1"),
        b2 = new JButton("Button 2");
    private JTextField txt = new JTextField(10);
    private ActionListener bl = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String name = ((JButton)e.getSource()).getText();
                txt.setText(name);
            }
        };
    public Button2b() {
        b1.addActionListener(bl);
        b2.addActionListener(bl);
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(txt);
    }
    public static void main(String[] args) {
        run(new Button2b(), 200, 150);
    }
} ///:~
```

Der Ansatz mit einer anonymen inneren Klasse wird (soweit möglich) bei den Beispielen in diesem Buch bevorzugt.

Übungsaufgabe 5: (4) Schreiben Sie mit Hilfe der Klasse `SwingConsole` aus Unterabschnitt 23.2.1 eine kleine Anwendung, die ein Texteingabefeld und drei Schaltflächen hat. Setzen Sie beim Betätigen jeder Schaltflächen einen anderen Text in das Texteingabefeld ein. ■

23.5 Textbereiche

[35] Ein Textbereich (`JTextArea`-Komponente) ist wie ein Texteingabefeld mit mehreren Eingabezeilen und mehr Funktionalität. Die Methode `append()` gestattet Ihnen, mühelos Ausgabezeilen an den Inhalt eines Textbereiches anzufügen. Im Unterschied zu Kommandzeilenprogrammen mit Ausgabe über den Standardausgabekanal können Sie die Ausgabe in einem Textbereich rückwärts rollen. Das folgende Beispiel zeigt den Inhalt der Klasse `net.mindview.util.Countries` aus Kapitel 18 in einer `JTextArea`-Komponente an:

```
//: gui/TextArea.java
// Using the JTextArea control.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextArea extends JFrame {
    private JButton
        b = new JButton("Add Data"),
        c = new JButton("Clear Data");
    private JTextArea t = new JTextArea(20, 40);
    private Map<String,String> m = new HashMap<String,String>();
    public TextArea() {
        // Use up all the data:
        m.putAll(Countries.capitals());
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    for(Map.Entry me : m.entrySet())
                        t.append(me.getKey() + ": "+ me.getValue()+"\n");
                }
            });
        c.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    t.setText("");
                }
            });
        setLayout(new FlowLayout());
        add(new JScrollPane(t));
        add(b);
        add(c);
    }
    public static void main(String[] args) {
        run(new TextArea(), 475, 425);
    }
} ///:~
```

Im Konstruktor wird das *Map*-Objekt `m` mit den Namen der Länder und ihrer Hauptstädte initialisiert. Beachten Sie, daß der Ereignisbehandler (Typ *`ActionListener`*) bei beiden Schaltflächen zugewiesen wird, ohne die Referenzen in einem Feld zwischenzuspeichern, da die Ereignisbehandlerobjekte an keiner weiteren Stelle im Programm benötigt werden. Die „Add Data"-Schaltfläche „formatiert" die Daten und fügt sie an den Inhalt des Textbereiches an. Die „Clear Data"-Schaltfläche ruft die `setText()`-Methode auf, um den Inhalt des Textbereiches zu löschen.

[36] Die `JTextArea`-Komponente wird vor der Übergabe an den Darstellungsbereich in einer `JScrollPane`-Komponente verpackt, um den Inhalt vorwärts oder rückwärts rollen zu können, falls der übergebene Text zu groß ist. Dies genügt, um Rollfunktionalität zu erwirken. Nachdem ich bei verschiedenen anderen GUI-Bibliotheken versucht habe, herauszufinden wie sich diese Funktionalität bewerkstelligen läßt, bin ich sehr beeindruckt von der Einfachheit und der Güte des Designs von Komponenten wie `JScrollPane`.

Übungsaufgabe 6: (7) Wandeln Sie das Beispiel *strings/TestRegularExpression.java* in ein interaktives Swing-Programm um, welches den zu durchsuchenden Text in einem Textbereich und den regulären Ausdruck in einem Texteingabefeld erwartet. Das Ergebnis wird in einem zweiten Textbereich angezeigt. ■

Übungsaufgabe 7: (5) Schreiben Sie mit Hilfe der Klasse `SwingConsole` aus Unterabschnitt 23.2.1 eine Anwendung, die alle Swing-Komponenten enthält, welche eine `addActionListener()`-Methode haben. (Tip: Schlagen Sie die Komponenten in der Dokumentation des Java Development Kits nach und suchen Sie im Index nach `addActionListener()`). Fangen Sie die Ereignisse ab und geben Sie jeweils eine passende Meldung in einem Texteingabefeld aus. ■

Übungsaufgabe 8: (6) Nahezu jede Swing-Komponente ist von der Klasse `java.awt.Component` abgeleitet, die eine Methode namens `setCursor()` definiert. Schlagen Sie die Beschreibung dieser Methode in der API-Dokumentation nach. Schreiben Sie eine Anwendung, die den Standardcursor durch einen der in der Klasse `java.awt.Cursor` definierten Typen ersetzt. ■

23.6 Die Layoutmanager von AWT

[37] Die Vorgehensweise mit der Sie bei Java die Komponenten auf einem Formular anordnen, unterscheidet sich wahrscheinlich von anderen GUI-Bibliotheken, mit den Sie bereits gearbeitet haben. Erstens wird alles programmiert. Es gibt keine „Resourcen“, mit denen die Platzierung der Komponenten steuern läßt. Zweitens werden die Komponenten nicht mittels absoluter Koordinaten im Darstellungsbereich positioniert, sondern mit Hilfe eines „Layoutmanagers“, der die Lage der Komponenten nach der Reihenfolge festlegt, in der sie dem Darstellungsbereich hinzugefügt wurden. Größe, Gestalt und Position der Komponenten unterscheiden sich von Layoutmanager zu Layoutmanager bemerkenswert. Außerdem passen sich die Layoutmanager den Abmessungen Ihres Applets oder Anwendungsfensters an, das heißt wenn sich die Ausdehnung des Fensters ändert, so ändern sich auch Größe, Gestalt und Position der Komponenten entsprechend.

[38] Objekte der Klassen `JApplet`, `JFrame`, `JWindow`, `JDialog`, `JPanel` und so weiter enthalten und visualisieren Komponenten. Die Klasse `java.awt.Container` definiert eine Methode namens `setLayout()`, die Ihnen gestattet, einen anderen Layoutmanager zu wählen. Wir untersuchen in diesem Abschnitt das Verhalten der verschiedenen Layoutmanager, indem wir Schaltflächen in einem Darstellungsbereich positionieren (einer der einfachsten Anwendungsfälle). In den Beispielen werden keine Ereignisse abgefangen, da wir uns nur für die Platzierung der Schaltflächen interessieren.

23.6.1 BorderLayout

[39] Solange Sie keinen anderen Typ wählen, verwendet eine `JFrame`-Komponente den voreingestellten Layoutmanager `BorderLayout`. Without any other instruction, platziert `BorderLayout` jede Komponente im Zentrum des Darstellungsbereiches und dehnt sie bis zu dessen Eckpunkten aus.

[40] Der Layoutmanager `BorderLayout` unterscheidet vier Randbereiche und einen im Zentrum angeordneten Teilbereich des Darstellungsbereiches. Wenn Sie in einem Darstellungsbereich, der den `BorderLayout`-Layoutmanager verwendet eine Komponente anlegen möchten, können Sie diejenige Version der überladenen `add()`-Methode aufrufen, welche als erstes Argument eine der vier folgenden Konstanten akzeptiert:

- `BorderLayOut.NORTH`: Komponente wird im oberen Randbereich positioniert.
- `BorderLayOut.SOUTH`: Komponente wird im unteren Randbereich positioniert.
- `BorderLayOut.EAST`: Komponente wird im rechten Randbereich positioniert.
- `BorderLayOut.WEST`: Komponente wird im linken Randbereich positioniert.
- `BorderLayOut.CENTER`: Komponente wird im zentralen Teilbereich positioniert.

Wenn Sie keinen Teilbereich angeben, wird die Komponente im zentralen Teilbereich (`CENTER`) platziert. Im folgenden Beispiel wird der voreingestellte Layoutmanager `BorderLayout` verwendet:

```
//: gui/BorderLayout1.java
// Demonstrates BorderLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class BorderLayout1 extends JFrame {
    public BorderLayout1() {
        add(BorderLayout.NORTH, new JButton("North"));
        add(BorderLayout.SOUTH, new JButton("South"));
        add(BorderLayout.EAST, new JButton("East"));
        add(BorderLayout.WEST, new JButton("West"));
        add(BorderLayout.CENTER, new JButton("Center"));
    }
    public static void main(String[] args) {
        run(new BorderLayout1(), 300, 250);
    }
} ///:~
```

In den vier Randbereichen wird die eingesetzte Komponente stets bezüglich einer Dimension gestaucht und bezüglich der anderen Dimension gestreckt. Im zentralen Teilbereich wird die Komponente in alle vier Richtungen gestreckt.

23.6.2 FlowLayout

[41] Der Layoutmanager `FlowLayout` läßt die Komponenten auf dem Formular zwischen dem linken und rechten Rand gleiten. Ist die oberste Reihe ausgefüllt, so beginnt der Layoutmanager eine weitere Reihe mit derselben Gleitfähigkeit.

[42–43] Das nächste Beispiel wählt den Layoutmanager `FlowLayout` und platziert einige Schaltflächen im Darstellungsbereich. Beachten Sie, daß die Komponenten bei `FlowLayout` ihre „natürliche“ Größe annehmen, das heißt, daß beispielsweise die Größe einer `JButton`-Komponente von den Abmessungen ihrer Beschriftung abhängt:

```
//: gui/FlowLayout1.java
// Demonstrates FlowLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class FlowLayout1 extends JFrame {
    public FlowLayout1() {
        setLayout(new FlowLayout());
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new FlowLayout1(), 300, 300);
    }
} ///:~
```

Da beim Layoutmanager `FlowLayout` alle Komponenten auf ihre kleinste Größe kompaktiert werden, kann das Ergebnis überraschend wirken. Eine Beschriftung (`JLabel`-Komponente) wird zum Beispiel bei `FlowLayout` stets auf die Abmessungen des Textes reduziert, so daß sich die Anzeige auch unter

rechtsbündiger Formatierung nicht ändert. Beachten Sie, daß die Komponenten bei Änderung der Fenstergröße neu positioniert werden.

23.6.3 GridLayout

[44] Der Layoutmanager `GridLayout` gestattet die Anordnung der Komponenten in einer Tabelle, wobei eine angelegte Komponente den in ihrer Zelle verfügbaren Raum zwischen oberem und unterem sowie zwischen linkem und rechtem Rand ausfüllt. Die benötigten Zeilen und Spalten werden per Konstruktor erfaßt und unterteilen den Darstellungsbereich gleichmäßig in Felder identischer Größe:

```
//: gui/GridLayout1.java
// Demonstrates GridLayout.
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class GridLayout1 extends JFrame {
    public GridLayout1() {
        setLayout(new GridLayout(7,3));
        for(int i = 0; i < 20; i++)
            add(new JButton("Button " + i));
    }
    public static void main(String[] args) {
        run(new GridLayout1(), 300, 300);
    }
} ///:~
```

Das Beispiel definiert 21 Zellen, aber nur 20 Schaltflächen. Die letzte Zelle bleibt leer, da der Layoutmanager `GridLayout` den verfügbaren Raum nicht neu aufteilt.

23.6.4 GridBagLayout

[45] Der Layoutmanager `GridBagLayout` bietet unglaubliche Steuerungsmöglichkeiten hinsichtlich des Layouts von Fensterbereichen und deren Reformatierung, wenn die Abmessungen des Fensters geändert werden. Es ist allerdings auch der kompliziertesteste Layoutmanager und nicht leicht zu verstehen. `GridBagLayout` ist in erster Linie dazu gedacht, Quelltext mit Hilfe eines GUI-Builders automatisch zu generieren. (Es ist möglich, daß der GUI-Builder `GridBagLayout` anstelle absoluter Positionierung verwendet.) Wenn Ihr Layout so kompliziert ist, daß Sie über den Einsatz von `GridBagLayout` nachdenken, dann sollten Sie erwägen, ein GUI-Builder zu verwenden. Wenn Sie das Gefühl haben, sich mit den verzwickten Einzelheiten auseinandersetzen zu müssen, empfehle ich Ihnen eines der diesem Zweck gewidmeten Swing-Bücher als Einstiegspunkt.

[46] Der nicht zur Swing-Bibliothek gehörige Layoutmanager `TableLayout` kommt als Alternative in Frage (siehe *http://java.sun.com*). Dieser Layoutmanager überlagert `GridBagLayout` und verdeckt den größten Teil von dessen Komplexität, wodurch der Umgang mit `GridBagLayout` erheblich vereinfacht wird.

23.6.5 Absolute Positionierung

[47] Die graphischen Komponenten können auch absolut positioniert werden:

- Übergeben Sie den Wert `null` als Layoutmanager Ihres Containers: `setLayout(null)`.

- Rufen Sie für jede Komponente die Methode `setBounds()` oder `reshape()` auf (je nach Sprachversion) und übergeben Sie ein Begrenzungsrechteck in Pixelkoordinaten. ~~You can do this in the constructor or in paint(), depending on what you want to achieve.~~

Einige GUI-Builder gebrauchen diesen Ansatz extensiv, aber es ist in der Regel nicht die beste Methode, um Quelltext zu generieren.

23.6.6 BoxLayout

[48] Nachdem viele Programmierer soviele Schwierigkeiten hatten, `GridBagLayout` zu verstehen und damit zu arbeiten, wurde die Swing-Bibliothek um den Layoutmanager `javax.swing.BoxLayout` (*nicht* im Package `java.awt`) ergänzt. `BoxLayout` bietet viele Vorteile von `GridBagLayout` ohne dessen Komplexität und bietet sich bei handgeschriebenen Layouts häufig an (wenn Ihr Layout zu komplex wird, sollten Sie aber einen GUI-Builder in Betracht ziehen, der das Layout für Sie generiert). Der Layoutmanager `BoxLayout` gestattet zusammen mit der Hilfsklasse `javax.swing.Box`, die Platzierung der Komponenten entweder in vertikaler oder in horizontaler Richtung sowie den Zwischenraum zwischen den Komponenten durch „Struts and Glue“ zu steuern, übersetzt etwa „Gestänge und Klebstoff“. Sie finden in den Online-Anhängen zu diesem Buch unter der Webadresse *http://www.mindview.net* einige grundlegende Beispiele für `BoxLayout`.

23.6.7 Welcher Layoutmanager eignet sich am besten?

[49] Swing ist mächtig. Sie können mit wenigen Zeilen viel erreichen. Die Beispiele in diesem Kapitel sind ziemlich einfach und es ist sinnvoll, sie zum Lernen selbst zu schreiben. Auch durch das Kombinieren einfacher Layouts läßt sich einiges bewerkstelligen. Ab einem gewissen Punkt ist es nicht mehr sinnvoll, GUIs manuell zu entwickeln, sondern wird zu kompliziert und ist keine sinvolle Verwendung Ihrer Arbeitszeit mehr. Die Entwickler von Java und Swing haben Sprache und Bibliothek so angelegt, daß sie den Einsatz von GUI-Buildern unterstützen, welche ausdrücklich dazu da sind, Ihnen die Programmierarbeit zu erleichtern. Solange Sie die Layouts und die Behandlung von Ereignissen (siehe folgenden Abschnitt) verstehen, ist es nicht besonders wichtig, daß Sie die Einzelheiten der Anordnung von Komponenten wirklich verstehen. Überlassen Sie diese Arbeit dem passenden Werkzeug (schließlich wurde Java entwickelt, um die Produktivität der Programmierer zu verbessern).

23.7 Das Swing-Ereignismodell

[50] Im Swing-Ereignismodell kann eine Komponente ein Ereignis auslösen („feuern“). Jeder Ereignistyp ist durch eine eigene Klasse vertreten. Ein ausgelöstes Ereignis kann von einem oder mehreren Ereignisbehandlern abgefangen werden, die auf dieses Ereignis reagieren. Die Quelle eines Ereignisses und der Ort an dem es behandelt wird, können also voneinander getrennt werden. Daß Sie typischerweise Swing-Komponenten in unveränderter Form einsetzen und anwendungsspezifische Anweisungen für den Fall hinterlegen, daß eine Komponente ein bestimmtes Ereignis verursacht, ist ein ausgezeichnetes Beispiel für die Trennung von Interface und Implementierung.

[51] Jeder Ereignisbehandler ist ein Objekt einer Klasse, die ein bestimmtes Interface implementiert. Sie haben als Programmierer nicht mehr zu tun, als ein Objekt der Ereignisbehandlerklasse zu erzeugen und bei der Komponente zu registrieren, die eventuell ein Ereignis dieses Typs auslöst. Die Registrierung erfolgt durch Aufrufen der Methode `addXXXListener()`, wobei `XXX` den Typ des Ereignisbehandlers angibt. Anhand der Namen der `addXXXListener()`-Methoden können Sie mühelos

in Erfahrung bringen, welche Ereignistypen eine Komponente unterstützt. Falls Sie versuchen, ein für eine bestimmte Komponente nicht definiertes Ereignis abzufangen, meldet der Compiler diesen Fehler zur Übersetzungszeit. Sie werden später in diesem Kapitel lernen, daß auch JavaBeans bei Methodennamen die `addXXXListener()`-Schreibweise verwenden, um beschreiben, welche Ereignisse eine JavaBean behandeln kann.

[52] Ihre gesamte Logik zur Behandlung eines Ereignisses befindet sich in der Ereignisbehandlerklasse. Wenn Sie eine Ereignisbehandlerklasse schreiben, besteht die einzige Beschränkung darin, daß Sie das passende Interface implementieren müssen. Sie können eine globale Ereignisbehandlerklasse anlegen, wobei sich innere Klassen in dieser Situation besonders anbieten; nicht alleine durch die logische Anordnung der Behandlerklassen in der Nähe der Benutzerschnittstelle beziehungsweise der Klassen in der Geschäftslogik, sondern auch durch die Eigenschaft, daß innere Klasse eine Referenz auf ihr Elternobjekt besitzen, wodurch Methodenaufrufe über Klassen- und Subsystemgrenzen hinweg möglich sind.

[53] Alle Beispiele in diesem Kapitel haben das Swing-Ereignismodell verwendet, aber der Rest dieses Abschnitts trägt die Einzelheiten dieses Modells zusammen.

23.7.1 Ereignis- und Ereignisbehandlertypen

[54] Jede Swing-Komponente verfügt über Methoden mit dem Namensschema `addXXXListener()` und `removeXXXListener()`, die das Registrieren von Ereignissen bei dieser Komponente beziehungsweise das Annullieren dort registrierter Ereignisse gestatten. Der Platzhalter `XXX` gibt zugleich den Typ des Argumentes der Methode an, zum Beispiel `addMyListener(MyListener m)`. Tabelle 23.1 auf Seite 1023 dokumentiert die wichtigsten Ereignistypen, Ereignisbehandler und Methoden sowie die wichtigsten Komponenten, die den jeweiligen Ereignistyp durch `addXXXListener()`- und `removeXXXListener()`-Methoden unterstützen. Beim Design des Ereignismodells wurde der Erweiterbarkeit Raum gegeben, das heißt Sie werden unter Umständen auf Ereignistypen und Ereignisbehandler stoßen, die in Tabelle 23.1 nicht vorkommen.

[55] Jeder Komponententyp unterstützt nur bestimmte Ereignistypen. Das Nachschlagen der pro Komponente unterstützten Ereignistypen ist recht ermüdend. Eine einfachere Lösung besteht darin, das Beispiel *ShowMethods.java* aus Abschnitt 15.6.1 (Seite 463) zu überarbeiten, damit es die von einer Swing-Komponente unterstützten Ereignistypen anzeigt.

[56] In Kapitel 15 wurde der Reflexionsmechanismus von Java vorgestellt und dazu verwendet, die Methoden einer bestimmten Klasse auszugeben und zwar entweder alle Methoden oder nur solche, die zu einem gegebenen Suchbegriff passen. Der Reflexionsmechanismus ist in der Lage, automatisch *alle* Methoden einer Klasse anzuzeigen, ohne daß Sie gezwungen sind, der Ableitungshierarchie zu folgen und die Basisklasse jeder einzelnen Ebene separat zu untersuchen. Der Reflexionsmechanismus spart somit Zeit: Da die meisten Methoden bei Java ausführliche deskriptive Namen haben, können Sie nach den Methoden suchen, deren Name ein bestimmtes Wort enthält. Wenn Sie eine Methode gefunden haben, die Ihnen passend erscheint, lesen Sie in der API-Dokumentation nach.

[57] Das folgende Beispiel ist eine Swing-Version von *ShowMethods.java*, speziell für die Suche nach den `addXXXListener()`-Methoden bei Swing-Komponenten:

```
//: gui/ShowAddListeners.java
// Display the ''addXXXListener'' methods of any Swing class.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.lang.reflect.*;
import java.util.regex.*;
```

```
import static net.mindview.util.SwingConsole.*;

public class ShowAddListeners extends JFrame {
    private JTextField name = new JTextField(25);
    private JTextArea results = new JTextArea(40, 65);
    private static Pattern addListener =
        Pattern.compile("(add\\w+?Listener\\(.*?\\))");
    private static Pattern qualifier =
        Pattern.compile("\\w+\\.");
    class NameL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String nm = name.getText().trim();
            if(nm.length() == 0) {
                results.setText("No match");
                return;
            }
            Class<?> kind;
            try {
                kind = Class.forName("javax.swing." + nm);
            } catch(ClassNotFoundException ex) {
                results.setText("No match");
                return;
            }
            Method[] methods = kind.getMethods();
            results.setText("");
            for(Method m : methods) {
                Matcher matcher = addListener.matcher(m.toString());
                if(matcher.find())
                    results.append(qualifier.matcher(
                        matcher.group(1)).replaceAll("") + "\n");
            }
        }
    }
    public ShowAddListeners() {
        NameL nameListener = new NameL();
        name.addActionListener(nameListener);
        JPanel top = new JPanel();
        top.add(new JLabel("Swing class name (press Enter):"));
        top.add(name);
        add(BorderLayout.NORTH, top);
        add(new JScrollPane(results));
        // Initial data and test:
        name.setText("JTextArea");
        nameListener.actionPerformed(new ActionEvent("", 0 ,""));
    }
    public static void main(String[] args) {
        run(new ShowAddListeners(), 500, 400);
    }
} ///:~
```

Der eingegebene Name der gesuchten Swing-Klasse steht in dem von `name` referenzierten Texteingabefeld. Das Ergebnis der Suche mit regulären Ausdrücken wird in dem von `results` referenzierten Textbereich angezeigt.

[58] Beachten Sie, daß es keine Schaltfläche oder sonstige Komponente gibt, um die Suche zu starten. Statt dessen hat das Texteingabefeld einen Ereignisbehandler. Die Ergebnisliste wird aktualisiert, wenn Sie einen Klassennamen eingeben und die Eingabetaste betätigen. Ist das Texteingabefeld nicht leer, so wird sein Inhalt verwendet, um mittels `Class.forName()` nach dem Klassenobjekt

Ereignisklasse, Behandlerinterface, add- und remove-Methode	**Komponenten, die dieses Ereignis unterstützen**
ActiveEvent, *ActionListener*, addActionListener() und removeActionListener()	JButton, JList, JTextField, JMenuItem und davon abgeleitete Komponenten, darunter JCheckBoxMenuItem, JMenu und JRadioButtonMenuItem
AdjustmentEvent, *AdjustmentListener*, addAdjustmentListener() und removeAdjustmentListener()	JScrollBar sowie jede Klasse, die das Interface *Adjustable* implementiert.
ComponentEvent, *ComponentListener*, addComponentListener() und removeComponentListener()	*Component und davon abgeleitete Klassen, darunter JButton, JCheckBox, JComboBox, Container, JPanel, JApplet, JScrollPane, Window, JDialog, FileDialog, JFrame, JLabel, JList, JScrollBar, JTextArea und JTextField
ContainerEvent, *ContainerListener* addContainerListener() und removeContainerListener()	Container und davon abgeleitete Klassen, darunter JScrollPane, Window, JDialog, FileDialog und JFrame
FocusEvent, *FocusListener*, addFocusListener() und removeFocusListener()	Container und derivatives*
KeyEvent, *KeyListener*, addKeyListener() und removeKeyListener()	Container und derivatives*
MouseEvent (Tasten und Bewegung), *MouseListener*, addMouseListener() und removeMouseListener()	Container und derivatives*
MouseEvent (Tasten und Bewegung), *MouseMotionListener*, addMouseMotionListener() und removeMouseMotionListener()	Container und derivatives*
WindowEvent, *WindowListener*, addWindowListener() und removeWindowListener()	Window und davon abgeleitete Klassen, darunter JDialog, FileDialog und JFrame
ItemEvent, *ItemListener*, addItemListener() und removeItemListener()	JCheckBox, JCheckBoxMenuItem, JComboBox, JList sowie jede Klasse, die das Interface *ItemSelectable* implementiert.
TextEvent, *TextListener*, addTextListener() und removeTextListener()	Jede von JTextComponent von abgeleitete Klassen, darunter JTextArea und JTextField

Tabelle 23.1: Die wichtigsten Ereignistypen, Ereignisbehandler und Methoden sowie die wichtigsten Komponenten, welche den jeweiligen Ereignistyp durch addXXXListener- und removeXXXListener-Methoden unterstützen. Beachten Sie, daß es keinen Ereignistyp MouseMotionEvent gibt, obwohl es so aussieht, als solle er existieren. Klicken und Bewegungen sind im Ereignistyp MouseEvent zusammengefaßt. Das zweite Vorkommen von MouseEvent in der Tabelle ist also kein Fehler.

zu suchen. Ist der gesuchte Klassenname falsch geschrieben, scheitert `forName()` und wirft eine Ausnahme vom Typ `ClassNotFoundException` aus. Die Ausnahme wird abgefangen und der Inhalt des Ergebnistextbereiches auf „No match“ gesetzt. Wenn Sie den gesuchten Klassennamen korrekt eingeben (beachten Sie die Groß-/Kleinschreibung), verläuft der Aufruf von `forName()` erfolgreich und die `getMethods()`-Methode liefert ein Array von `Method`-Objekten.

[59] Das Programm verwendet zwei reguläre Ausdrücke. Der erste Ausdruck (`addListeners`) sucht nach „add“, gefolgt von einem oder mehreren Wortzeichen, gefolgt von „Listeners“ sowie der eingeklammerten Argumentliste. Beachten Sie, daß der gesamte reguläre Ausdruck in ungeschützten Klammern steht, bei einem Treffer also als „Gruppe“ verfügbar ist. In der `NameL`-Methode `actionPerformed()` erzeugt die `Pattern`-Methode `matcher()` zu jeder Methode ein `Matcher`-Objekt. Die `find()`-Methode des `Matcher`-Objektes liefert nur dann `true`, wenn ein Treffer gefunden wurde. In diesem Fall wird die erste eingeklammerte Gruppe mittels `group(1)` zur Weiterverarbeitung ausgewählt. Die Zeichenkette enthält noch den Packagenamen, der mit Hilfe des zweiten Ausdrucks (`qualifier`) entfernt wird (siehe Beispiel *ShowMethods.java.* in Kapitel 15).

[60] Am Schluß des Konstruktors wird ein Startwert in das Texteingabefeld eingesetzt (`JTextArea`) und das erforderliche Ereignis ausgelöst, um die Suche mit dem Startwert auszuführen.

[61] Dieses Programm ist ein bequemes Werkzeug zum Untersuchen der Eigenschaften und Fähigkeiten von Swing-Komponenten. Nachdem Sie wissen, welche Ereignistypen eine bestimmte Komponente unterstützt, brauchen Sie nichts mehr nachzuschlagen, um eines dieser Ereignisse behandeln zu können:

- Entfernen Sie die Endung „Event“ vom Namen der Ereignisbehandlerklasse und hängen Sie statt dessen die Endung „Listener“ an das Restwort an. Sie erhalten den Namen des Interfaces, das Sie in Form einer inneren Klasse implementieren müssen.
- Implementieren Sie das obige Interface, in dem Sie die Methoden des Ereignisbehandlers ausprogrammieren. Wenn Sie sich für Mausbewegungen interessieren, so implementieren Sie die Methode `mouseMoved()` des Interfaces *`MouseMotionListener`*. (Sie müssen natürlich auch die anderen Methoden ausprogrammieren, wobei es hierfür eine Abkürzung gibt, wie Sie ~~später in diesem Kapitel~~ lernen werden.)
- Erzeugen Sie ein Objekt der Ereignisbehandlerklasse und registieren Sie es bei Ihrer Komponente mit der Methode, deren Namen Sie aus „add“ und dem Namen der Ereignisbehandlerklasse kombinieren, zum Beispiel `addMouseMotionListener()`.

[62] Tabelle 23.2 auf Seite 1025 zeigt eine Auswahl von Ereignisbehandlerinterfaces. Die Liste ist keinesfalls erschöpfend, unter anderem, weil Ihnen das Swing-Ereignismodell gestattet, eigene Ereignistypen und -behandler zu entwickeln. Sie werden daher immer wieder auf Bibliotheken stoßen, die eigene Ereignistypen definiert haben. Die in diesem Kapitel vermittelten Kenntnisse werden Ihnen helfen, zu verstehen, was diese Typen bedeuten.

23.7.1.1 Die Adapterklassen der Ereignisbehandlerinterfaces

[63] Einige Ereignisbehandlerinterfaces deklarieren nur eine einzige Methode (siehe Tabelle 23.2) und sind daher mühelos zu implementieren. Interfaces mit mehreren Methoden sind dagegen weniger angenehm zu verwenden. Wenn Sie beispielsweise einen Mausklick abfangen möchten (der nicht bereits abgefangen wird, wie etwa bei einer Schaltfläche), dann müssen Sie die Methode `mouseClicked()` ausprogrammieren. Da *`MouseListener`* ein Interface ist, müssen Sie aber auch alle anderen dort deklarierten Methoden implementieren. Das ist ein lästiges Unterfangen.

Ereignisbehandlerinterface und -Adapterklasse	**Im Interface deklarierte Methoden**
ActionListener	actionPerformed()
AdjustmentListener	adjustmentValueChanged(Adjustmentevent)
ComponentListener, ComponentAdapter	componentHidden(ComponentEvent), componentShown(ComponentEvent), componentMoved(ComponentEvent) und componentResized(ComponentEvent)
ContainerListener, ContainerAdapter	componentAdded(ContainerEvent) und componentRemoved(ContainerEvent)
FocusListener, FocusAdapter	focusGained(FocusEvent) und focusLost(FocusEvent)
KeyListener, KeyAdapter	keyPressed(KeyEvent), keyReleased(KeyEvent) und keyTyped(KeyEvent)
MouseListener, MouseAdapter	mouseClicked(MouseEvent), mouseEntered(MouseEvent), mouseExited(MouseEvent), mousePressed(MouseEvent) und mouseRelease(MouseEvent)
MouseMotionListener, MouseMotionAdapter	mouseDragged(MouseEvent) und mouseMoved(MouseEvent)
WindowListener, WindowAdapter	windowOpened(WindowEvent), windowClosing(WindowEvent), windowClosed(WindowEvent), windowActivated(WindowEvent), windowDeactivated(WindowEvent), windowIconified(WindowEvent) und windowDeiconified(WindowEvent)
ItemListener	itemStateChanged(ItemEvent)

Tabelle 23.2: Einige Ereignisbehandlerinterfaces, teilweise mit Adapterklassen und die dort deklarierten Methoden.

[64] Einige (allerdings nicht alle) Ereignisbehandlerinterfaces mit zwei oder mehr Methode haben aus diesem Grund Adapterklassen, deren Namen Sie in Tabelle 23.2 ablesen können. Jede Adapterklasse liefert zu jeder im entsprechenden Interface deklarierten Methode eine leere Standardimplementierung. Wenn Sie eine Behandlerklasse von einer solchen Adapterklasse ableiten, überschreiben Sie nur die tatsächlich benötigten Methoden. Der Ereignisbehandler für den oben angeführten Mausklick könnte zum Beispiel so aussehen:

```
class MyMouseListener extends MouseAdapter {
    public void mouseClicked(MouseEvent e) {
        // Respond to mouse click...
    }
}
```

Sinn und Zweck der Adapterklassen ist, Ihnen das Schreiben von Ereignisbehandlerklassen zu erleichtern.

[65] Adapterklasse haben aber auch einen Nachteil, genauer einen Fallstrick. Angenommen Sie leiten einen Ereignisbehandler von `MouseAdapter` ab, wie oben:

```
class MyMouseListener extends MouseAdapter {
    public void MouseClicked(MouseEvent e) {
        // Respond to mouse click...
```

```
        }
    }
```

Der Ereignisbehandler funktioniert nicht und es wird Sie wahnsinnig machen, herauszufinden warum, da sich das Programm übersetzen läßt und tadellos funktioniert, abgesehen davon, daß beim Mausklick Ihre Methode nicht aufgerufen wird. Erkennen Sie das Problem? Es ist der Methodenname: `MouseClicked()` statt `mouseClicked()`. Ein einfacher Ausrutscher in der Groß-/Kleinschreibung des Methodennamens führt dazu, daß die Behandlerklasse eine neue Methode erhält. Allerdings ist dies nicht die Methode, die bei einem Mausklick aufgerufen wird und Sie bekommen nicht das gewünschte Ergebnis. Trotz der Unbequemlichkeit gewährleistet das direkte Implementieren eines Interfaces, daß die deklarierten Methoden korrekt ausprogrammiert sind.

[66] Eine bessere Alternative, um zu gewährleisten, daß Sie eine Methode tatsächlich überschreiben, ist die Anwendung der eingebauten Annotation `@Override`.

Übungsaufgabe 9: (5) Entwickeln Sie, ausgehend vom Beispiel *ShowAddListeners.java* (Seite 1021), ein Programm mit dem vollen Funktionsumfang des Beispiels *typeinfo.ShowMethods.java* aus Kapitel 15 ■

23.7.2 Verfolgung mehrerer Ergebnisse

[67] Es lohnt sich, ein Programm zu schreiben, welches das Verhalten einer `JButton`-Komponente über die Statusinformation „Schaltfläche betätigt“ beziehungsweise „Schaltfläche nicht betätigt“ hinaus verfolgt. Dieses Beispiel zeigt insbesondere, wie Sie eine eigene Schaltfläche von `JButton` ableiten können.[6]

[68] Im folgenden Beispiel ist `MyButton` eine innere Klasse der Klasse `TrackEvent`, kann also seine Elternkomponente erreichen und dessen Texteingabefelder ändern. Dies ist notwending, um die Statusinformationen in den Feldern der Elternkomponente zu hinterlegen. Es ist eine eingeschränkte Lösung, da `MyButton` nur zusammen mit `TrackEvent` verwendet werden kann. Ein solches Verhältnis wird als „stark gekoppelt“ bezeichnet.

```
//: gui/TrackEvent.java
// Show events as they happen.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class TrackEvent extends JFrame {
    private HashMap<String,JTextField> h = new HashMap<String,JTextField>();
    private String[] event = {
        ''focusGained'', ''focusLost'', ''keyPressed'',
        ''keyReleased'', ''keyTyped'', ''mouseClicked'',
        ''mouseEntered'', ''mouseExited'', ''mousePressed'',
        ''mouseReleased'', ''mouseDragged'', ''mouseMoved''
    };
    private MyButton
        b1 = new MyButton(Color.BLUE, ''test1''),
        b2 = new MyButton(Color.RED, ''test2'');
    class MyButton extends JButton {
        void report(String field, String msg) {
```

[6] Bei Java 1.0 und Java 1.1 war es nicht möglich, eine brauchbare Klasse von `JButton` abzuleiten. Dies war einer von zahlreichen fundamenalen Designfehlern.

```
            h.get(field).setText(msg);
        }
        FocusListener fl = new FocusListener() {
                public void focusGained(FocusEvent e) {
                    report(''focusGained'', e.paramString());
                }
                public void focusLost(FocusEvent e) {
                    report(''focusLost'', e.paramString());
                }
            };
        KeyListener kl = new KeyListener() {
                public void keyPressed(KeyEvent e) {
                    report(''keyPressed'', e.paramString());
                }
                public void keyReleased(KeyEvent e) {
                    report(''keyReleased'', e.paramString());
                }
                public void keyTyped(KeyEvent e) {
                    report(''keyTyped'', e.paramString());
                }
            };
        MouseListener ml = new MouseListener() {
                public void mouseClicked(MouseEvent e) {
                    report(''mouseClicked'', e.paramString());
                }
                public void mouseEntered(MouseEvent e) {
                    report(''mouseEntered'', e.paramString());
                }
                public void mouseExited(MouseEvent e) {
                    report(''mouseExited'', e.paramString());
                }
                public void mousePressed(MouseEvent e) {
                    report(''mousePressed'', e.paramString());
                }
                public void mouseReleased(MouseEvent e) {
                    report(''mouseReleased'', e.paramString());
                }
            };
        MouseMotionListener mml = new MouseMotionListener() {
                public void mouseDragged(MouseEvent e) {
                    report(''mouseDragged'', e.paramString());
                }
                public void mouseMoved(MouseEvent e) {
                    report(''mouseMoved'', e.paramString());
                }
            };
        public MyButton(Color color, String label) {
            super(label);
            setBackground(color);
            addFocusListener(fl);
            addKeyListener(kl);
            addMouseListener(ml);
            addMouseMotionListener(mml);
        }
    }
    public TrackEvent() {
        setLayout(new GridLayout(event.length + 1, 2));
```

```
            for(String evt : event) {
                JTextField t = new JTextField();
                t.setEditable(false);
                add(new JLabel(evt, JLabel.RIGHT));
                add(t);
                h.put(evt, t);
            }
            add(b1);
            add(b2);
        }
        public static void main(String[] args) {
            run(new TrackEvent(), 700, 500);
        }
    } ///:~
```

Der Konstruktor der Klasse `MyButton` konfiguriert per `setBackground()` die Farbe der Schaltfläche. Die Ereignisbehandler werden mit Hilfe der `addXXXListener()`-Methoden registiert.

[69] Die Klasse `TrackEvent` besitzt ein `HashMap`-Objekt, welches den Ereignistyp (eine Zeichenkette) auf eine `JTextField`-Komponente abbildet, über welche die Informationen zu diesem Ereignis angezeigt werden. Die Daten hätten auch statisch gespeichert werden können, aber Sie werden sicher zustimmen, daß sie per `HashMap` leichter und benutzen und zu ändern sind. Wenn Sie einen Ereignistyp zu `TrackEvent` hinzufügen oder aus dem Programm entfernen möchten genügt es, den entsprechenden Eintrag im `event`-Array vorzunehmen beziehungsweise dort zu löschen und alles übrige geschieht automatisch.

[70] Die `report()`-Methode erwartet einen Ereignisnamen und die Parameterliste des Ereignisses. Die Methode schlägt das mit dem Ereignisnamen verknüpfte Texteingabefeld im `HashMap`-Objekt der äußeren Klasse nach und speichert die Parameterliste anschließend in diesem Feld.

[71] Es ist nett mit diesem Beispiel zu spielen, weil Sie wirklich sehen können, was mit den Ereignissen Ihrem Programm geschieht.

Übungsaufgabe 10: (6) Schreiben Sie mit Hilfe der Klasse `SwingConsole` aus Unterabschnitt 23.2.1 eine Anwendung mit einer Schaltfläche (`JButton`) und einem Texteingabefeld (`JTextField`). Schreiben und verknüpfen Sie den passenden Ereignisbehandler, so daß eingegebene Zeichen im Texteingabefeld erscheinen, wenn die Schaltfläche den Fokus hat. ■

Übungsaufgabe 11: (4) Leiten Sie eine eigene Klasse von `JButton` ab. Bei jeder Betätigung der Schaltfläche soll die Hintergrundfarbe einem zufällig gewählten Wert entsprechend geändert werden. Sehen können im Beispiel *ColorBoxes.java* (Seite 1073) nachlesen, wie Sie Farbwerte zufällig bestimmen können. ■

Übungsaufgabe 12: (4) Tragen Sie im Beispiel *TrackEvent.java* einen neuen Ereignistyp mit Ereignisbehandler ein. Finden Sie selbst heraus, welchen Typ Sie wählen können. ■

23.8 Die wichtigsten Swing-Komponenten (Auswahl)

[72] Nachdem Sie Layoutmanager und das Swing-Ereignismodell verstehen, sind Sie ausreichend vorbereitet, um die Anwendung der verschiedenen Swing-Komponenten kennenzulernen. Dieser Abschnitt präsentiert eine Auswahl von Swing-Komponenten sowie der Eigenschaften und Fähigkeiten, die Sie voraussichtlich am häufigsten brauchen werden. Die einzelnen Beispiele sind bewußt relativ kurz, damit Sie den Quelltext in Ihre eigenen Programme übernehmen können.

[73] Beachten Sie:

- Sie können mühelos nachvollziehen, was jedes dieser Beispiele tut, indem Sie die Quelltextbeispiele dieses Kapitels herunterladen (*http://www.mindview.net*).
- Die Dokumentation des Java Development Kits beschreibt alle Klassen und alle Methoden der Swing-Bibliothek (in diesem Kapitel werden nur wenige behandelt).
- Aufgrund der Namenskonvention bei den Swing-Ereignissen, läßt sich einfach erraten, welcher ein Ereignisbehandler für einen bestimmten Ereignistyp geschrieben und wie er installiert werden muß. Verwenden Sie das Hilfsprogramm *ShowAddListeners.java* von Seite 1021 zur Untersuchung des Ereignisverhaltens einer bestimmten Komponente.
- Bei komplizierteren Anforderungen ist es ratsam einen GUI-Builder zu verwenden.

23.8.1 Schaltflächen und Schaltflächengruppen

[74] Swing hat eine Reihe verschiedener Schaltflächen. Alle Schaltflächen, das heißt Ankreuzfelder (*check boxes*), Radiobuttons und sogar Menüpunkte (*menu items*) sind von `AbstractButton` abgeleitet. (Da auch Menüpunkte dazugehören, wäre „`AbstractSelector`“ oder eine gleichermaßen allgemeingültige Bezeichnung vielleicht eine bessere Wahl gewesen.) Sie finden ~~später in diesem Kapitel~~ Beispiele für Menüpunkte. Das folgende Beispiel zeigt die verschiedenen Typen von Schaltflächen:

```
//: gui/Buttons.java
// Various Swing buttons.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.plaf.basic.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Buttons extends JFrame {
    private JButton jb = new JButton("JButton");
    private BasicArrowButton
        up = new BasicArrowButton(BasicArrowButton.NORTH),
        down = new BasicArrowButton(BasicArrowButton.SOUTH),
        right = new BasicArrowButton(BasicArrowButton.EAST),
        left = new BasicArrowButton(BasicArrowButton.WEST);
    public Buttons() {
        setLayout(new FlowLayout());
        add(jb);
        add(new JToggleButton("JToggleButton"));
        add(new JCheckBox("JCheckBox"));
        add(new JRadioButton("JRadioButton"));
        JPanel jp = new JPanel();
        jp.setBorder(new TitledBorder("Directions"));
        jp.add(up);
        jp.add(down);
        jp.add(left);
        jp.add(right);
        add(jp);
    }
    public static void main(String[] args) {
        run(new Buttons(), 350, 200);
    }
} ///:~
```

Die ersten vier Schaltflächen `up`, `down`, `right` und `left` sind vom Typ `BasicArrowButton` (Pfeiltasten) aus dem Package `javax.swing.plaf.basic`, then continues with the various specific types of buttons. Wenn Sie das Beispiel ausprobieren, stellen Sie fest, daß die „JToggleButton"-Schaltfläche ihren zuletzt eingenommenen Zustand behält („ein" beziehungsweise „ausgeschaltet"). Das Ankreuzfeld und der Radiobutton verhalten sich identisch, wiederum entweder „ein" oder „aus" (beide Klassen sind von `JToggleButton`).

23.8.1.1 Schaltflächengruppen

[75] Wenn mehrere Radiobuttons „Exklusiv-Oder-Verhalten" zeigen sollen, müssen Sie sie zu einer Schaltflächengruppe (`ButtonGroup`) zusammenfassen. Das folgende Beispiel zeigt, daß jede von `AbstractButton` abgeleitete Schaltfläche Element einer solchen Gruppe sein kann.

[76] Die Schaltflächengruppen werden in diesem Beispiel mit Hilfe des Reflexionsmechanismus erzeugt, um die unnötige Wiederholung vieler Anweisungen zu umgehen. Die `makeBPanel()`-Methode erzeugt eine Schaltflächengruppe im Darstellungsbereich einer `JPanel`-Komponente. Das zweite Argument dieser Methode ist ein `String`-Array. Die Methode legt je `String`-Objekt aus diesem Array eine mit dem Namen des Typs beschriftete Schaltfläche an, welcher `makeBPanel()` als erstes Argument übergeben wird:

```
//: gui/ButtonGroups.java
// Uses reflection to create groups
// of different types of AbstractButton.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class ButtonGroups extends JFrame {
    private static String[] ids = {
        ''June'', ''Ward'', ''Beaver'', ''Wally'', ''Eddie'', ''Lumpy''
    };
    static JPanel
        makeBPanel(Class<? extends AbstractButton> kind, String[] ids) {
        ButtonGroup bg = new ButtonGroup();
        JPanel jp = new JPanel();
        String title = kind.getName();
        title = title.substring(title.lastIndexOf('.') + 1);
        jp.setBorder(new TitledBorder(title));
        for(String id : ids) {
            AbstractButton ab = new JButton(''failed'');
            try {
                // Get the dynamic constructor method
                // that takes a String argument:
                Constructor ctor = kind.getConstructor(String.class);
                // Create a new object:
                ab = (AbstractButton)ctor.newInstance(id);
            } catch(Exception ex) {
                System.err.println(''can't create '' + kind);
            }
            bg.add(ab);
            jp.add(ab);
        }
        return jp;
    }
```

```
    public ButtonGroups() {
        setLayout(new FlowLayout());
        add(makeBPanel(JButton.class, ids));
        add(makeBPanel(JToggleButton.class, ids));
        add(makeBPanel(JCheckBox.class, ids));
        add(makeBPanel(JRadioButton.class, ids));
    }
    public static void main(String[] args) {
        run(new ButtonGroups(), 500, 350);
    }
} ///:~
```

[77] Die Beschriftung des Rahmens um die Schaltflächengruppe ist der Name der Klasse ohne Packagezugehörigkeit. Das `AbstractButton`-Feld `ab` wird mit einer `JButton`-Komponente mit der Beschriftung „failed“ vorbelegt, so daß Sie eventuelle Probleme auf dem Bildschirm sehen, falls Sie die Meldung durch den Ausnahmebehandler nicht beachten. Die `Class`-Methode `getConstructor()` liefert ein `Constructor`-Objekt, dessen `newInstance()`-Methode ein Array von Argumenten der Typen erwartet, die `getConstructor()` in Form von Klassenobjekten übergeben wurden. Hier wird `newInstance()` mit einem `String`-Objekt und dem von `ids` referenzierten `String`-Array aufgerufen.

[78] Sie erhalten das „Exklusiv-Oder-Verhalten“ bei Schaltflächen, indem Sie eine Schaltflächengruppe (`ButtonGroup`) definieren und jede Schaltfläche, die dieses Verhalten annehmen soll, dieser Gruppe hinzufügen. Wenn Sie das Programm aufrufen, sehen Sie, daß alle Schaltflächen außer `JButton` das „Exklusiv-Oder-Verhalten“ zeigen.

23.8.2 Icons

[79] Icons (*`Icon`*-Objekte) sind bei `JLabel`-Komponenten und sämtlichen von `AbstractButton` abgeleiteten Typen erlaubt (darunter `JButton`, `JCheckBox`, `JRadioButton` und die verschiedenen Untertypen von `JMenuItem`). Die Zuweisung eines Icons an eine `JLabel`-Komponente ist recht einfach (siehe später in diesem Kapitel). Das folgende Beispiel zeigt die übrigen Anwendungsfälle von Icons bei `JButton`-Komponenten und den davon abgeleiteten Typen.

[80] Sie können beliebige *.gif* Dateien als Icons verwenden. Die Icons zu diesem Beispiel stammen aus der Quelltextdistribution dieses Buches, die Sie von *http://www.mindview.net* herunterladen können. Verwenden Sie ein `ImageIcon`-Objekt, dem Sie den Dateinamen übergeben, um eine Graphikdatei zu öffnen und ein Icon in das Programm zu integrieren. Anschließend können Sie das erhaltene *`Icon`*-Objekt im Programm verwenden:

```
//: gui/Faces.java
// Icon behavior in JButtons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Faces extends JFrame {
    private static Icon[] faces;
    private JButton jb, jb2 = new JButton("Disable");
    private boolean mad = false;
    public Faces() {
        faces = new Icon[] {
            new ImageIcon(getClass().getResource("Face0.gif")),
            new ImageIcon(getClass().getResource("Face1.gif")),
            new ImageIcon(getClass().getResource("Face2.gif")),
```

```
            new ImageIcon(getClass().getResource("Face3.gif")),
            new ImageIcon(getClass().getResource("Face4.gif")),
        };
        jb = new JButton("JButton", faces[3]);
        setLayout(new FlowLayout());
        jb.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if(mad) {
                        jb.setIcon(faces[3]);
                        mad = false;
                    } else {
                        jb.setIcon(faces[0]);
                        mad = true;
                    }
                    jb.setVerticalAlignment(JButton.TOP);
                    jb.setHorizontalAlignment(JButton.LEFT);
                }
            });
        jb.setRolloverEnabled(true);
        jb.setRolloverIcon(faces[1]);
        jb.setPressedIcon(faces[2]);
        jb.setDisabledIcon(faces[4]);
        jb.setToolTipText("Yow!");
        add(jb);
        jb2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if(jb.isEnabled()) {
                        jb.setEnabled(false);
                        jb2.setText("Enable");
                    } else {
                        jb.setEnabled(true);
                        jb2.setText("Disable");
                    }
                }
            });
        add(jb2);
    }
    public static void main(String[] args) {
        run(new Faces(), 250, 125);
    }
} ///:~
```

Die Konstruktoren vieler Swing-Komponenten erwarten ein *Icon*-Objekt, aber Sie können auch die `setIcon()`-Methode aufrufen, um ein Icon auszutauschen. Das Beispiel zeigt darüber hinaus, wie einer `JButton`-Komponente (oder einem anderen von `AbstractButton` abgeleiteten Typ) je nach Ereignis unterschiedliche Icons angezeigt werden können, zum Beispiel beim Betätigen oder Deaktivieren der Schaltfläche oder wenn der Mauszeiger ohne zu Klicken über die Schaltfläche bewegt wird. Die Schaltfläche wirkt dadurch animiert.

23.8.3 Tooltips

[81] Die linke Schaltfläche im vorigen Beispiel (*Faces.java*) hatte einen Tooltip. Fast alle Komponenten, die Sie beim Anlegen einer GUI wählen können, sind von der Klasse `JComponent` abgeleitet, die eine Methode namens `setToolTipText(String)` definiert. Referenziert `jc` eine beliebige von `JComponent` abgeleitete Komponente Ihres Formulars, so genügt

```
jc.setToolTipText("My tip");
```

um einen Tooltip für diese Komponente anzulegen. Verweilt der Mauszeiger für eine vordefinierte Zeitspanne über dieser Komponente, so erscheint neben dem Mauszeiger ein kleiner Rahmen mit dem Text „My tip“.

23.8.4 Texteingabefelder

[82] Das folgende Beispiel führt die Funktionalität von Texteingabefeldern vor:

```
//: gui/TextFields.java
// Text fields and Java events.
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.text.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TextFields extends JFrame {
    private JButton
        b1 = new JButton("Get Text"),
        b2 = new JButton("Set Text");
    private JTextField
        t1 = new JTextField(30),
        t2 = new JTextField(30),
        t3 = new JTextField(30);
    private String s = "";
    private UpperCaseDocument ucd = new UpperCaseDocument();
    public TextFields() {
        t1.setDocument(ucd);
        ucd.addDocumentListener(new T1());
        b1.addActionListener(new B1());
        b2.addActionListener(new B2());
        t1.addActionListener(new T1A());
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(t1);
        add(t2);
        add(t3);
    }
    class T1 implements DocumentListener {
        public void changedUpdate(DocumentEvent e) {}
        public void insertUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
            t3.setText("Text: "+ t1.getText());
        }
        public void removeUpdate(DocumentEvent e) {
            t2.setText(t1.getText());
        }
    }
    class T1A implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            t3.setText("t1 Action Event " + count++);
        }
```

```
        }
        class B1 implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                if(t1.getSelectedText() == null)
                    s = t1.getText();
                else
                    s = t1.getSelectedText();
                t1.setEditable(true);
            }
        }
        class B2 implements ActionListener {
            public void actionPerformed(ActionEvent e) {
                ucd.setUpperCase(false);
                t1.setText("Inserted by Button 2: " + s);
                ucd.setUpperCase(true);
                t1.setEditable(false);
            }
        }
        public static void main(String[] args) {
            run(new TextFields(), 375, 200);
        }
    }

    class UpperCaseDocument extends PlainDocument {
        private boolean upperCase = true;
        public void setUpperCase(boolean flag) {
            upperCase = flag;
        }
        public void
            insertString(int offset, String str, AttributeSet attSet)
            throws BadLocationException {
            if (upperCase) str = str.toUpperCase();
            super.insertString(offset, str, attSet);
        }
    } ///:~
```

Das Texteingabefeld `t3` hat die Aufgabe, das Auslösen des Ereignisbehandlers von Texteingabefeld `t1` anzuzeigen. Der Ereignisbehandler wird nur beim Betätigen der Eingabetaste ausgelöst.

[83] Das Texteingabefeld `t1` ist mit mehreren Ereignisbehandlern verknüpft. `T1` implementiert das Interface *`DocumentListener`* und reagiert auf Änderungen des „Dokumentes“ (hier Inhalt des Texteingabefeldes). `T1` kopiert automatisch den Inhalt von `t1` nach `t2`. Außerdem gehört das Dokument in `t1` dem von `PlainDocument` abgeleiteten Typ `UpperCaseDocument` an, der alle Buchstaben in Großbuchstaben umwandelt. It erkennt automatisch die Rücklöschtaste (*backspace*) und löscht die entsprechenden Zeichen, adjusting the caret und verhält sich auch sonst erwartungsgemäß.

Übungsaufgabe 13: (3) Ändern Sie das Beispiel *TextFields.java* so, daß die Buchstaben in `t2` die Groß-/Kleinschreibung beibehalten, mit der sie eingegeben wurden, statt automatisch in Großbuchstaben umgewandelt zu werden. ■

23.8.5 Rahmen

[84] Die Klasse `JComponent` definiert eine Methode namens `setBorder()`, mit der Sie verschiedene interessante Rahmen um jede sichtbare Komponente zeichnen können. Das folgende Beispiel führt eine Reihe der verschiedenen verfügbaren Rahmen vor. Die Methode `showBorder()` erzeugt eine `JPanel`-Komponente mit dem gewählten Rahmen. Die Methode ermittelt den gewünschten Rahmen

per RTTI (wobei die Packagezuordnung entfernt wird) und gibt den Namen des Rahmentyps als Beschriftung (`JLabel`) in der eingerahmten `JPanel`-Komponente aus:

```
//: gui/Borders.java
// Different Swing borders.
import javax.swing.*;
import javax.swing.border.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Borders extends JFrame {
    static JPanel showBorder(Border b) {
        JPanel jp = new JPanel();
        jp.setLayout(new BorderLayout());
        String nm = b.getClass().toString();
        nm = nm.substring(nm.lastIndexOf('.') + 1);
        jp.add(new JLabel(nm, JLabel.CENTER), BorderLayout.CENTER);
        jp.setBorder(b);
        return jp;
    }
    public Borders() {
        setLayout(new GridLayout(2,4));
        add(showBorder(new TitledBorder("Title")));
        add(showBorder(new EtchedBorder()));
        add(showBorder(new LineBorder(Color.BLUE)));
        add(showBorder(new MatteBorder(5,5,30,30,Color.GREEN)));
        add(showBorder(new BevelBorder(BevelBorder.RAISED)));
        add(showBorder(new SoftBevelBorder(BevelBorder.LOWERED)));
        add(showBorder(new CompoundBorder(
            new EtchedBorder(), new LineBorder(Color.RED))));
    }
    public static void main(String[] args) {
        run(new Borders(), 500, 300);
    }
} ///:~
```

Sie können eigene Rahmentypen entwickeln und zusammen mit Schaltflächen, Beschriftung und so weiter verwenden, kurz mit jeder von `JComponent` abgeleiteten Komponente.

23.8.6 Ein kleiner Editor

[85] Eine `JTextPane`-Komponente bietet mühelose Editierfunktionalität. Das folgende Beispiel zeigt einen einfachen Anwendungsfall, wobei der größte Teil der Funktionalität nicht zum Tragen kommt:

```
//: gui/TextPane.java
// The JTextPane control is a little editor.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

public class TextPane extends JFrame {
    private JButton b = new JButton("Add Text");
    private JTextPane tp = new JTextPane();
    private static Generator sg = new RandomGenerator.String(7);
    public TextPane() {
        b.addActionListener(new ActionListener() {
```

```
                    public void actionPerformed(ActionEvent e) {
                        for(int i = 1; i < 10; i++)
                            tp.setText(tp.getText() + sg.next() + ''\n'');
                    }
                });
            add(new JScrollPane(tp));
            add(BorderLayout.SOUTH, b);
        }
        public static void main(String[] args) {
            run(new TextPane(), 475, 425);
        }
    } ///:~
```

Die Schaltfläche setzt zufällig erzeugten „Text“ ein. Die `JTextPane`-Komponente ist dazu gedacht, Text vor Ort editieren zu können und hat daher keine `append()`-Methode. Im obigen Beispiel (zugegebenermaßen eine armseelige Nutzung der Fähigkeiten der `JTextPane`-Komponente) muß der Inhalt abgefragt, geändert und per `setText()` in den Darstellungsbereich zurückplatziert werden.

[86] Die Komponenten werden im Darstellungsbereich der `JFrame`-Komponete bezüglich des Standardlayoutmanagers `BorderLayout` angelegt. Die `JTextPane`-Komponente wird im Darstellungsbereich, verpackt in einer `JScrollPane`-Komponente angelegt, ohne einen Teilbereich anzugeben, füllt also den mittleren Teilbereich und wird bis zu dessen Eckpunkten hin ausgedehnt. Die Schaltfläche wird im Randbereich `SOUTH` angelegt, füllt also den unteren Rand des Darstellungsbereiches aus.

[87] Beachten Sie die eingebauten Eigenschaften und Fähigkeiten der `JTextPane`-Komponente, etwa den automatischen Zeilenumbruch. Die Dokumentation des Java Development Kits beschreibt auch die zahlreichen anderen Eigenschaften und Fähigkeiten.

Übungsaufgabe 14: (2) Ändern Sie das Beispiel *TextPane.java* so, daß es statt einer `JTextPane`- eine `JTextArea`-Komponente verwendet. ■

23.8.7 Ankreuzfelder

[88] Ein Ankreuzfeld erfaßt eine einzelne „Ja“/“Nein“-Auswahl und besteht aus einem kleinen Kästchen und einer Beschriftung. Das Kästchen enthält typischerweise ein kleines × (beziehungsweise eine andere Kennzeichnung dafür, daß das Feld „angekreuzt“ ist) oder ist leer, je nachdem ob die Auswahl selektiert wurde oder nicht.

[89] In der Regel wird eine `JCheckBox`-Komponente mit Hilfe eines Konstruktors erzeugt, der die Beschriftung als Argument erwartet. Sie können sowohl den Status des Ankreuzfeldes als auch seine Beschriftung nach der Objekterzeugung abfragen und ändern.

[90] Das Ankreuzen und Zurücksetzen einer `JCheckBox`-Komponente löst ein Ereignis aus, das Sie wie bei einer Schaltfläche abfangen können, nämlich mit einem Ereignisbehandler vom Typ *`ActionListener`*. Das folgende Beispiel zählt in einem Textbereich alle Ankreuzfelder auf, deren Zustand sich geändert hat:

```
//: gui/CheckBoxes.java
// Using JCheckBoxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class CheckBoxes extends JFrame {
    private JTextArea t = new JTextArea(6, 15);
```

```
    private JCheckBox
        cb1 = new JCheckBox("Check Box 1"),
        cb2 = new JCheckBox("Check Box 2"),
        cb3 = new JCheckBox("Check Box 3");
    public CheckBoxes() {
        cb1.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    trace("1", cb1);
                }
            });
        cb2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    trace("2", cb2);
                }
            });
        cb3.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    trace("3", cb3);
                }
            });
        setLayout(new FlowLayout());
        add(new JScrollPane(t));
        add(cb1);
        add(cb2);
        add(cb3);
    }
    private void trace(String b, JCheckBox cb) {
        if(cb.isSelected())
            t.append("Box " + b + " Set\n");
        else
            t.append("Box " + b + " Cleared\n");
    }
    public static void main(String[] args) {
        run(new CheckBoxes(), 200, 300);
    }
} ///:~
```

Die `trace()`-Methode übergibt die Beschriftung der angekreuzten beziehungsweise zurückgesetzten `JCheckBox`-Komponente und ihren aktuellen Zustand per `append()` an den Textbereich, so daß eine wachsende Liste der Namen von Ankreuzfeldern, zusammen mit ihrem Zustand angezeigt wird.

Übungsaufgabe 15: (5) Legen Sie in dem Programm aus Übungsaufgabe 5 (Seite 1015) ein Ankreuzfeld an, fangen Sie das Ereignis ab und setzen Sie einen anderen Text in das Texteingabefeld ein. ■

23.8.8 Radiobuttons

[91] Das Konzept der Radiobuttons in der GUI-Programmierung stammt von den vorelektronischen Autoradios mit mechanischen Knöpfen. Wird ein Knopf gedrückt, springen die anderen Knöpfe zurück. Radiobuttons gestatten also eine von mehreren Alternativen zu wählen.

[92] Das Funktionieren mehrerer `JRadioButton`-Komponenten als miteinander verknüpfte Elemente erfordert, daß sie zu einer Gruppe zusammengefaßt (`ButtonGroup`) werden. Eine `JRadioButton`-Komponente kann mit Hilfe eines zweiten Konstruktorargumentes auf `true` gesetzt werden. Falls mehr als eine Komponente auf `true` gesetzt ist, gilt die Einstellung nur für die letzte Komponente.

[93] Das folgende einfache Beispiel führt die Verwendung von Radiobuttons vor. Das Auswählen eines Radiobuttons wird über einen Ereignisbehandler vom Typ *ActionListener* angezeigt:

```
//: gui/RadioButtons.java
// Using JRadioButtons.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class RadioButtons extends JFrame {
    private JTextField t = new JTextField(15);
    private ButtonGroup g = new ButtonGroup();
    private JRadioButton
        rb1 = new JRadioButton("one", false),
        rb2 = new JRadioButton("two", false),
        rb3 = new JRadioButton("three", false);
    private ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                t.setText("Radio button " +
                    ((JRadioButton) e.getSource()).getText());
            }
        };
    public RadioButtons() {
        rb1.addActionListener(al);
        rb2.addActionListener(al);
        rb3.addActionListener(al);
        g.add(rb1); g.add(rb2); g.add(rb3);
        t.setEditable(false);
        setLayout(new FlowLayout());
        add(t);
        add(rb1);
        add(rb2);
        add(rb3);
    }
    public static void main(String[] args) {
        run(new RadioButtons(), 200, 125);
    }
} ///:~
```

Der Zustand der Gruppe wird in einem Texteingabefeld angezeigt. Als reines Anzeigefeld ohne Datenerfassung ist es nicht editierbar und damit eine Alternative zur `JLabel`-Komponente.

23.8.9 Drop-Down-Listen

[94] Ähnlich einer Gruppe von Radiobuttons verlangt auch eine Drop-Down-Liste, daß der Benutzer eine Möglichkeit aus einer Anzahl von Alternativen wählt. Die Darstellung ist aber kompakter und es ist einfacher, die Elemente in der Liste zu ändern, ohne den Benutzer zu überraschen. (Sie können auch Radiobuttons dynamisch ändern, was aber irritierend wirkt.)

[95] Die `JComboBox`-Komponente unterscheidet sich dardurch von der Windows-Combobox, daß sie nicht die Auswahl aus einer Liste *oder* Eingabe über die Tastatur gestattet. Sie können dieses Verhalten aber mittels `setEditable()` aktivieren. Eine `JComboBox`-Komponente gestattet, genau ein Element aus der Liste auszuwählen. Die `JComboBox`-Komponente im folgenden Beispiel ist mit vier Elementen vorinitialisiert. Über die Schaltfläche können weitere Elemente hinzugefügt werden:

```
//: gui/ComboBoxes.java
// Using drop-down lists.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class ComboBoxes extends JFrame {
    private String[] description = {
        ''Ebullient'', ''Obtuse'', ''Recalcitrant'', ''Brilliant'',
        ''Somnescent'', ''Timorous'', ''Florid'', ''Putrescent''
    };
    private JTextField t = new JTextField(15);
    private JComboBox c = new JComboBox();
    private JButton b = new JButton(''Add items'');
    private int count = 0;
    public ComboBoxes() {
        for(int i = 0; i < 4; i++)
            c.addItem(description[count++]);
        t.setEditable(false);
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    if(count < description.length)
                        c.addItem(description[count++]);
                }
            });
        c.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    t.setText(''index: ''+ c.getSelectedIndex() + '' '' +
                        ((JComboBox) e.getSource()).getSelectedItem());
                }
            });
        setLayout(new FlowLayout());
        add(t);
        add(c);
        add(b);
    }
    public static void main(String[] args) {
        run(new ComboBoxes(), 200, 175);
    }
} ///:~
```

Das Texteingabefeld zeigt den „selektierten Index“, das heißt die Position des gewählten Elementes in der Liste sowie den in der Drop-Down-Liste sichtbaren Text an.

23.8.10 Listboxen

[96] Listboxen (`JList`-Komponenten) unterscheiden sich grundlegend von Drop-Down-Listen (`JComboBox`-Komponenten), nicht nur in ihrer Erscheinungsform. Während eine Drop-Down-Liste beim Anklicken „aufklappt“, beansprucht eine Listbox stets eine feste Anzahl von Zeilen im Darstellungsbereich. Die Methode `getSelectedValues()` liefert ein `String`-Array mit den selektierten Elementen.

[97] Listboxen gestattet Mehrfachauswahl. Wenn Sie beim Selektieren die Steuerungstaste gedrückt halten (während Sie Elemente mit der Maus anklicken), können Sie so viele Elemente auswählen, als erforderlich. Wenn Sie ein Element selektieren und beim Anklicken eines zweiten Elementes die

Umschalttaste drücken, werden auch die Elemente zwischen den bereits selektierten Elementen ausgewählt. Indem Sie die Steuerungstaste gedrückt halten, können Sie einzelne Elemente deselektieren:

```
//: gui/List.java
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class List extends JFrame {
    private String[] flavors = {
        ''Chocolate'', ''Strawberry'', ''Vanilla Fudge Swirl'',
        ''Mint Chip'', ''Mocha Almond Fudge'', ''Rum Raisin'',
        ''Praline Cream'', ''Mud Pie''
    };
    private DefaultListModel lItems = new DefaultListModel();
    private JList lst = new JList(lItems);
    private JTextArea t = new JTextArea(flavors.length, 20);
    private JButton b = new JButton(''Add Item'');
    private ActionListener bl = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                if(count < flavors.length) {
                    lItems.add(0, flavors[count++]);
                } else {
                    // Disable, since there are no more
                    // flavors left to be added to the List
                    b.setEnabled(false);
                }
            }
        };
    private ListSelectionListener ll = new ListSelectionListener() {
            public void valueChanged(ListSelectionEvent e) {
                if(e.getValueIsAdjusting()) return;
                t.setText('''');
                for(Object item : lst.getSelectedValues())
                    t.append(item + ''\n'');
            }
        };
    private int count = 0;
    public List() {
        t.setEditable(false);
        setLayout(new FlowLayout());
        // Create Borders for components:
        Border brd = BorderFactory.createMatteBorder(1, 1, 2, 2, Color.BLACK);
        lst.setBorder(brd);
        t.setBorder(brd);
        // Add the first four items to the List
        for(int i = 0; i < 4; i++)
            lItems.addElement(flavors[count++]);
        add(t);
        add(lst);
        add(b);
        // Register event listeners
        lst.addListSelectionListener(ll);
        b.addActionListener(bl);
    }
```

```
    public static void main(String[] args) {
        run(new List(), 250, 375);
    }
} ///:~
```

Sie sehen, daß auch Listen eingerahmt werden können.

[98] Wenn Sie lediglich die Elemente eines `String`-Arrays als Elemente einer Listbox verwenden möchten, gibt es eine einfachere Lösung. Es genügt, das Array dem `JList`-Konstruktor zu übergeben, der die Listbox automatisch aufbaut. Der einzige Grund für die Verwendung des „Listenmodells“ (`DefaultListModel`) im obigen Beispiel besteht darin, daß die Liste nun zur Laufzeit geändert werden kann.

[99] Die `JList`-Komponente hat nicht automatisch Rollfunktionalität. Es genügt selbstverständlich, die `JList`-Komponente in einer `JScrollPane`-Komponente zu verpacken, die sich für Sie um alle Einzelheiten kümmert.

Übungsaufgabe 16: (5) Vereinfachen Sie das Beispiel *List.java*, indem Sie das `String`-Array direkt an den Konstruktor übergeben und das dynamische Hinzufügen von Listenelementen entfernen. ■

23.8.11 Karteikasten mit Reitern

[100] Die `JTabbedPane`-Komponente repräsentiert einen Karteikasten oder Ordner mit Reitern. Durch Anklicken eines Reiters erscheint die entsprechende Ebene im Vordergrund:

```
//: gui/TabbedPane1.java
// Demonstrates the Tabbed Pane.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class TabbedPane1 extends JFrame {
    private String[] flavors = {
        ''Chocolate'', ''Strawberry'', ''Vanilla Fudge Swirl'',
        ''Mint Chip'', ''Mocha Almond Fudge'', ''Rum Raisin'',
        ''Praline Cream'', ''Mud Pie''
    };
    private JTabbedPane tabs = new JTabbedPane();
    private JTextField txt = new JTextField(20);
    public TabbedPane1() {
        int i = 0;
        for(String flavor : flavors)
            tabs.addTab(flavors[i], new JButton(''Tabbed pane '' + i++));
        tabs.addChangeListener(new ChangeListener() {
                public void stateChanged(ChangeEvent e) {
                    txt.setText(''Tab selected: '' + tabs.getSelectedIndex());
                }
            });
        add(BorderLayout.SOUTH, txt);
        add(tabs);
    }
    public static void main(String[] args) {
        run(new TabbedPane1(), 400, 250);
    }
} ///:~
```

Die `JTabbedPane`-Komponente stapelt die Reiter automatisch übereinander, wenn nicht alle in eine Reihe passen, wie Sie durch Verändern der Fenstergröße nachvollziehen können.

23.8.12 Dialogfenster (Teil 1 von 2)

[101] Anwendungen mit graphischer Benutzeroberfläche verfügen häufig über eine standardisierte Menge von Dialogfenstern (*message boxes*), um dem Benutzer schnell eine Information zu präsentieren oder vom Benutzer abzufragen. In der Swing-Bibliothek sind Dialogfenster durch die Klasse `JOptionPane` vertreten. `JOptionPane` bietet viele verschiedene, teilweise recht anspruchsvolle Varianten. Am häufigsten werden Sie voraussichtlich die Varianten benötigen, die Sie mit Hilfe der statischen `JOptionPane`-Methoden `showMessageDialog()` und `showConfirmationDialog()` erzeugen können. Das folgende Beispiel zeigt einige der über die Klasse `JOptionPane` verfügbaren Dialogfenster:

```
//: gui/MessageBoxes.java
// Demonstrates JOptionPane.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class MessageBoxes extends JFrame {
    private JButton[] b = {
        new JButton("Alert"), new JButton("Yes/No"),
        new JButton("Color"), new JButton("Input"),
        new JButton("3 Vals")
    };
    private JTextField txt = new JTextField(15);
    private ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
                String id = ((JButton) e.getSource()).getText();
                if(id.equals("Alert"))
                    JOptionPane.showMessageDialog(null,
                        "There's a bug on you!", "Hey!",
                        JOptionPane.ERROR_MESSAGE);
                else if(id.equals("Yes/No"))
                    JOptionPane.showConfirmDialog(null,
                        "or no", "choose yes",
                        JOptionPane.YES_NO_OPTION);
                else if(id.equals("Color")) {
                    Object[] options = { "Red", "Green" };
                    int sel = JOptionPane.showOptionDialog(
                        null, "Choose a Color!", "Warning",
                        JOptionPane.DEFAULT_OPTION,
                        JOptionPane.WARNING_MESSAGE, null,
                        options, options[0]);
                    if(sel != JOptionPane.CLOSED_OPTION)
                        txt.setText("Color Selected: " + options[sel]);
                } else if(id.equals("Input")) {
                    String val = JOptionPane.showInputDialog(
                        "How many fingers do you see?");
                    txt.setText(val);
                } else if(id.equals("3 Vals")) {
                    Object[] selections = {"First", "Second", "Third"};
                    Object val = JOptionPane.showInputDialog(
                        null, "Choose one", "Input",
                        JOptionPane.INFORMATION_MESSAGE,
```

```
                        null, selections, selections[0]);
                    if(val != null)
                        txt.setText(val.toString());
                }
            }
        };
    public MessageBoxes() {
        setLayout(new FlowLayout());
        for(int i = 0; i < b.length; i++) {
            b[i].addActionListener(al);
            add(b[i]);
        }
        add(txt);
    }
    public static void main(String[] args) {
        run(new MessageBoxes(), 200, 200);
    }
} ///:~
```

Ich habe mich für die etwas riskante Lösung entschieden, die Beschriftungen der Schaltflächen abzufragen, um nur einen Ereignisbehandler schreiben zu müssen. Das Problem besteht darin, daß es leicht ist eine Beschriftung mit einem geringfügigen Fehler wiederzugeben, typischerweise bei der Groß-/Kleinschreibung, und derartige Fehler sind schwierig zu finden. Beachten Sie, daß die Methoden `showOptionDialog()` und `showInputDialog()` Objekte zurückgegeben, die den von Benutzer eingegebenen Wert enthalten.

Übungsaufgabe 17: (5) Schreiben Sie mit Hilfe der Klasse `SwingConsole` aus Unterabschnitt 23.2.1 eine kleine Anwendung. Lesen Sie die Komponente `JPasswordField` in der Dokumentation des Java Development Kits nach und legen Sie ein solches Feld in Ihrer Anwendung an. Zeigen die erfolgreiche Anmeldung mit Hilfe eines Dialogfensters an (`JOptionPane`). ■

Übungsaufgabe 18: (4) Ändern Sie das Beispiel *MessageBoxes.java* so, daß jede Schaltfläche einen individuellen Ereignisbehandler erhält (statt der Auswertung der Beschriftung). ■

23.8.13 Menüs

[102] Jede Komponente, die eine Menüleiste haben kann, darunter `JApplet`, `JFrame`, `JDialog` sowie die von diesen abgeleiteten Typen, besitzt eine `setJMenuBar()`-Methode, die ein `JMenuBar`-Objekt erwartet (eine Komponente kann höchstens eine Menüleiste haben). Anschließend legen Sie in der `JMenuBar`-Komponente (Menüleiste) `JMenu`-Komponenten (Menüs) und in diesen `JMenuItem`-Komponenten (Menüpunkte) an. Jede `JMenuItem`-Komponente kann einen Ereignisbehandler vom Typ *`ActionListener`* registrieren, der bei Wahl des Menüpunktes ausgelöst wird.

[103] Bei Java und Swing müssen Sie alle Menüs von Hand zusammensetzen. Hier ein sehr einfaches Beispiel:

```
//: gui/SimpleMenus.java
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class SimpleMenus extends JFrame {
    private JTextField t = new JTextField(15);
    private ActionListener al = new ActionListener() {
            public void actionPerformed(ActionEvent e) {
```

```
                t.setText(((JMenuItem)e.getSource()).getText());
            }
        };
    private JMenu[] menus = {
        new JMenu("Winken"), new JMenu("Blinken"),
        new JMenu("Nod")
    };
    private JMenuItem[] items = {
        new JMenuItem("Fee"), new JMenuItem("Fi"),
        new JMenuItem("Fo"), new JMenuItem("Zip"),
        new JMenuItem("Zap"), new JMenuItem("Zot"),
        new JMenuItem("Olly"), new JMenuItem("Oxen"),
        new JMenuItem("Free")
    };
    public SimpleMenus() {
        for(int i = 0; i < items.length; i++) {
            items[i].addActionListener(al);
            menus[i % 3].add(items[i]);
        }
        JMenuBar mb = new JMenuBar();
        for(JMenu jm : menus)
            mb.add(jm);
        setJMenuBar(mb);
        setLayout(new FlowLayout());
        add(t);
    }
    public static void main(String[] args) {
        run(new SimpleMenus(), 200, 150);
    }
} ///:~
```

Der Divisionsrestoperator in `i%3` verteilt die Menüpunkte auf die drei Menüs. Mit jedem Menüpunkt muß ein Ereignisbehandler vom Typ *`ActionListener`* verknüpft sein. Hier wird überall derselbe Ereignisbehandler verwendet, aber in der Regel erhält jeder Menüpunkt einen eigenen Ereignisbehandler.

[104] Die Klasse `JMenuItem` ist von `AbstractButton` abgeleitet, wodurch Menüpunkte ein schaltflächenähnliches Verhalten haben. Eine `JMenuItem`-Komponente repräsentiert einen Menüpunkt, der in ein Drop-Down-Menü eingesetzt werden kann. Es gibt drei von `JMenuItem` abgeleitete Komponenten: Die `JMenu`-Komponente (Menü) enthält andere `JMenuItem`-Komponenten, so daß Sie geschachtelte Menüs programmieren können. Die `JCheckBoxMenuItem`-Komponente zeichnet neben dem Menüpunkt ein Ankreuzfeld, um anzuzeigen, ob der Menüpunkt selektiert ist. Die `JRadioButtonMenuItem`-Komponente verhält sich wie die `JCheckBoxMenuItem`-Komponente, zeichnet aber einen Radiobutton neben den Menüpunkt.

[105] Das folgende Beispiel ist etwas anspruchsvoller und bedient sich erneut der Eissorten, um Menüs zu erzeugen. Das Beispiel führt die Schachtelung von Menüs, **mnemonischen Tastenkombinationen** (*keyboard mnemonics*) sowie `JCheckBoxMenuItem`-Komponenten vor und zeigt eine Möglichkeit, Menüs dynamisch zu ändern:

```
//: gui/Menus.java
// Submenus, check box menu items, swapping menus,
// mnemonics (shortcuts) and action commands.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;
```

```
public class Menus extends JFrame {
    private String[] flavors = {
        ''Chocolate'', ''Strawberry'', ''Vanilla Fudge Swirl'',
        ''Mint Chip'', ''Mocha Almond Fudge'', ''Rum Raisin'',
        ''Praline Cream'', ''Mud Pie''
    };
    private JTextField t = new JTextField(''No flavor'', 30);
    private JMenuBar mb1 = new JMenuBar();
    private JMenu
        f = new JMenu(''File''),
        m = new JMenu(''Flavors''),
        s = new JMenu(''Safety'');
    // Alternative approach:
    private JCheckBoxMenuItem[] safety = {
        new JCheckBoxMenuItem(''Guard''),
        new JCheckBoxMenuItem(''Hide'')
    };
    private JMenuItem[] file = { new JMenuItem(''Open'') };
    // A second menu bar to swap to:
    private JMenuBar mb2 = new JMenuBar();
    private JMenu fooBar = new JMenu(''fooBar'');
    private JMenuItem[] other = {
        // Adding a menu shortcut (mnemonic) is very
        // simple, but only JMenuItems can have them
        // in their constructors:
        new JMenuItem(''Foo'', KeyEvent.VK_F),
        new JMenuItem(''Bar'', KeyEvent.VK_A),
        // No shortcut:
        new JMenuItem(''Baz''),
    };
    private JButton b = new JButton(''Swap Menus'');
    class BL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuBar m = getJMenuBar();
            setJMenuBar(m == mb1 ? mb2 : mb1);
            validate(); // Refresh the frame
        }
    }
    class ML implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JMenuItem target = (JMenuItem)e.getSource();
            String actionCommand = target.getActionCommand();
            if(actionCommand.equals(''Open'')) {
                String s = t.getText();
                boolean chosen = false;
                for(String flavor : flavors)
                    if(s.equals(flavor))
                        chosen = true;
                if(!chosen)
                    t.setText(''Choose a flavor first!'');
                else
                    t.setText(''Opening '' + s + ''. Mmm, mm!'');
            }
        }
    }
    class FL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
```

```
            JMenuItem target = (JMenuItem)e.getSource();
            t.setText(target.getText());
        }
    }
    // Alternatively, you can create a different
    // class for each different MenuItem. Then you
    // don't have to figure out which one it is:
    class FooL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Foo selected");
        }
    }
    class BarL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Bar selected");
        }
    }
    class BazL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            t.setText("Baz selected");
        }
    }
    class CMIL implements ItemListener {
        public void itemStateChanged(ItemEvent e) {
            JCheckBoxMenuItem target = (JCheckBoxMenuItem) e.getSource();
            String actionCommand = target.getActionCommand();
            if(actionCommand.equals("Guard"))
                t.setText("Guard the Ice Cream! " +
                          "Guarding is " + target.getState());
            else if(actionCommand.equals("Hide"))
                t.setText("Hide the Ice Cream! " +
                          "Is it hidden? " + target.getState());
        }
    }
    public Menus() {
        ML ml = new ML();
        CMIL cmil = new CMIL();
        safety[0].setActionCommand("Guard");
        safety[0].setMnemonic(KeyEvent.VK_G);
        safety[0].addItemListener(cmil);
        safety[1].setActionCommand("Hide");
        safety[1].setMnemonic(KeyEvent.VK_H);
        safety[1].addItemListener(cmil);
        other[0].addActionListener(new FooL());
        other[1].addActionListener(new BarL());
        other[2].addActionListener(new BazL());
        FL fl = new FL();
        int n = 0;
        for(String flavor : flavors) {
            JMenuItem mi = new JMenuItem(flavor);
            mi.addActionListener(fl);
            m.add(mi);
            // Add separators at intervals:
            if((n++ + 1) % 3 == 0)
                m.addSeparator();
        }
        for(JCheckBoxMenuItem sfty : safety)
```

```
            s.add(sfty);
        s.setMnemonic(KeyEvent.VK_A);
        f.add(s);
        f.setMnemonic(KeyEvent.VK_F);
        for(int i = 0; i < file.length; i++) {
            file[i].addActionListener(ml);
            f.add(file[i]);
        }
        mb1.add(f);
        mb1.add(m);
        setJMenuBar(mb1);
        t.setEditable(false);
        add(t, BorderLayout.CENTER);
        // Set up the system for swapping menus:
        b.addActionListener(new BL());
        b.setMnemonic(KeyEvent.VK_S);
        add(b, BorderLayout.NORTH);
        for(JMenuItem oth : other)
            fooBar.add(oth);
        fooBar.setMnemonic(KeyEvent.VK_B);
        mb2.add(fooBar);
    }
    public static void main(String[] args) {
        run(new Menus(), 300, 200);
    }
} ///:~
```

In diesem Programm habe ich die Menüpunkte in Arrays gespeichert und anschließend in einer Schleife über jedes Array für jeden Menüpunkt die `add()`-Methode des zugehörigen Menüs aufgerufen. Dadurch wird das Hinzufügen oder Entfernen eines Menüpunktes weniger lästig.

[106] Das Programm erzeugt zwei Menüleisten (`JMenuBar`-Komponenten), um zu zeigen, daß Menüleisten zur Laufzeit ausgetauscht werden können. Sie können nachvollziehen, wie eine Menüleiste aus einzelnen Menüs (`JMenu`-Komponenten) aufgebaut ist, wobei jedes Menü wiederum aus Menüpunkten (`JMenuItem`- oder `JCheckBoxMenuItem`-Komponenten) oder Untermenüs (`JMenu`-Komponenten) besteht. Eine zusammengesetzte Menüleiste kann per `setJMenuBar()` im aktuellen Programm installiert werden. Beachten Sie, daß der Ereignisbehandler der Schaltfläche mit Hilfe der `getJMenuBar()`-Methode die gegenwärtig installierte Menüleiste abfragt und anschließend die andere platziert.

[107] Beachten Sie beim Vergleich der Beschriftung des Menüpunktes mit der Zeichenkette „Open“, daß die korrekte Groß-/Kleinschreibung wesentlich ist, Java aber keinen Fehler signalisiert, wenn es keine Übereinstimmung mit dem Vergleichswert „Open“ gibt. Diese Art von Zeichenkettenvergleichen ist die Ursache vieler Programmierfehler.

[108] Der binäre Status einer `JCheckBoxMenuItem`- oder `JRadioButtonMenuItem`-Komponente wird automatisch erfaßt. Der Ereignisbehandler der `JCheckBoxMenuItem`-Komponente zeigt zwei verschiedene Möglichkeiten, um abzufragen, welches Ankreuzfeld selektiert wurde: Zeichenkettenvergleich (weniger sicher, kommt aber hin und wieder vor) sowie die Statusabfrage beim Ereignisobjekt (hier vom Typ `ItemEvent`). Die `getState()`-Methode dient dazu, den Status der Komponente zu ermitteln. Insbesondere können Sie einer `JCheckBoxMenuItem`-Komponente per `setState()` einen Status zuweisen.

[109] Die Ereignistypen für Menüs sind ein wenig inkonsistent und können Verwirrung verursachen: `JMenuItem`-Komponenten registrieren Ereignisbehandler vom Typ *`ActionListener`*, während `JCheckBoxMenuItem`-Komponenten Ereignisbehandler vom Typ *`ItemListener`* haben. `JMenu`-Komponenten unterstützten den Typ *`ActionListener`*, wobei Ereignisbehandler bei Menüs

selbst üblicherweise wenig nützen. In der Regel verknüpfen Sie Komponenten vom Typ `JMenuItem`, `JCheckBoxMenuItem` oder `JRadioButtonMenuItem` mit einem Ereignisbehandler, auch wenn das obige Beispiel Behandler der Typen *`ItemListener`* und *`ActionListener`* (anscheinend wahllos) bei allen möglichen Komponenten registriert.

[110] Swing unterstützt mnemonische Tastaturkombinationen, so daß Sie alle von `AbstractButton` abgeleiteten Komponenten (also Schaltflächen, Menüpunkte und so weiter) anstelle der Maus über die Tastatur steuern können. Eine solche Tastaturkombination ist leicht zu definieren. Die Klasse `JMenuItem` hat einen Konstruktor, der die gewünschte Taste als zweites Argument erwartet. Die meisten von `AbstractButton` abgeleiteten Komponenten haben allerdings keinen solchen Konstruktor. Die allgemeinere Lösung besteht somit darin, die `setMnemonic()`-Methode aufzurufen. Im obigen Beispiel werden der Schaltfläche und einigen Menüpunkten mnemomische Tastaturkombinationen zugewiesen und erscheinen automatisch (als unterstrichene Buchstaben) auf den Komponenten.

[111] Das Beispiel zeigt außerdem die Verwendung der Methode `setActionCommand()`. Dabei fällt auf, daß das °**Aktionskommando** (*action command*) in beiden Fällen exakt mit der Beschriftung der Menükomponente übereinstimmt. Warum wird statt der alternativen Beschriftung nicht die ursprüngliche verwendet? Die Antwort lautet „Internationalisierung“. Wenn Sie dieses Programm an eine andere Sprache anpassen, möchten Sie nach Möglichkeit nur die Beschriftung des Menüs ändern, nicht aber den Quelltext (wodurch zweifellos neue Fehler entstehen würden). Indem Sie die `setActionCommand()`-Methode aufrufen bleibt das Aktionskommando unverändert, die Beschriftung der Schaltfläche läßt sich aber dennoch anpassen. Das Programm arbeitet mit Aktionskommando, ist also von einer Änderung der Beschriftung des Menüpunktes nicht betroffen. Beachten Sie, daß nicht alle Menükomponenten in diesem Programm auf ihr Aktionskommando hin untersucht werden. Das Aktionskommando der nicht untersuchten Komponenten ist nicht gesetzt.

[112] Die Arbeit findet größtenteils in den Ereignisbehandlern statt. `BL` bewirkt das Austauschen der Menüleisten. `ML` ermittelt die Quelle des Ereignisses durch Abfragen des `ActionEvent`-Objektes mit Typumwandlung nach `JMenuItem` und Extraktion des Aktionskommando, welches anschließend in einer verschachtelten `if`-Struktur ausgewertet wird.

[113] Der `FL`-Ereignisbehandler ist zwar einfach aufgebaut, kümmert sich aber um alle Eissorten im Menü „Flavor“. Diese Vorgehensweise bietet sich an, wenn Ihre Logik einfach genug ist. Im allgemeinen werden Sie sich dagegen für die mittels `FooL`, `BarL` und `BazL` implementierte Variante entscheiden, wobei jeder Ereignisbehandler mit nur einer einzigen Menükomponente verknüpft ist, so daß keine Logik erforderlich ist, um die Quelle des Ereignisses zu ermitteln und Sie genau wissen, wer den Ereignisbehandler aufgerufen hat. Trotz des Überflusses an Klassen, enthält jede Klasse weniger Anweisungen und der Ansatz ist insgesamt sicherer.

[114] Sie sehen, daß die Programmierung eines Menüs schnell langwierig und unübersichtlich wird. Dies ist wiederum eine Situation, in sich der Einsatz eines GUI-Builders lohnt. Ein guter GUI-Builder kümmert sich auch um die Pflege der Menüs.

Übungsaufgabe 19: (3) Ändern Sie das Beispiel *Menus.java* so, daß Radiobuttons anstelle der Ankreuzfelder in den Menüs vorkommen. ■

Übungsaufgabe 20: (6) Schreiben Sie ein Programm, das eine Textdatei in einzelne Wörter zerlegt. Verwenden Sie diese Wörter als Beschriftungen von Menüs und Untermenüs. ■

23.8.14 Kontextmenüs

[115] Eine `JPopupMenu`-Komponente (Kontextmenü) läßt sich am leichtesten implementieren, indem Sie eine innere Klasse von `MouseAdapter` ableiten und jeder Komponente, die ein Kontextmenü erhalten soll, ein Objekt dieser inneren Klasse hinzufügen:

```
//: gui/Popup.java
// Creating popup menus with Swing.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class Popup extends JFrame {
    private JPopupMenu popup = new JPopupMenu();
    private JTextField t = new JTextField(10);
    public Popup() {
        setLayout(new FlowLayout());
        add(t);
        ActionListener al = new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    t.setText(((JMenuItem)e.getSource()).getText());
                }
            };
        JMenuItem m = new JMenuItem("Hither");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Yon");
        m.addActionListener(al);
        popup.add(m);
        m = new JMenuItem("Afar");
        m.addActionListener(al);
        popup.add(m);
        popup.addSeparator();
        m = new JMenuItem("Stay Here");
        m.addActionListener(al);
        popup.add(m);
        PopupListener pl = new PopupListener();
        addMouseListener(pl);
        t.addMouseListener(pl);
    }
    class PopupListener extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            maybeShowPopup(e);
        }
        public void mouseReleased(MouseEvent e) {
            maybeShowPopup(e);
        }
        private void maybeShowPopup(MouseEvent e) {
            if(e.isPopupTrigger())
                popup.show(e.getComponent(), e.getX(), e.getY());
        }
    }
    public static void main(String[] args) {
        run(new Popup(), 300, 200);
    }
} ///:~
```

Jedem Menüpunkt wird derselbe Ereignisbehandler zugewiesen. Es fragt die Beschriftung des Menüpunktes ab zu setzt sie in das Texteingabefeld ein.

23.8.15 Graphikausgabe

[116] Eine gute GUI-Bibliothek wie Swing gestattet einfaches Konstruieren von Graphiken. Bei entsprechenden Beispielen zeigt sich stets das Problem, daß die Berechnungen für die Positionierung der einzelnen Teile typischerweise erheblich komplizierter sind, als die Aufrufe der eigentlichen Zeichenmethoden. Außerdem sind die Berechnungen und Zeichenmethoden häufig eng miteinander verknüpft, wodurch die Programmierschnittstelle komplizierter wirkt, als sie eigentlich ist.

[117] Wir verwenden zur Darstellung von Daten auf dem Bildschirm der Einfachheit halber die Rückgabewerte der eingebauten Methode `Math.sin()`, also die Werte der mathematischen Sinusfunktion. Am unteren Rand des Darstellungsbereichs befindet sich ein Schieberegler (`JSlider`-Komponente), über den die Anzahl der der anzeigten Sinusschwingungen eingestellt werden kann. Dieser Mechanismus soll einerseits die Aufgabe etwas interessanter gestalten und andererseits ein weiteres Beispiel für den einfachen Umgang mit den Komponenten der Swing-Bibliothek liefern. Wenn Sie die Abmessungen des Fensters ändern, paßt sich die angezeigte Kurve der neuen Fenstergröße an.

[118] Obwohl grundsätzlich jede von `JComponent` abgeleitete Komponente may be painted und daher als Medium zum Zeichnen verwendet werden kann, leiten Sie typischerweise eine individuelle Komponente von `JPanel` ab. Die einzige Methode, die Sie überschreiben müssen, ist `paintComponent()`. Diese Methode wird stets aufgerufen, wenn die zugehörige Komponente neu gezeichnet werden muß (in der Regel müssen Sie sich hierüber keine Gedanken machen, da Swing alles notwendige veranlaßt). Swing übergibt der `paintComponent()`-Methode beim Aufruf ein Objekt vom Typ `Graphics`, mit dessen Hilfe Sie im Darstellungsbereich zeichnen oder paint können.

[119] Im folgenden Beispiel steckt alle „Intelligenz“ hinsichtlich des Zeichnens in der Klasse `SineWave`, die sowohl das Programm als auch den Schieberegler konfiguriert. Die `setCycles()`-Methode in der Klasse `SineWave` gestattet einem anderen Objekt, in diesem Fall dem Schieberegler, die Anzahl der Zyklen einzustellen:

```
//: gui/SineWave.java
// Drawing with Swing, using a JSlider.
import javax.swing.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

class SineDraw extends JPanel {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw() { setCycles(5); }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        int maxWidth = getWidth();
        double hstep = (double) maxWidth / (double) points;
        int maxHeight = getHeight();
        pts = new int[points];
        for(int i = 0; i < points; i++)
            pts[i] = (int) (sines[i] * maxHeight/2 * .95 + maxHeight/2);
        g.setColor(Color.RED);
```

```
            for(int i = 1; i < points; i++) {
                int x1 = (int) ((i - 1) * hstep);
                int x2 = (int) (i * hstep);
                int y1 = pts[i-1];
                int y2 = pts[i];
                g.drawLine(x1, y1, x2, y2);
            }
        }
        public void setCycles(int newCycles) {
            cycles = newCycles;
            points = SCALEFACTOR * cycles * 2;
            sines = new double[points];
            for(int i = 0; i < points; i++) {
                double radians = (Math.PI / SCALEFACTOR) * i;
                sines[i] = Math.sin(radians);
            }
            repaint();
        }
    }

    public class SineWave extends JFrame {
        private SineDraw sines = new SineDraw();
        private JSlider adjustCycles = new JSlider(1, 30, 5);
        public SineWave() {
            add(sines);
            adjustCycles.addChangeListener(new ChangeListener() {
                    public void stateChanged(ChangeEvent e) {
                        sines.setCycles(((JSlider) e.getSource()).getValue());
                    }
                });
            add(BorderLayout.SOUTH, adjustCycles);
        }
        public static void main(String[] args) {
            run(new SineWave(), 700, 400);
        }
    } ///:~
```

[120] Alle Felder und Arrays dienen der Berechnung der Punkte der Sinuskurve: `cycles` gibt an, wieviele Sinusschwingungen gezeichnet werden sollen, `points` gibt an, wieviele Punkte in x-Richtung gezeichnet werden sollen, `sines` enthält die `double`-Werte der `sin()`-Funktion und `pts` beinhaltet die ganzzahligen y-Koordinaten der im Darstellungsbereich der `JPanel`-Komponente gezeichneten Kurve. Die `setCycles()`-Methode bewertet die Felder `cycles` und `points` und initialisiert das `sines`-Array mit Funktionswerten. Durch Aufrufen der `repaint()`-Methode erzwingt `setCycles()` die Verarbeitung der Methode `paintComponent()`, wodurch der verbleibende Teil der Berechnung und das Neuzeichnen der Kurve veranlaßt werden.

[121] Die erste Anweisung beim Überschreiben der `paintComponent()`-Methode ist der Aufruf der Basisklassenversion. Danach steht es Ihnen frei, alles zu tun, was Sie möchten. Das bedeutet in der Regel, daß Sie die Methoden des `Graphics`-Objektes aufrufen (siehe API-Dokumentation der Klasse `Graphics`), um im Darstellungsbereich der `JPanel`-Komponente Pixel zu platzieren und zu färben. Wie Sie selbst nachvollziehen können, bezieht sich der größte Teil der Anweisungen auf die Berechnungen. Die beiden einzigen Methoden, die tatsächlich die Bildschirmausgabe betreffen, sind `setColor()` und `drawLine()`. Wenn Sie ein eigenes Programm geschrieben haben, um Daten zu visualisieren, haben Sie wahrscheinlich ähnliche Erfahrungen gemacht: Sie verbrauchen die meiste Zeit damit, wie Sie die Daten zur Visualisierung vorbereiten müssen, während das tatsächliche Zeichnen relativ einfach ist.

[122] Als ich dieses Programm entwickelte, investierte ich die meiste Zeit mit der Visualisierung der Sinuskurve. Nachdem dieser Teil funktionierte, kam mir der Gedanke, es sei doch nett, die Anzahl der Schwingungen dynamisch verändern zu können. Aufgrund meiner Programmiererfahrung in anderen Sprachen bin ich bei solchen Versuchen zurückhaltend, aber es stellte sich heraus, daß dies der einfachste Teil der Arbeit war. Ich legte also im `JFrame`-Container eine `JSlider`-Komponente an (die Argumente des Konstruktors sind der linke beziehungsweise rechte Grenzwert und der Startwert). Ein Blick in die Dokumentation des Java Development Kits zeigte, daß es nur einen einzigen Ereignisbehandler gibt, nämlich `addChangeListener()`, welcher ausgelöst wird, wenn der Schieberegler weit genug bewegt wird, um einen neuen Wert zu erzeugen. Die einzige im Interface *`ChangeListener`* deklarierte Methode heißt `stateChanged()` und verfügt in ihrem Methodenkörper über ein `ChangeEvent`-Objekt, mit dessen Hilfe ich die Quelle des Ereignisses erreichen und den neuen Wert abfragen konnte. Durch einen Aufruf der `SineWave`-Methode `setCycles()` konnte der neue Wert übernommen und die Kurve im Darstellungsbereich neu gezeichnet werden.

[123] Die meisten Swing-Probleme lassen sich über eine ähnliche Vorgehensweise lösen. Die Lösung wird Ihnen leicht fallen, auch wenn Sie eine der beteiligten Komponenten noch nie zuvor verwendet haben.

[124] Für kompliziertere Visualisierungsprobleme gibt es anspruchsvollere Alternativen, darunter JavaBeans von Drittanbietern und die Java-2D-API. Diese Bibliotheken gehen über den Rahmen dieses Buches hinaus. Ziehen Sie sie aber in Betracht, wenn das Programmieren Ihrer Visualisierung gar zu beschwerlich wird.

Übungsaufgabe 21: (5) Ändern Sie das Beispiel *SineWave.java*, indem Sie die Klasse `SineWave` in eine JavaBean mit Abfrage- und Änderungsmethoden umwandeln. ■

Übungsaufgabe 2: (7) Schreiben Sie mit Hilfe der Klasse `SwingConsole` aus Unterabschnitt 23.2.1 eine kleine Anwendung mit drei Schiebereglern (`JSlider`-Komponenten) für die Farbwerte rot, grün und blau eines `java.awt.Color`-Objektes. Der Darstellungsbereich (`JPanel`-Komponente) zeigt die durch die RGB-Werte definierte Farbe an. Legen Sie auch drei editierbare Texteingabefelder an, die die aktuellen RGB-Werte enthalten. ■

Übungsaufgabe 23: (8) Schreiben Sie, ausgehend vom Beispiel *SineWave.java*, ein Programm, das ein rotierendes Quadrat auf dem Bildschirm anzeigt. Ein Schieberegler steuert die Rotationsgeschwindigkeit und ein zweiter die Größe des Quadrats. ■

Übungsaufgabe 24: (7) Erinnern Sie sich an das Spielzeug [*(sketching box)*] mit zwei Knöpfen, wobei der eine die senkrechte und der andere die waagerechte Bewegung des Zeichenkopfes steuert? Schreiben Sie, ausgehend vom Beispiel *SineWave.java*, ein solches Programm. Verwenden Sie Schieberegler anstelle der Knöpfe und legen Sie eine Schaltfläche an, um den Inhalt der Zeichenfläche zu löschen. ■

Übungsaufgabe 25: (8) Schreiben Sie, ausgehend vom Beispiel *SineWave.java*, ein Programm, das eine animierte Sinuskurve zeichnet, die wie beim Oszilloskop über den sichtbaren Fensterbereich hinausreicht. Steuern Sie die Animation per `java.util.Timer` und die Geschwindigkeit der Animation per Schieberegler. ■

Übungsaufgabe 26: (5) Ändern Sie Übungsaufgabe 25, so daß das Programm mehrere Darstellungsbereiche mit Sinuskurven anzeigt. Die Anzahl der Darstellungsbereiche wird per Kommandozeilenargument übergeben. ■

Übungsaufgabe 27: (5) Ändern Sie Übungsaufgabe 25, so daß die Animation per `javax.swing.Timer` gesteuert wird. Beachten Sie die Unterschiede zwischen `javax.swing.Timer` und `java.util.Timer`. ■

Übungsaufgabe 28: (7) Schreiben Sie eine Klasse, die einen Spielwürfel repräsentiert (nur eine gewöhnliche Klasse ohne graphische Benutzeroberfläche). Erzeugen Sie fünf Objekte dieser Klasse und „werfen“ Sie sie wiederholt. Visualisieren Sie die Summe der Augenzahlen während Sie die fünf Würfel immer wieder werfen. ■

23.8.16 Dialogfenster (Teil 2 von 2)

[125] Ein Dialogfenster ist ein plötzlich aus einem anderen Fenster auftauchendes Hilfsfenster und soll eine bestimmte Aufgabe erfüllen, ohne das ursprüngliche Fenster mit diesen Einzelheiten zu „verunreinigen“. Dialogfenster treten bei fenstergesteuerten Anwendungen häufig auf.

[126] Um ein Dialogfenster anzulegen, leiten Sie eine neue Klasse von `JDialog` ab, einem weiteren Untertyp von `Window`, wie etwa `JFrame`. Eine `JDialog`-Komponente besitzt einen Layoutmanager (Voreinstellung ist `BorderLayout`) und kann verschiedene Ereignisbehandler registrieren. Zunächst ein einfaches Beispiel:

```
//: gui/Dialogs.java
// Creating and using Dialog Boxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

class MyDialog extends JDialog {
    public MyDialog(JFrame parent) {
        super(parent, ''My dialog'', true);
        setLayout(new FlowLayout());
        add(new JLabel(''Here is my dialog''));
        JButton ok = new JButton(''OK'');
        ok.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    dispose(); // Closes the dialog
                }
            });
        add(ok);
        setSize(150,125);
    }
}

public class Dialogs extends JFrame {
    private JButton b1 = new JButton(''Dialog Box'');
    private MyDialog dlg = new MyDialog(null);
    public Dialogs() {
        b1.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    dlg.setVisible(true);
                }
            });
        add(b1);
    }
    public static void main(String[] args) {
        run(new Dialogs(), 125, 75);
    }
} ///:~
```

Nach dem Erzeugen einer `JDialog`-Komponente, muß deren `setVisible(true)`-Methode aufgerufen werden, um die Komponente anzuzeigen und zu aktivieren. Beim Schließen des Dialogfensters

müssen Sie die von ihm beanspruchten Resourcen per `dispose()` wieder freigeben.

[127] Das folgende Beispiel ist etwas komplexer. Das Dialogfenster beinhaltet ein Gitter (Layoutmanager `GridLayout`) spezieller Schaltflächen vom Typ `ToeButton` (abgeleitet von `JDialog`). Eine `ToeButton`-Schaltfläche zeichnet einen Rahmen um sich selbst und beinhaltet je nach Zustand ein leeres Feld, ein Kreuz (×) oder einen Kreis (○). Die Schaltfläche ist zu Beginn leer und wird je nach dem, welcher Spieler an der Reihe ist, auf ein Kreuz oder einen Kreis umgeschaltet. Der Ereignisbehandler der Schaltfläche gestattet aber auch da Hin- und Herschalten zwischen den beiden Symbolen. Außerdem kann das Diagramm im Dialogfenster durch Eingabe entsprechender Werte im Hauptfenster der Anwendung beliebige Zeilen- und Spaltenzahlen annehmen:

```
//: gui/TicTacToe.java
// Dialog boxes and creating your own components.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class TicTacToe extends JFrame {
    private JTextField
        rows = new JTextField("3"),
        cols = new JTextField("3");
    private enum State { BLANK, XX, OO }
    static class ToeDialog extends JDialog {
        private State turn = State.XX; // Start with x's turn
        ToeDialog(int cellsWide, int cellsHigh) {
            setTitle("The game itself");
            setLayout(new GridLayout(cellsWide, cellsHigh));
            for(int i = 0; i < cellsWide * cellsHigh; i++)
                add(new ToeButton());
            setSize(cellsWide * 50, cellsHigh * 50);
            setDefaultCloseOperation(DISPOSE_ON_CLOSE);
        }
        class ToeButton extends JPanel {
            private State state = State.BLANK;
            public ToeButton() { addMouseListener(new ML()); }
            public void paintComponent(Graphics g) {
                super.paintComponent(g);
                int
                    x1 = 0, y1 = 0,
                    x2 = getSize().width - 1,
                    y2 = getSize().height - 1;
                g.drawRect(x1, y1, x2, y2);
                x1 = x2/4;
                y1 = y2/4;
                int wide = x2/2, high = y2/2;
                if(state == State.XX) {
                    g.drawLine(x1, y1, x1 + wide, y1 + high);
                    g.drawLine(x1, y1 + high, x1 + wide, y1);
                }
                if(state == State.OO)
                    g.drawOval(x1, y1, x1 + wide/2, y1 + high/2);
            }
            class ML extends MouseAdapter {
                public void mousePressed(MouseEvent e) {
                    if(state == State.BLANK) {
                        state = turn;
```

```
                        turn = (turn == State.XX ? State.OO : State.XX);
                    }
                    else
                        state = (state == State.XX ? State.OO : State.XX);
                    repaint();
                }
            }
        }
    }
    class BL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JDialog d = new ToeDialog(new Integer(rows.getText()),
                                      new Integer(cols.getText()));
            d.setVisible(true);
        }
    }
    public TicTacToe() {
        JPanel p = new JPanel();
        p.setLayout(new GridLayout(2,2));
        p.add(new JLabel("Rows", JLabel.CENTER));
        p.add(rows);
        p.add(new JLabel("Columns", JLabel.CENTER));
        p.add(cols);
        add(p, BorderLayout.NORTH);
        JButton b = new JButton("go");
        b.addActionListener(new BL());
        add(b, BorderLayout.SOUTH);
    }
    public static void main(String[] args) {
        run(new TicTacToe(), 200, 200);
    }
} ///:~
```

[128] Da statische Komponenten nur in der äußersten Klassenebene erlaubt sind, dürfen innere Klassen keine statischen Klassen (auch „geschachtelte Klassen“) beinhalten.

[129] Die `paintComponent()`-Methode zeichnet den Rahmen um die Schaltfläche und setzt das Kreuz beziehungsweise den Kreis ein. Der Methodenkörper enthält zwar viel ermüdende Rechnerei, ist aber geradlinig.

[130] Ein Mausklick wird vom *`MouseListener`*-Ereignisbehandler abgefangen, der zunächst prüft, ob die Schaltfläche bereits ein Kreuz oder einen Kreis enthält. Ist die Schaltfläche noch unberührt, so ermittelt der Ereignisbehandler über das Elternfenster, welcher Spieler an der Reihe ist und legt dadurch den Zustand der `ToeButton`-Schaltfläche fest. Mit Hilfe der Zugriffsmöglichkeiten einer inneren Klasse reicht die `ToeButton`-Schaltfläche zurück in das Elternfenster, um dessen `turn`-Feld umzuschalten. Zeigt die Schaltfläche bereits ein Kreuz oder einen Kreis, so wird die Anzeige auf das komplementäre Symbol umgeschaltet. Beachten Sie die elegante Anwendung des ternären Operators (siehe Kapitel 4). Die `ToeButton`-Schaltfläche wird nach jeder Zustandsänderung neu gezeichnet.

[131] Der Konstruktor der Klasse `ToeDialog` ist einfach zu verstehen: Er legt bindet den Darstellungsbereich der Schaltfläche an den Layoutmanager `GridLayout` mit der angeforderten Anzahl von `ToeButton`-Schaltflächen mit einer Kantenlänge von 50 Pixel pro Seite.

[132] Die Klasse `TicTacToe` konfiguriert die ganze Anwendung mittels zweier Texteingabefelder (Eingabe der Zeilen- und Spaltenanzahl des Gitters) sowie der „go“-Schaltfläche samt Ereignisbehandler vom Typ *`ActionListener`*. Beim Betätigen der Schaltfläche müssen die Inhalte der beiden Texteingabefelder abgefragt und mit Hilfe des `Integer`-Konstruktors für `String`-Argumente in `int`-Werte

umgewandelt werden.

23.8.17 Dateiauswahlfenster

[133] Manche Betriebssysteme haben eine Reihe spezieller eingebauter Dialogfenster für Einstellungen wie Zeichensätze, Farben oder Drucker. Nahezu alle Betriebssysteme mit graphischer Benutzeroberfläche unterstützen das Öffnen und Sichern von Dateien. Die Komponente `JFileChooser` kapselt diese Funktionalität, um ihre Nutzung zu vereinfachen.

[134] Das folgende Beispiel führt zwei Varianten der `JFileChooser`-Komponente vor, nämlich ein Dateiauswahlfenster zum Öffnen und eines zum Sichern von Dateien. Die Anweisungen sind Ihnen mittlerweile größtenteils vertraut. Die interessanten Dinge geschehen in den Ereignisbehandlern der beiden Schaltflächen:

```
//: gui/FileChooserTest.java
// Demonstration of File dialog boxes.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class FileChooserTest extends JFrame {
    private JTextField
        fileName = new JTextField(),
        dir = new JTextField();
    private JButton
        open = new JButton("Open"),
        save = new JButton("Save");
    public FileChooserTest() {
        JPanel p = new JPanel();
        open.addActionListener(new OpenL());
        p.add(open);
        save.addActionListener(new SaveL());
        p.add(save);
        add(p, BorderLayout.SOUTH);
        dir.setEditable(false);
        fileName.setEditable(false);
        p = new JPanel();
        p.setLayout(new GridLayout(2,1));
        p.add(fileName);
        p.add(dir);
        add(p, BorderLayout.NORTH);
    }
    class OpenL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Demonstrate "Open" dialog:
            int rVal = c.showOpenDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                fileName.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                fileName.setText("You pressed cancel");
                dir.setText("");
            }
```

```
        }
    }
    class SaveL implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            JFileChooser c = new JFileChooser();
            // Demonstrate ''Save'' dialog:
            int rVal = c.showSaveDialog(FileChooserTest.this);
            if(rVal == JFileChooser.APPROVE_OPTION) {
                fileName.setText(c.getSelectedFile().getName());
                dir.setText(c.getCurrentDirectory().toString());
            }
            if(rVal == JFileChooser.CANCEL_OPTION) {
                fileName.setText(''You pressed cancel'');
                dir.setText('''');
            }
        }
    }
    public static void main(String[] args) {
        run(new FileChooserTest(), 250, 150);
    }
} ///:~
```

Die `JFileChooser`-Komponente hat viele Varianten, darunter solche mit Filtern, um die Menge der wählbaren Dateien einzuschränken.

[135] Die Methode `showOpenDialog()` öffnet ein „Datei öffnen"-Dialogfenster, die Methode `showSaveDialog()` dagegen ein „Datei sichern"-Dialogfenster. Beide Methoden kehren erst nach dem Schließen des Dialogfensters zurück. Das `JFileChooser`-Objekt existiert auch nach dem Schließen des Dialogfensters, so daß Sie Informationen abfragen können. Die Methoden `getSelectedFile()` und `getCurrentDirectory()` sind zwei Beispiele für das Abfragen des Ergebnisses der Operation. Liefert eine dieser beiden Methoden `null`, so hat der Benutzer den Dialog abgebrochen.

Übungsaufgabe 29: (3) Lesen Sie in der Dokumentation des Java Development Kit die Komponente `JColorChooser` nach. Schreiben Sie ein Programm mit einer Schaltfläche, das ein Dialogfenster mit einer `JColorChooser`-Komponente präsentiert. ■

23.8.18 Beschriftung von Komponenten mit HTML-Anweisungen

[136] Jede Komponente mit Beschriftung akzeptiert auch HTML-Anweisungen, die nach den Regeln von HTML formatiert werden. Sie können also Swing-Komponenten schick beschriften, zum Beispiel:

```
//: gui/HTMLButton.java
// Putting HTML text on Swing components.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import static net.mindview.util.SwingConsole.*;

public class HTMLButton extends JFrame {
    private JButton b = new JButton(''<html><b><font size=+2>'' +
                                    ''<center>Hello!<br><i>Press me now!'');
    public HTMLButton() {
        b.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    add(new JLabel(''<html>'' + ''<i><font size=+4>Kapow!''));
                    // Force a re-layout to include the new label:
```

```
                    validate();
                }
            });
        setLayout(new FlowLayout());
        add(b);
    }
    public static void main(String[] args) {
        run(new HTMLButton(), 200, 500);
    }
} ///:~
```

Die Beschriftung muß mit dem Starttag `<html>` beginnen, woran sich die übrigen HTML-Tags anschließen. Beachten Sie, daß Sie nicht gezwungen sind, die üblichen schließenden Tags zu verwenden.

[137] Der Ereignisbehandler ist vom Typ *`ActionListener`* und legt eine weitere Beschriftung an, die ebenfalls aus HTML-Anweisungen besteht. Die Beschriftung wird allerdings nicht beim Erzeugen des `JLabel`-Objektes angelegt, sondern Sie müssen die `validate()`-Methode des Containers aufrufen, um eine Neuanordnung der Komponenten zu erzwingen, wobei auch die neue Beschriftung sichtbar wird.

[138] Weitere Komponenten, die Beschriftungen mit HTML-Anweisungen erlauben sind `JTabbedPane`, `JMenuItem`, `JToolTip`, `JRadioButton` und `JCheckBox`.

Übungsaufgabe 30: (3) Schreiben Sie ein Programm, das je eine `JTabbedPane`-, `JMenuItem`-, `JToolTip`-, `JRadioButton`- und `JCheckBox`-Komponente mit HTML-Anweisungen als Beschriftungen erzeugt. ■

23.8.19 Schieberegler und Fortschrittsbalken

[139] Eine `JSlider`-Komponente (Schieberegler, siehe Beispiel *SineWave.java*) gestattet dem Benutzer durch Vor- und Rückwärtsbewegen einer Markierung einen Wert einzustellen. Dies ist in Situationen wie etwa dem Einstellen der Lautstärke intuitiv. Eine `JProgressBar`-Komponente (Fortschrittsbalken) visualisiert eine Information in relativer Form zwischen den Endpunkten „Voll“ und „Leer“, vermittelt dem Benutzer also einen sichtbaren Eindruck. Mein Lieblingsbeispiel ist, Schieberegler und Fortschrittsbalken miteinander zu verbinden, so daß sich der Fortschrittsbalken bei einer Änderung des Schiebereglers entsprechend mitbewegt. Das folgende Beispiel führt außerdem die Klasse `ProgressMonitor` vor, ein funktionell besser ausgestattetes Dialogfenster:

```
//: gui/Progress.java
// Using sliders, progress bars and progress monitors.
import javax.swing.*;
import javax.swing.border.*;
import javax.swing.event.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class Progress extends JFrame {
    private JProgressBar pb = new JProgressBar();
    private ProgressMonitor pm =
        new ProgressMonitor(this, ''Monitoring Progress'', ''Test'', 0, 100);
    private JSlider sb = new JSlider(JSlider.HORIZONTAL, 0, 100, 60);
    public Progress() {
        setLayout(new GridLayout(2,1));
        add(pb);
        pm.setProgress(0);
        pm.setMillisToPopup(1000);
```

```
            sb.setValue(0);
            sb.setPaintTicks(true);
            sb.setMajorTickSpacing(20);
            sb.setMinorTickSpacing(5);
            sb.setBorder(new TitledBorder(''Slide Me''));
            pb.setModel(sb.getModel()); // Share model
            add(sb);
            sb.addChangeListener(new ChangeListener() {
                    public void stateChanged(ChangeEvent e) {
                        pm.setProgress(sb.getValue());
                    }
                });
        }
        public static void main(String[] args) {
            run(new Progress(), 300, 200);
        }
    } ///:~
```

[140] Der Schlüssel für die Verknüpfung zwischen Schieberegler und Fortschrittsbalken ist das gemeinsam verwendete Datenmodell:

```
pb.setModel(sb.getModel());
```

Natürlich ließe sich die Steuerung der Abhängigkeit auch mittels zweier Ereignisbehandler bewerkstelligen, aber das Datenmodell ist in einfachen Fällen die weniger komplizierte Lösung. Das `ProgressMonitor`-Objekt verfügt über kein eigenes Datenmodell, so daß hier ein Ereignisbehandler erforderlich ist. Beachten Sie, daß sich der „Fortschrittsmonitor“ nur vorwärts bewegt und sein Fenster geschlossen wird, wenn die Anzeige das rechte Ende erreicht. Die `JProgressBar`-Komponente ist unkompliziert, während `JSlider` über eine Reihe von Optionen verfügt, darunter seine Orientierung (horizontal oder vertikal) und Skalenstriche. Beachten Sie, wie leicht sich ein Rahmen mit Beschriftung anlegen läßt.

Übungsaufgabe 31: (8) Schreiben Sie einen „asymptotischen Fortschrittsanzeiger“, der vor dem Erreichen den Endpunktes immer langsamer wird. Implementieren Sie ein zufälliges unberechenbares Verhalten, so daß die Anzeige scheinbar periodisch schneller wird. ■

Übungsaufgabe 32: (6) Ändern Sie das Beispiel *Progress.java* so, daß die Verknüpfung zwischen Schieberegler und Fortschrittsbalken nicht über ein gemeinsames Datenmodell, sondern über Ereignisbehandler bewerkstelligt wird. ■

23.8.20 Wählen eines Look-and-Feels

[141] Das sogenannte „Austauschbare Look-and-Feel“ gestattet Ihrem Programm, das Look-and-Feel verschiedener Betriebsysteme zu emulieren. Das Look-and-Feel kann sogar dynamisch ausgetauscht werden, also zur Laufzeit des Programms. Im allgemeinen wollen Sie aber entweder ein plattformübergreifendes Look-and-Feel (also das „Metal“-Look-and-Feel von Swing) oder die für das unterliegende Betriebssystem spezifische Einstellung (in den meisten Fällen die beste Wahl, um den Benutzer nicht zu verwirren). Die Anweisung zur Auswahl des Look-and-Feels ist einfach, muß aber vor dem Anlegen der ersten visuellen Komponente erteilt werden, da die Komponenten basierend auf dem aktuellen Look-and-Feel gezeichnet werden und nicht ohne weiteres neu gezeichnet werden, wenn Sie die Einstellung während der Laufzeit des Programms ändern (dieser Eingriff ist kompliziert und tritt nur selten auf; ich verweise daher auf Swing-spezifische Bücher).

[142] Wenn Sie das plattformübergreifende für Swing-Anwendungen charakteristische „Metal“-Look-and-Feel verwenden möchten, müssen Sie nichts weiter tun, denn es ist voreingestellt. Wenn Sie

statt dessen die Einstellung des unterliegenden Betriebssystems übernehmen wollen,[7] so genügt es, typischerweise zu Beginn der `main()`-Methode, zumindest aber vor dem Anlegen der ersten Komponente, die folgenden Zeilen einzusetzen:

```
Manager.setLookAndFeet(
    UIManager.getSystemLookAndFeelClassName());
} catch(Exception e) {
    throw new RuntimeException(e);
}
```

Sie können die `catch`-Klausel auch leer lassen, da die Klasse `UIManager` per Voreinstellung das plattformunabhängige Look-and-Feel einstellt, falls Ihre Versuche, eine der Alternativen zu wählen, scheitern. Andererseits kann sich die Ausnahme bei der Fehlersuche als nützlich erweisen, so daß Sie eventuell doch einige Anweisungen in die `catch`-Klausel einsetzen.

[143] Das folgende Programm wählt das Look-and-Feel anhand eines Kommandozeilenargumentes und zeigt eine Auswahl von Komponenten unter der gewählten Einstellung:

```
//: gui/LookAndFeel.java
// Selecting different looks & feels.
// {Args: motif}
import javax.swing.*;
import java.awt.*;
import static net.mindview.util.SwingConsole.*;

public class LookAndFeel extends JFrame {
    private String[] choices =
        ''Eeny Meeny Minnie Mickey Moe Larry Curly''.split('' '');
    private Component[] samples = {
        new JButton(''JButton''),
        new JTextField(''JTextField''),
        new JLabel(''JLabel''),
        new JCheckBox(''JCheckBox''),
        new JRadioButton(''Radio''),
        new JComboBox(choices),
        new JList(choices),
    };
    public LookAndFeel() {
        super(''Look And Feel'');
        setLayout(new FlowLayout());
        for(Component component : samples)
            add(component);
    }
    private static void usageError() {
        System.out.println(''Usage:LookAndFeel [cross|system|motif]'');
        System.exit(1);
    }
    public static void main(String[] args) {
        if(args.length == 0) usageError();
        if(args[0].equals(''cross'')) {
            try {
                UIManager.setLookAndFeel(
                    UIManager.getCrossPlatformLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else if(args[0].equals(''system'')) {
```

[7] Man kann darüber streiten, ob das „Metal“-Look-and-Feel von Swing dem Betriebssystem gerecht wird.

```
            try {
                UIManager.setLookAndFeel(
                    UIManager.getSystemLookAndFeelClassName());
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else if(args[0].equals("motif")) {
            try {
                UIManager.setLookAndFeel(
                    "com.sun.java." + "swing.plaf.motif.MotifLookAndFeel");
            } catch(Exception e) {
                e.printStackTrace();
            }
        } else usageError();
        // Note the look & feel must be set before
        // any components are created.
        run(new LookAndFeel(), 300, 300);
    }
} ///:~
```

Eine Möglichkeit, das Look-and-Feel zu wählen, ist, den Namen explizit anzugeben (`MotifLookAndFeel`). Dieses und das voreingestellte „Metal"-Look-and-Feel sind die einzigen beiden Einstellungen, die Sie unter jedem Betriebsystem verwenden können. Es gibt zwar Look-and-Feels für Windows und Macintosh, aber diese können nur unter der entsprechenden Plattform verwendet werden. (Sie erhalten die Namen dieser Look-and-Feels, wenn Sie unter dem jeweiligen Betriebsystem die `getSystemLookAndFeelClassName()`-Methode aufrufen.)

[144] Sie können auch ein eigenes Look-and-Feel entwickeln, etwa wenn Sie eine Bibliothek für ein Unternehmen schreiben, das einen individuellen Auftritt wünscht. Dies ist allerdings eine große Aufgabe und geht weit über den Rahmen dieses Buches hinaus (Sie werden feststellen, daß dieses Thema den Rahmen vieler einschlägiger Swing-Bücher sprengt).

23.8.21 Bäume, Tabellen und Zwischenablage

[145] In den Online-Anhängen zu diesem Buch unter der Webadresse *http://www.mindview.net* finden Sie eine kurze Einführung und einige Beispiel zu diesen Themen.

23.9 Java Web Start und das Java Network Launching Protocol

[146] Ein Applet kann aus Sicherheitsgründen signiert werden. Dieser Vorgang wird im Online-Anhang zu diesem Kapitel unter der Webadresse *http://www.mindview.net* gezeigt. Signierte Applets sind mächtig und können effektiv an die Stelle von Anwendungen treten, müssen aber in einem Webbrowser ausgeführt werden. Der Betrieb eines Applets erfordert die zusätzlichen Unkosten eines auf dem Clientrechner laufenden Browsers. Außerdem ist die Benutzerschnittstelle eingeschränkt und häufig visuell verwirrend, da der Webbrowser eigene Menüs und Werkzeugleisten hat, die über dem Applet liegen.[8]

[147] Das Java Network Launch Protocol (JNLP) löst das Problem, ohne die Vorteile von Applets zu opfern. JNLP gestattet, eine selbständige Java-Anwendung auf den Clientrechner herunterzuladen und dort zu installieren. Die Java-Anwendung kann von der Kommandozeile, über ein Desktop-Icon oder den Anwendungsmanager Ihrer JNLP-Implementierung aufgerufen werden. Die

[8] Dieser Abschnitt wurde von Jeremy Meyer entwickelt.

Java-Anwendung kann sogar von der Website aus gestartet werden, von der sie urprünglich heruntergeladen wurde.

[148] Eine JNLP-Anwendung kann zur Laufzeit dynamisch Resourcen aus dem Internet herunterladen sowie automatisch ihre Version prüfen, wenn der Benutzer mit dem Internet verbunden ist, hat also alle Vorteile eines Applets zusammen mit den Vorteilen einer selbständigen Anwendung.

[149] JNLP-Anwendungen müssen, wie Applets, vom Clientsystem mit einiger Vorsicht behandelt werden. Daher unterliegen JNLP-Anwendungen denselben Sicherbeschränkungen wie Applets (Sandkasten). JNLP-Anwendungen können, wie Applets, als signierte *.jar* Dateien deployt werden, wobei der Benutzer die Wahl hat, der Partei, welche die Unterschrift geleistet hat, zu vertrauen oder nicht. Im Gegensatz zu Applets, haben unsignierte *.jar* Dateien über Dienste in der JNLP-API noch immer Zugriff auf bestimmte Resourcen des Clientsystems. Der Benutzer muß diese Anfragen während der Programmausführung bestätigen.

[150] JNLP ist ein Protokoll, keine Implementierung, das heißt Sie brauchen eine Implementierung, um das Protokoll benutzen zu können. Java Web Start (JAWS) ist die frei verfügbare offizielle Referenzimplementierung von Sun Microsystems und gehört zur Distribution der SE 5. Wenn Sie Java Web Start für Ihre eigenen Entwicklungsprojekte verwenden wollen, müssen Sie gewährleisten, daß die *.jar* Datei (*javaws.jar*) im Klassenpfad steht. Die einfachste Lösung ist, die Archivdatei *javaws.jar* im Verzeichnis *jre/lib* Ihrer normalen Java-Installation zu deponieren. Wenn Sie Ihre JNLP-Anwendung per Webserver deployen, müssen Sie dafür sorgen, daß Ihr Server den MIME-Typ `application/x-java-jnlp-file` erkennt. Falls Sie die aktuelle Version des Tomcat-Servers verwenden (*http://jakarta.apache.org/tomcat*), ist diese Einstellung vorkonfiguriert. Konsultieren Sie die Betriebsanleitung für Benutzer Ihrs Servers.

[151] Es ist nicht schwierig, eine JNLP-Anwendung zu erzeugen. Sie schreiben eine gewöhnliche Anwendung und archivieren sie in einer *.jar* Datei. Anschließend stellen Sie eine Startdatei zur Verfügung, eine schlichte XML-Datei, die dem Clientsystem alle erforderlichen Informationen liefert, um die Anwendung herunterladen und installieren zu können. Wenn Sie sich entscheiden, Ihre *.jar* Datei nicht zu signieren, müssen Sie zum Zugriff auf die einzelnen Typen von Resourcen im System des Benutzers die von der JNLP-API angebotenen Dienste verwenden.

[152] Das folgende Beispiel ist eine Variante von *FileChooserTest.java*, welches die JNLP-Dienste nutzt, um ein Dateiauswahlfenster zu öffnen, damit die Klasse als JNLP-Anwendung mittels einer unsignierten *.jar* Datei deployt werden kann:

```
//: gui/jnlp/JnlpFileChooser.java
// Opening files on a local machine with JNLP.
// {Requires: javax.jnlp.FileOpenService;
// You must have javaws.jar in your classpath}
// To create the jnlpfilechooser.jar file, do this:
// cd ..
// cd ..
// jar cvf gui/jnlp/jnlpfilechooser.jar gui/jnlp/*.class
package gui.jnlp;
import javax.jnlp.*;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;

public class JnlpFileChooser extends JFrame {
     private JTextField fileName = new JTextField();
     private JButton
         open = new JButton("Open"),
```

```
    save = new JButton(''Save'');
private JEditorPane ep = new JEditorPane();
private JScrollPane jsp = new JScrollPane();
private FileContents fileContents;
public JnlpFileChooser() {
    JPanel p = new JPanel();
    open.addActionListener(new OpenL());
    p.add(open);
    save.addActionListener(new SaveL());
    p.add(save);
    jsp.getViewport().add(ep);
    add(jsp, BorderLayout.CENTER);
    add(p, BorderLayout.SOUTH);
    fileName.setEditable(false);
    p = new JPanel();
    p.setLayout(new GridLayout(2,1));
    p.add(fileName);
    add(p, BorderLayout.NORTH);
    ep.setContentType(''text'');
    save.setEnabled(false);
}
class OpenL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileOpenService fs = null;
        try {
            fs = (FileOpenService)
                ServiceManager.lookup(''javax.jnlp.FileOpenService'');
        } catch(UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
        if(fs != null) {
            try {
                fileContents =
                    fs.openFileDialog(''.'', new String[] {''txt'', ''*''});
                if(fileContents == null)
                    return;
                fileName.setText(fileContents.getName());
                ep.read(fileContents.getInputStream(), null);
            } catch(Exception exc) {
                throw new RuntimeException(exc);
            }
            save.setEnabled(true);
        }
    }
}
class SaveL implements ActionListener {
    public void actionPerformed(ActionEvent e) {
        FileSaveService fs = null;
        try {
            fs = (FileSaveService)
                ServiceManager.lookup(''javax.jnlp.FileSaveService'');
        } catch(UnavailableServiceException use) {
            throw new RuntimeException(use);
        }
        if(fs != null) {
            try {
                fileContents =
```

```
                        fs.saveFileDialog(''.'', new String[] {''txt''},
                              new ByteArrayInputStream(ep.getText().getBytes()),
                              fileContents.getName());
                    if(fileContents == null)
                        return;
                    fileName.setText(fileContents.getName());
                } catch(Exception exc) {
                    throw new RuntimeException(exc);
                }
            }
        }
    }
    public static void main(String[] args) {
        JnlpFileChooser fc = new JnlpFileChooser();
        fc.setSize(400, 300);
        fc.setVisible(true);
    }
} ///:~
```

[153] Beachten Sie, daß die Klassen `FileOpenService` und `FileSaveService` aus dem Package `javax.jnlp` importiert werden und nirgends im Quelltext ein Dateiauswahlfenster (`JFileChooser`-Komponente) direkt referenziert wird. Die beiden Dienste müssen über die `ServiceManager`-Methode `lookup()` angefordert werden und die Resourcen des Clientsystems können nur mit Hilfe der von `lookup()` zurückgegebenen Objekte bedient werden. Die Dateien im Dateisystem des Clientsystems werden mit Hilfe des zur JNLP-Bibliothek gehörigen Interfaces *`FileContent`* gelesen beziehungsweise geschrieben. Jeder Versuch, beispielsweise über ein `File`- oder `FileReader`-Objekt auf eine dieser Resourcen zuzugreifen würde, wie bei einem Zugriffsversuch über ein unsigniertes Applet, eine Ausnahme vom Typ `SecurityException` hervorrufen. Wenn Sie Klassen verwenden und nicht auf die Interfaces der JNLP-Dienste beschränkt sein wollen, müssen Sie die *.jar* Datei signieren.

[154] Das auskommentierte `jar`-Kommando im Beispiel *JnlpFileChooser.java* erzeugt die benötigte *.jar* Datei. Es folgt eine Startdatei für das obige Beispiel:

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<jnlp spec = ''1.0+''
  codebase=''file:C:/AAA-TIJ4/code/gui/jnlp''
  href=''filechooser.jnlp''>
  <information>
    <title>FileChooser demo application</title>
    <vendor>Mindview Inc.</vendor>
    <description>
      Jnlp File chooser Application
    </description>
    <description kind=''short''>
      Demonstrates opening, reading and writing a text file
    </description>
    <icon href=''mindview.gif''/>
    <offline-allowed/>
  </information>
  <resources>
    <j2se version=''1.3+'' href=''http://java.sun.com/products/autodl/j2se''/>
    <jar href=''jnlpfilechooser.jar'' download=''eager''/>
  </resources>
  <application-desc main-class=''gui.jnlp.JnlpFileChooser''/>
</jnlp>
```

Sie finden diese Startdatei unter dem Namen *filechooser.jnlp* in der Quelltextdistribution zu diesem Buch, die Sie unter der Webadresse *http://www.mindview.net* herunterladen können, in dem Verzeichnis, das auch die *.jar* Datei enthält. Die erste und letzte Zeile wurden bei dieser Version entfernt. Die Startdatei ist eine XML-Datei und hat das Wurzelelement `<jnlp>`. Die wenigen Unterelemente sind größtenteils selbsterklärend.

[155] Das `spec`-Attribut des `<jnlp>`-Elementes teilt dem Clientsystem mit, mit welcher Version des JNLP-Protokolls die Anwendung betrieben werden kann. Das `codebase`-Attribut verweist auf die URL unter der sich die Startdatei und die übrigen Resourcen befinden. Im obigen Fall referenziert `codebase` ein Verzeichnis auf dem lokalen Rechner (eine gute Lösung, um die Anwendung zu testen). Beachten Sie, daß Sie dieses Attribut so bewerten müssen, daß er auf das entsprechende Verzeichnis ihres Rechners verweist, damit das Programm geladen werden kann. Das `href`-Attribut gibt den Namen der Startdatei an.

[156] Das Element `<information>` hat mehrere Unterelemente und liefert Auskünfte über die Anwendung. Diese Informationen werden von der Administratorkonsole von Java Web Start oder einem äquivalenten Werkzeug ausgewertet, welche die JNLP-Anwendung installiert und dem Benutzer den Aufruf von der Kommandozeile, per Desktop-Icon und so weiter gestattet.

[157] Das Element `<resources>` hat eine ähnliche Aufgabe, wie das `<applet>`-Element von HTML. Das Unterelement `<j2se>` gibt die zum Betrieb der Anwendung erforderliche Version der Standard Edition an, das Unterelement `<jar>` die *.jar* Datei, in der sich die zum Aufrufen der Anwendung benötigte Klasse befindet. Das `download`-Attribut des `<jar>`-Elementes kann mit `eager` oder `lazy` bewertet werden und teilt der JNLP-Implementierung mit, ob das gesamte Archiv vor dem Aufrufen der Anwendung heruntergeladen werden muß.

[158] Das Element `<application-desc>` gibt der JNLP-Implementierung an, welche Klasse ausführbar, das heißt der Einstiegspunkt der *.jar* Datei ist.

[159] Das `<security>`-Element ist ein weiteres nützliches Unterelement von `<jnlp>`, wird aber im obigen Beispiel nicht verwendet. Es hat die folgende Struktur:

```
<security>
    <all-permissions/>
<security/>
```

Das `<security>`-Element tritt auf, wenn Sie Ihre Anwendung als signierte *.jar* Datei deployen. Im obigen Beispiel ist es nicht erforderlich, da alle lokalen Resourcen über die JNLP-Dienste erreicht werden.

[160] Das JNLP-Format kennt noch einige weitere Tags, deren Einzelheiten Sie in der Spezifikation unter der Webadresse *http://java.sun.com/products/javawebstart/downloadspec.html* finden.

[161] Sie brauchen nun noch eine HTML-Seite mit einem Hyperlink auf die *.jnlp* Datei, um die Anwendung herunterladen zu können, zum Beispiel (ohne die erste und die letzte Zeile):

```
<html>
Follow the instructions in JnlpFileChooser.java to
build jnlpfilechooser.jar, then:
<a href=''filechooser.jnlp''>click here</a>
</html>
```

Nachdem Sie die Anwendung einmal heruntergeladen haben, können Sie sie mit Hilfe der Administratorkonsole konfigurieren. Wenn Sie Java Web Start unter Windows betreiben, werden Sie beim zweiten Start der Anwendung gefragt, ob Sie auf dem Desktop eine Verknüpfung zu Ihrer Anwendung ablegen möchten. Dieses Verhalten ist konfigurierbar.

[162] Das obige Beispiel nutzt nur zwei JNLP-Dienste, wobei die gegenwärtige Version sieben Dienste zur Verfügung stellt. Jeder Dienst erfüllt eine bestimmte Aufgabe, etwa Drucken oder Ausschneiden und Einfügen (*cut and paste*) in die Zwischenablage. Unter der Webadresse *http://java.sun.com* finden Sie weitere Informationen.

23.10 Threads und Swing

[163] Wenn Sie mit der Swing-Bibliothek programmieren, sind Threads im Spiel. In Unterabschnitt 23.2 haben Sie gelernt, daß Sie sämtliche Methodenaufrufe bei Swing-Komponenten über die statische `SwingUtilities`-Methode `invokeLater()` an den Ereignisbehandlerthread übertragen sollen (Seite 1010f). Die Tatsache, daß Sie keinen eigenen Thread erzeugen müssen, bedeutet allerdings auch, daß Sie von Threadangelegenheiten überrascht werden können. Sie müssen sich vergegenwärtigen, daß der Ereignisbehandlerthread von Swing stets vorhanden ist und Swing-Ereignisse behandelt, indem er sie einzeln aus der Ereigniswarteschlange entnimmt und verarbeitet. Indem Sie an der Ereignisbehandlerthread denken, unterstützen Sie, daß Ihre Anwendung keine Verklemmungen oder Wettlaufsituationen hervorruft.

[164] Dieser Abschnitt ist Theadangelegenheiten gewidmet, die beim Arbeiten mit Swing auftreten können.

23.10.1 Aufgaben mit langer Verarbeitungsdauer

[165] Ein der häufigsten grundlegenden Fehler in der GUI-Programmierung mit Swing besteht darin, daß Sie den Ereignisbehandlerthread irrtümlich dazu verwenden, eine langwierige Aufgabe zu verarbeiten, zum Beispiel:

```
//: gui/LongRunningTask.java
// A badly designed program.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

public class LongRunningTask extends JFrame {
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task");
    public LongRunningTask() {
        b1.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent evt) {
                    try {
                        TimeUnit.SECONDS.sleep(3);
                    } catch(InterruptedException e) {
                        System.out.println("Task interrupted");
                        return;
                    }
                    System.out.println("Task completed");
                }
            });
        b2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent evt) {
                    // Interrupt yourself?
```

```
                    Thread.currentThread().interrupt();
                }
            });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
    }
    public static void main(String[] args) {
        run(new LongRunningTask(), 200, 150);
    }
} ///:~
```

Wenn Sie die „Start Long Running Task“-Schaltfläche (`b1`) betätigen, beansprucht die Verarbeitung der langwierige Aufgabe den Ereignisbehandlerthread. Die gedrückte Schaltfläche wird nicht in ihren Ruhezustand zurückgestellt, weil der Ereignisbehandlerthread, der normalerweise für das Neuzeichnen des Bildschirms verantwortlich ist, besetzt ist. Sie haben keine Möglichkeit, in die Programmausführung einzugreifen, auch nicht mit Hilfe der „End Long Running Task“-Schaltfläche (`b2`), da das Programm erst reagiert, wenn die von der ersten Schaltfläche gestartete zeitaufwändige Aufgabe verarbeitet ist und der Ereignisbehandlerthread wieder zur Verfügung steht. Der Ereignisbehandler der `b2`-Schaltfläche ist ein gescheiterter Versuch, das Problem durch Unterbrechen des Ereignisbehandlerthreads zu lösen.

[166] Die Lösung besteht selbstverständlich darin, Aufgabe mit langer Laufzeit mit Hilfe separater Threads zu verarbeiten. Im folgenden Beispiel wird ein „single Threadexekutor“ (`newSingleThreadExecutor()`) verwendet, der noch nicht erledigte Aufgaben in einer Warteschlange speichert und stets höchstens eine Aufgabe verarbeitet:

```
//: gui/InterruptableLongRunningTask.java
// Long-running tasks in threads.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import static net.mindview.util.SwingConsole.*;

class Task implements Runnable {
    private static int counter = 0;
    private final int id = counter++;
    public void run() {
        System.out.println(this + '' started'');
        try {
            TimeUnit.SECONDS.sleep(3);
        } catch(InterruptedException e) {
            System.out.println(this + '' interrupted'');
            return;
        }
        System.out.println(this + '' completed'');
    }
    public String toString() { return ''Task '' + id; }
    public long id() { return id; }
};

public class InterruptableLongRunningTask extends JFrame {
    private JButton
        b1 = new JButton(''Start Long Running Task''),
        b2 = new JButton(''End Long Running Task'');
    ExecutorService executor = Executors.newSingleThreadExecutor();
    public InterruptableLongRunningTask() {
```

```
            b1.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        Task task = new Task();
                        executor.execute(task);
                        System.out.println(task + '' added to the queue'');
                    }
                });
            b2.addActionListener(new ActionListener() {
                    public void actionPerformed(ActionEvent e) {
                        executor.shutdownNow(); // Heavy-handed
                    }
                });
            setLayout(new FlowLayout());
            add(b1);
            add(b2);
        }
        public static void main(String[] args) {
            run(new InterruptableLongRunningTask(), 200, 150);
        }
    } ///:~
```

[167] Diese Lösung ist zwar besser als das vorige Beispiel (*LongRunningTask.java*), aber das Betätigen der Schaltfläche „End Long Running Task“ bewirkt, daß die `shutdown()`-Methode des Exekutors aufgerufen, der Exekutor also heruntergefahren wird. Wenn Sie dem Exekutor mehr als eine Aufgabe zur Verarbeitung übergeben haben, ruft das Herunterfahren eine Ausnahme hervor. Das Programm verliert also durch Betätigen der Schaltfläche „End Long Running Task“ seine Funktionstüchtigkeit. Wir brauchen eine Möglichkeit, die Verarbeitung der aktuellen (und eventueller schwebender) Aufgaben zu beenden, ohne den Exekutor mit allen Aufgaben außer Betrieb zu nehmen. Der seit der SE 5 vorhandene *`Callable`*/*`Future`*-Mechanismus (siehe Unterabschnitt 22.4.3) gestattet das gezielte Abbrechen einzelner Aufgaben. Wir definieren eine neue Klasse namens `TaskManager`, welche Paare vom Typ `TaskItem` enthält, bestehend aus einem *`Callable`*-Objekt (der Aufgabe) und dem nach der Verarbeitung zurückgegebenen *`Future`*-Objekt. Das *`Callable`*/*`Future`*-Paar ist erforderlich, um die ursprüngliche Aufgabe verfolgen zu können und zusätzliche Informationen zur Verfügung zu haben, die über das *`Future`*-Objekt nicht erreichbar sind. Zunächst die Paar-Klasse `TaskItem`:

```
//: net/mindview/util/TaskItem.java
// A Future and the Callable that produced it.
package net.mindview.util;
import java.util.concurrent.*;

public class TaskItem<R,C extends Callable<R>> {
    public final Future<R> future;
    public final C task;
    public TaskItem(Future<R> future, C task) {
        this.future = future;
        this.task = task;
    }
} ///:~
```

Die Klasse *`Future`* aus dem Package `java.util.concurrent` hat keinen Zugriff auf die ursprüngliche Aufgabe, da das entsprechende *`Callable`*-Objekt nach der Verarbeitung nicht mehr zwingend existieren muß, wenn Sie Verbindung mit dem *`Future`*-Objekt aufnehmen, um die Ergebnisse abzufragen. Durch die neue Klasse `TaskItem` erzwingen wir, daß das *`Callable`*-Objekt erhalten bleibt.

[168] Die Klasse `TaskManager` gehört zum Package `net.mindview.util`, steht also als allgemeine Hilfsklasse zur Verfügung:

```
//: net/mindview/util/TaskManager.java
// Managing and executing a queue of tasks.
package net.mindview.util;
import java.util.concurrent.*;
import java.util.*;

public class TaskManager<R,C extends Callable<R>>
    extends ArrayList<TaskItem<R,C>> {
    private ExecutorService exec = Executors.newSingleThreadExecutor();
    public void add(C task) {
        add(new TaskItem<R,C>(exec.submit(task),task));
    }
    public List<R> getResults() {
        Iterator<TaskItem<R,C>> items = iterator();
        List<R> results = new ArrayList<R>();
        while(items.hasNext()) {
            TaskItem<R,C> item = items.next();
            if(item.future.isDone()) {
                try {
                    results.add(item.future.get());
                } catch(Exception e) {
                    throw new RuntimeException(e);
                }
                items.remove();
            }
        }
        return results;
    }
    public List<String> purge() {
        Iterator<TaskItem<R,C>> items = iterator();
        List<String> results = new ArrayList<String>();
        while(items.hasNext()) {
            TaskItem<R,C> item = items.next();
            // Leave completed tasks for results reporting:
            if(!item.future.isDone()) {
                results.add("Cancelling " + item.task);
                item.future.cancel(true); // May interrupt
                items.remove();
            }
        }
        return results;
    }
} ///:~
```

Die Klasse `TaskManager` ist von `ArrayList` abgeleitet und enthält `TaskItem`-Elemente. `TaskManager` hat einen „single Threadexekutor“ und wenn Sie die `add()`-Methode mit einem *`Callable`*-Objekt aufrufen, übergibt `add()` das *`Callable`*-Objekt dem Exekutor und speichert das zurückerhaltene *`Future`*-Objekt zusammen mit der ursprünglichen Aufgabe. Falls Sie das *`Callable`*-Objekt brauchen, haben Sie nun eine Referenz zur Verfügung. Beispielsweise ruft `purge()` die `toString()`-Methode des *`Callable`*-Objektes auf.

[169] Nun setzen wir `TaskManager` ein, um die Aufgaben mit langer Laufzeit in unserem Beispiel zu verwalten:

```
//: gui/InterruptableLongRunningCallable.java
// Using Callables for long-running tasks.
import javax.swing.*;
import java.awt.*;
```

```
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class CallableTask extends Task
    implements Callable<String> {
    public String call() {
        run();
        return ''Return value of '' + this;
    }
}

public class
    InterruptableLongRunningCallable extends JFrame {
    private JButton
        b1 = new JButton(''Start Long Running Task''),
        b2 = new JButton(''End Long Running Task''),
        b3 = new JButton(''Get results'');
    private TaskManager<String,CallableTask> manager =
        new TaskManager<String,CallableTask>();
    public InterruptableLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    CallableTask task = new CallableTask();
                    manager.add(task);
                    System.out.println(task + '' added to the queue'');
                }
            });
        b2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    for(String result : manager.purge())
                        System.out.println(result);
                }
            });
        b3.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    // Sample call to a Task method:
                    for(TaskItem<String,CallableTask> tt :
                            manager)
                        tt.task.id(); // No cast required
                    for(String result : manager.getResults())
                        System.out.println(result);
                }
            });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(b3);
    }
    public static void main(String[] args) {
        run(new InterruptableLongRunningCallable(), 200, 150);
    }
} ///:~
```

Die Klasse `CallableTask` unterscheidet sich von `Task` (definiert im Beispiel *InterruptableLongRunningTask.java*, Seite 1067) nur dadurch, daß sie einen Wert zurückgibt, genauer eine Zeichenkette, welche die Aufgabe identifiziert.

[170] Hilfsklassen wie `SwingWorker` (nicht aus der Java-Standarddistribution, siehe Internetseite von Sun Microsystems) und `Foxtrott` (*http://foxtrot.sourceforge.net*) dienen der Lösung eines ähnlichen Problems. Zum Zeitpunkt meiner Arbeit an diesem Buch waren diese Hilfsklassen noch nicht genug weit entwickelt, um sie mit dem *`Callable`*/*`Future`*-Mechanismus zusammen nutzen zu können.

[171] Es ist häufig wichtig, dem Benutzer einen visuellen Eindruck davon zu vermitteln, daß eine Aufgabe verarbeitet wird und wie weit die Verarbeitung bereits fortgeschritten ist. Hierfür wird üblicherweise eine `JProgressBar`-Komponente oder ein `ProgressMonitor`-Dialogfenster eingesetzt. Das folgende Beispiel verwendet ein `ProgressMonitor`-Dialogfenster:

```
//: gui/MonitoredLongRunningCallable.java
// Displaying task progress with ProgressMonitors.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.concurrent.*;
import net.mindview.util.*;
import static net.mindview.util.SwingConsole.*;

class MonitoredCallable implements Callable<String> {
    private static int counter = 0;
    private final int id = counter++;
    private final ProgressMonitor monitor;
    private final static int MAX = 8;
    public MonitoredCallable(ProgressMonitor monitor) {
        this.monitor = monitor;
        monitor.setNote(toString());
        monitor.setMaximum(MAX - 1);
        monitor.setMillisToPopup(500);
    }
    public String call() {
        System.out.println(this + '' started'');
        try {
            for(int i = 0; i < MAX; i++) {
                TimeUnit.MILLISECONDS.sleep(500);
                if(monitor.isCanceled())
                    Thread.currentThread().interrupt();
                final int progress = i;
                SwingUtilities.invokeLater(
                    new Runnable() {
                        public void run() {
                                monitor.setProgress(progress);
                        }
                    });
            }
        } catch(InterruptedException e) {
            monitor.close();
            System.out.println(this + '' interrupted'');
            return ''Result: '' + this + '' interrupted'';
        }
        System.out.println(this + '' completed'');
        return ''Result: '' + this + '' completed'';
    }
    public String toString() { return ''Task '' + id; }
};

public class MonitoredLongRunningCallable extends JFrame {
```

```
    private JButton
        b1 = new JButton("Start Long Running Task"),
        b2 = new JButton("End Long Running Task"),
        b3 = new JButton("Get results");
    private TaskManager<String,MonitoredCallable> manager =
        new TaskManager<String,MonitoredCallable>();
    public MonitoredLongRunningCallable() {
        b1.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    MonitoredCallable task = new MonitoredCallable(
                        new ProgressMonitor(
                            MonitoredLongRunningCallable.this,
                            "Long-Running Task", "", 0, 0)
                        );
                    manager.add(task);
                    System.out.println(task + " added to the queue");
                }
            });
        b2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    for(String result : manager.purge())
                        System.out.println(result);
                }
            });
        b3.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    for(String result : manager.getResults())
                        System.out.println(result);
                }
            });
        setLayout(new FlowLayout());
        add(b1);
        add(b2);
        add(b3);
    }
    public static void main(String[] args) {
        run(new MonitoredLongRunningCallable(), 200, 500);
    }
} ///:~
```

[172] Der Konstruktor der Klasse `MonitoredCallable` erwartet ein `ProgressMonitor`-Objekt und die `call()`-Methode aktualisiert den Fortschrittsbalken jede halbe Sekunde. Beachten Sie, daß `MonitoredCallable` eine separate Aufgabe ist und daher nicht direkt in die Steuerung der GUI eingreifen soll. Daher wird die statische `SwingUtilities`-Methode `invokeLater()` aufgerufen, um die Änderungsinformation über den Fortschrittsbalken an das `ProgressMonitor`-Objekt zu übergeben. Das Swing-Tutorial von Sun Microsystems zeigt einen alternativen Ansatz mit Hilfe der Swing-Klasse `Timer`, welche den Status der Aufgabe prüft und den Monitor aktualisiert.

[173] Wird die „Cancel"-Schaltfläche des Monitors betätigt, so liefert `monitor.isCanceled()` `true`. Im obigen Beispiel ruft die Aufgabe einfach die `interrupt()`-Methode ihres eigenen Threads auf, woraufhin die Programmausführung in die `catch`-Klausel eintritt und den Monitor per `close()`-Methode beendet.

[174] Der übrige Quelltext entspricht effektiv dem vorigen Beispiel, mit Ausnahme der Erzeugung des `ProgressMonitor`-Objektes im Konstruktor der Klasse `MonitoredLongRunningCallable`.

Übungsaufgabe 33: (6) Ändern Sie das Beispiel *InterruptableLongRunningCallable.java* so, daß die Aufgabe parallel statt sequentiell verarbeitet werden. ■

23.10.2 Visualisierte Threadaktivität

[175] Die Klasse `CBox` im folgenden Beispiel ist von `JPanel` abgeleitet, implementiert gleichzeitig das Interface ***`Runnable`*** und zeichnet ein Quadrat dessen Fläche verschiedene Farben annimmt. Die Kantenlänge des Gitters aus diesen Quadraten sowie die Wartezeit bis zum nächsten Umschalten der Farbe können auf der Kommandozeile übergeben werden. Durch Spielen mit diesen beiden Parametern entdecken Sie eventuell einige interessante und vielleicht unerklärliche Eigenschafen in der Threadunterstützung Ihres Betriebssystems:

```
//: gui/ColorBoxes.java
// A visual demonstration of threading.
import javax.swing.*;
import java.awt.*;
import java.util.concurrent.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

class CBox extends JPanel implements Runnable {
    private int pause;
    private static Random rand = new Random();
    private Color color = new Color(0);
    public void paintComponent(Graphics g) {
        g.setColor(color);
        Dimension s = getSize();
        g.fillRect(0, 0, s.width, s.height);
    }
    public CBox(int pause) { this.pause = pause; }
    public void run() {
        try {
            while(!Thread.interrupted()) {
                color = new Color(rand.nextInt(0xFFFFFF));
                repaint(); // Asynchronously request a paint()
                TimeUnit.MILLISECONDS.sleep(pause);
            }
        } catch(InterruptedException e) {
            // Acceptable way to exit
        }
    }
}

public class ColorBoxes extends JFrame {
    private int grid = 12;
    private int pause = 50;
    private static ExecutorService exec =
        Executors.newCachedThreadPool();
    public void setUp() {
        setLayout(new GridLayout(grid, grid));
        for(int i = 0; i < grid * grid; i++) {
            CBox cb = new CBox(pause);
            add(cb);
            exec.execute(cb);
        }
    }
```

```
    public static void main(String[] args) {
        ColorBoxes boxes = new ColorBoxes();
        if(args.length > 0)
            boxes.grid = new Integer(args[0]);
        if(args.length > 1)
            boxes.pause = new Integer(args[1]);
        boxes.setUp();
        run(boxes, 500, 400);
    }
} ///:~
```

Die Klasse `ColorBoxes` legt mit Hilfe des Layoutmanagers `GridLayout` ein zweidimensionales Gitter von Zellen an. Anschließend wird jeder Zelle ein `CBox`-Objekt (Aufgabe) zugeordnet und die individuelle Wartezeit zwischen zwei Farbumschaltungen eingestellt. In der `main()`-Methode sehen Sie, wie Sie die voreingestellten Parameterwerte über Kommandozeilenargumente ändern können.

[176] Die eigentliche Arbeit wird in der Klasse `CBox` verrichtet, die von `JPanel` abgeleitet ist und das Interface *`Runnable`* implementiert, so daß jedes `CBox`-Objekt eine unabhängige Aufgabe ist. Diese Aufgaben werden mit Hilfe eines Threadpools verarbeitet.

[177–179] Das `color`-Feld gibt die aktuelle Farbe an. Farben werden mit Hilfe der Klasse `Color` erzeugt, wobei der Konstruktor ein 24 Bit langen Wert erwartet, der hier zufällig erzeugt wird. Die `paintComponent()`-Methode legt die Farbe des Quadrates fest und zeichnet dessen Fläche mit der eingestellten Farbe neu. Die `run()`-Methode enthält eine unendliche Schleife, die das `color`-Feld auf einen neuen Zufallswert setzt und anschließend die `repaint()`-Methode aufruft, um das Quadrat neu zu zeichnen. Anschließend „schläft“ der Thread für die per Kommandozeile übergebene Anzahl Millisekunden.

[180] Das Aufrufen von `paint()` in der `run()`-Methode bedarf der Untersuchung. Auf den ersten Blick werden viele Threads erzeugt, die jeweils das Neuzeichnen ihres Quadrates erzwingen. Dadurch scheint das Prinzip verletzt zu werden, nach dem solche Aufgaben nur an die Warteschlange des Ereignisbehandlerthreads übergeben werden sollen. Allerdings ändern diese Threads die gemeinsame Resource nicht wirklich. Ein Aufruf der `repaint()`-Methode erzwingt kein unmittelbares Neuzeichnen, sondern setzt ein Flag, welches anzeigt, daß der Ereignisbehandlerthread diesen Bereich beim nächsten Neuzeichnen berücksichtigen soll. Das obige Programm verursacht also keine Threadprobleme bei Swing.

[181] Ruft der Ereignisbehandlerthread die `paint()`-Methode auf, um den Bildschirm neu zu zeichnen, so ruft er zunächst `paintComponent()`, danach `paintBorder()` und schließlich `paintChildren()` auf. Wenn Sie die `paint()`-Methode in einer abgeleiteten Komponente überschreiben, müssen Sie die Basisklassenversion aufrufen, damit die gerade beschriebenen Aktionen ausgeführt werden.

[182] *Precisely because this design is flexible and threading is tied to each JPanel element*, können Sie mit beliebig vielen Threads experimentieren. (Die Anzahl der Threads ist durch Ihre Laufzeitumgebung eingeschränkt.)

[183] Das Programm *ColorBoxes.java* liefert außerdem einen interessanten Orientierungspunkt, da sich bei verschiedenen Laufzeitumgebungen sowie unter verschiedenen Betriebssystemen dramatische Unterschiede in Performanz und Verhalten ergeben können.

Übungsaufgabe 34: (4) Ändern Sie das Beispiel *ColorBoxes.java* so, daß es Punkte („Sterne“) im Darstellungsbereich verteilt und anschließend die Farben dieser Punkte ändert. ■

23.11 Visuelle Programmierung und JavaBeans

[184] Sie haben beim Lesen dieses Buches gesehen, wie wertvoll Java hinsichtlich der Wiederverwendbarkeit von Quelltext ist. Die Klasse ist die Einheit mit dem besten Wiederverwendungswert, da sie eine zusammenhängende Einheit von Eigenschaften (Feldern) und Verhalten (Methoden) umfaßt, die entweder direkt per Komposition oder per Ableitung wiederverwendet werden kann.

[185] Ableitung und Polymophie sind zwar wesentliche Elemente der objektorientierten Programmierung, aber was Sie sich wirklich wünschen, wenn Sie eine Anwendung zusammensetzen, sind Komponenten, die exakt das können, was Sie brauchen. Sie wünschen sich, diese Komponenten in Ihren Entwurf zu platzieren, wie ein Elektronik-Ingenieur Chips auf einer Platine einsetzt. Es ist wünschenswert, diesen Programmierstil des „modularen Zusammenbaus“ zu beschleunigen.

[186] Die „Visuelle Programmierung“ war erstmals mit Visual Basic (VB) von Microsoft erfolgreich, sogar sehr erfolgreich, gefolgt von einem Design zweiter Generation in Form von Borland Delphi (die Hauptinspiration für das Design der JavaBeans). Bei diesen Programmierwerkzeugen waren die Komponenten visuell dargestellt. Das ist sinnvoll, da die Komponenten in der Regel eine visuelle Komponente wie eine Schaltfläche oder ein Texteingabefeld repräsentieren. Die visuelle Darstellung entspricht häufig exakt der Darstellung einer Komponente im laufenden Programm. Ein Teil der Arbeit beim visuellen Programmieren besteht darin, eine Komponente aus einer Palette auszuwählen und auf dem Formular zu platzieren. Die Application Builder IDE generiert während dessen die erforderlichen Anweisungen, durch welche die Komponente später im laufenden Programm angezeigt wird.

[187] Das Platzieren einer Komponente auf dem Formular ist im allgemeinen noch nicht hinreichend, damit das Programm komplett ist. Häufig müssen Sie noch die Eigenschaften der Komponente konfigurieren, etwa die Farbe, die Beschriftung oder die Datenbankverbindung. Sie bearbeiten die Eigenschaften Ihrer Komponente in der IDE, während Sie das Programm entwickeln. Die Einstellungen werden gesichert, so daß Sie beim nächsten Programmstart regeneriert werden können.

[188] Sie haben sich sicher an die Idee gewöhnt, daß ein Objekt mehr ist, als die Gesamtheit seiner Eigenschaften: Ein Objekt hat auch ein Verhalten. Während der Entwurfsphase wird das Verhalten einer visuellen Komponente teilweise durch Ereignisse dargestellt, das heißt durch Geschehen, welches die Komponente betreffen kann. In der Regel definieren Sie, was beim Eintreten eines Ereignisses geschehen soll, in dem Sie spezielle Anweisungen für diesen Fall hinterlegen.

[189] Hier ist die wesentliche Stelle: Die Entwicklungsumgebung nutzt den Reflexionsmechanismus, um die Komponente dynamisch abzufragen und zu ermitteln, welche Eigenschaften und Ereignisse die Komponente unterstützt. Sind diese Informationen einmal bekannt, können die Eigenschaften präsentiert und erforderlichenfalls geändert werden (der Zustand wird gesichert, when you build the program). Auch die Ereignisse können angezeigt werden. Im allgemeinen doppelklicken Sie auf ein Ereignis, woraufhin die IDE ein Gerüst generiert und mit diesem Ereignis verknüpft. Sie müssen lediglich die Anweisungen einsetzen, die beim Eintreten des Ereignisses ausgeführt werden sollen.

[190] Alles in allem nimmt Ihnen die IDE viel Arbeit ab und gestatten Ihnen dadurch, sich auf die Erscheinungsform und Funktionalität des Programms zu konzentrieren, während sich die IDE um die Verbindungsdetails kümmert. Visuelle Programmierwerkzeuge sind so erfolgreich, weil sie die Entwicklung einer Anwendung dramatisch beschleunigen, zumindest die Benutzerschnittstelle, häufig aber auch andere Teile der Anwendung.

23.11.1 Was ist eine JavaBean?

[191] Nachdem sich der Staub gelegt hat, zeigt sich, daß eine Komponente nicht mehr als ein Block von Anweisungen ist, typischerweise im Körper einer Klasse. Die Schlüsselfrage lautet: „Wie ermittelt die IDE, welche Eigenschaften und Ereignisse eine Komponente hat?“ Das Anlegen einer Komponente in Visual Basic erforderte vom Programmierer ursprünglich ein ziemlich kompliziertes Stück Quelltext zu schreiben, wobei bestimmte Konventionen hinsichtlich der exponierten Eigenschaften und Ereignisse eingehalten werden mußten (die Vorgehensweise wurde im Laufe der Jahre erleichtert). Delphi war ein visuelles Programmierwerkzeug der zweiten Generation und die Sprache wurde actively around visuelle Programmierung entworfen, wodurch sich visuelle Komponenten viel leichter erzeugen ließen. Java hat das Erzeugen visueller Komponenten mit JavaBeans auf die höchste Entwicklungsstufe gestellt, denn eine JavaBean ist nichts weiter als eine Klasse. Es ist nicht notwendig, zusätzliche Anweisungen oder spezielle Spracherweiterungen zu verwenden, um „etwas“ in eine JavaBean zu verwandeln. Sie müssen nichts weiter tun, als bei der Benennung Ihrer Methoden einige Richtlinien zu befolgen. Die IDE ermittelt anhand des Methodennamens, ob es sich um eine Eigenschaft, ein Ereignis oder eine gewöhnliche Methode handelt.

[192] In der Dokumentation des Java Development Kits wird diese Namenskonvention irrtümlicherweise als „Entwurfsmuster“ bezeichnet. Das ist bedauerlich, da das Gebiet der Entwurfsmuster (siehe *Thinking in Patterns* unter der Webadresse *http://www.mindview.net*) bereits ohne solche Verwechselungen anspruchsvoll genug ist. Die Richtlinien für die Benennung von Methodennamen bei JavaBeans ist eine Konvention, kein Entwurfsmuster. Die Regeln sind nicht schwierig:

- Zu einer Eigenschaft `xxx` legen Sie typischerweise zwei Methoden an: `getXxx()` und `setXxx()`. Der erste Buchstabe nach „get“ beziehungsweise „set“ wird von Werkzeugen, welche die Methodennamen untersuchen automatisch in einen Kleinbuchstaben umgewandelt, um auf den Namen der Eigenschaft zu schließen. Der Rückgabetyp der `getXxx()`-Methode ist der Typ des Argumentes der `setXxx()`-Methode. Zwischen dem Namen der Eigenschaft (`xxx`) und dem Rückgabe- beziehungsweise Argumenttyp von `getXxx()` beziehungsweise `setXxx()` besteht keine Beziehung.
- Bei einer `boolean`-Eigenschaft können Sie den obigen `getXxx()`/`setXxx()`-Ansatz verwenden oder auch `getXxx()` durch `isXxx()` ersetzen.
- Gewöhnliche Methoden der JavaBean-Klasse unterliegen keiner Namenskonvention, sind aber öffentlich.
- Bei Ereignissen gilt die Richtlinie für Swing-Ereignisbehandler: Die Methoden `addBounceListener(BounceListener)` und `removeBounceListener(BounceListener)` registrieren ein Ereignis vom Typ `BounceEvent` beziehungsweise löschen die Registrierung. In der Regel kommen Sie mit den eingebauten Ereignisbehandlern aus. Sie können aber auch eigene Ereignistypen und Behandlerinterfaces definieren.

[193] Das folgende Beispiel zeigt eine einfache JavaBean nach den obigen Regeln:

```
//: frogbean/Frog.java
// A trivial JavaBean.
package frogbean;
import java.awt.*;
import java.awt.event.*;

class Spots {}

public class Frog {
    private int jumps;
    private Color color;
```

```
    private Spots spots;
    private boolean jmpr;
    public int getJumps() { return jumps; }
    public void setJumps(int newJumps) {
        jumps = newJumps;
    }
    public Color getColor() { return color; }
    public void setColor(Color newColor) {
        color = newColor;
    }
    public Spots getSpots() { return spots; }
    public void setSpots(Spots newSpots) {
        spots = newSpots;
    }
    public boolean isJumper() { return jmpr; }
    public void setJumper(boolean j) { jmpr = j; }
    public void addActionListener(ActionListener l) {
        //...
    }
    public void removeActionListener(ActionListener l) {
        // ...
    }
    public void addKeyListener(KeyListener l) {
        // ...
    }
    public void removeKeyListener(KeyListener l) {
        // ...
    }
    // An ''ordinary'' public method:
    public void croak() {
        System.out.println('Ribbet!'');
    }
} ///:~
```

Zuerst fällt auf, daß `Frog` nur eine gewöhnliche Klasse ist. In der Regel sind alle Ihre Felder als `private` deklariert und nur durch Methoden ~~or properties~~ erreichbar. Der Namenskonvention folgend, heißen die Eigenschaften der JavaBean `Frog`: `jumps`, `color`, `spots` und `jumper` (beachten Sie, daß der erste Buchstabe jedes Eigenschaftsnamens klein geschrieben ist). Obwohl in den ersten drei Fällen der Name des Feldes mit dem Namen der Eigenschaft übereinstimmt, zeigt die vierte Eigenschaft (`jumper`), daß der Eigenschaftsname keinen bestimmten Feldnamen erzwingt (tatsächlich nicht einmal, daß überhaupt ein Feld für diese Eigenschaft existiert).

[194] Die JavaBean `Frog` behandelt zwei Typen von Ereignissen: `ActionEvent` und `KeyEvent`, wie Sie den Bezeichnern der `add`- und `remove`-Methoden für den entsprechenden Ereignisbehandler entnehmen können. Schließlich verrät das Schlüsselwort `public`, im Gegensatz zur Konformität bezüglich einer Namenskonvention, daß die `croak()`-Methode Bestandteil der JavaBean ist

23.11.2 Analyse von JavaBeans: Die Klassen Introspector und BeanInfo

[195] Wenn Sie eine JavaBean aus der Palette wählen und auf einem Formular platzieren, vollzieht sich ein entscheidender Vorgang: Die IDE muß nämlich in der Lage sein, ein Objekt der JavaBean-Klasse zu erzeugen (gewährleistet, wenn die Klasse einen Standardkonstruktor besitzt) und anschließend, ohne Kenntnis des Quelltextes dieser Klasse, alle benötigten Informationen ermitteln, um eine Übersicht über die Eigenschaften (*property sheet*) sowie die Ereignisbehandler der JavaBean anzeigen zu können.

[196] Ein Teil der Lösung ergibt sich aus Kapitel 15: Der Reflexionsmechanismus von Java entdeckt sämtliche Methoden einer unbekannten Klasse. Der Reflexionsmechanismus bietet sich zur Untersuchung von JavaBeans an, so daß keine zusätzlichen Schlüsselwörter benötigt werden, wie bei anderen visuellen Programmiersprachen. Tatsächlich war die Unterstützung von JavaBeans einer der Hauptgründe für die Aufnahme des Reflexionsmechanismus' in den Sprachumfang (der Reflexionsmechanismus unterstützt außerdem die Serialisierung von Objekten sowie die Remote Method Invocation und ist auch in der gewöhnlichen Programmierung nützlich). Sie können sich also vorstellen, daß die IDE per Reflexionsmechanismus die Methoden jeder JavaBean untersucht, um deren Eigenschaften und Ereignisse zu ermitteln.

[197] Diese Vorgehensweise wäre zwar möglich, aber die Designer von Java wollten ein standardisiertes Werkzeug liefern, um nicht nur die Verwendung von JavaBeans zu erleichtern, sondern auch um eine standardisierte Schnittstelle zum Erzeugen komplizierterer JavaBeans zur Verfügung zu stellen. Die Klasse `java.beans.Introspector` ist dieses Werkzeug und die statische Methode `getBeanInfo()` ist ihre wichtigste Methode. Die Methode erwartet eine Referenz auf ein Klassenobjekt, fragt sämtliche Gesichtspunkte der Klasse ab und gibt ein *`java.beans.BeanInfo`*-Objekt zurück, welches Sie analysieren können, um Eigenschaften, Methoden und Ereignisse zu ermitteln.

[198] Im allgemeinen brauchen Sie sich nicht um solche Dinge zu kümmern. Sie verwenden standardisierte JavaBeans und müssen die technischen Vorgänge unter der Haube nicht kennen. Sie platzieren die JavaBeans einfach auf Ihrem Formular, konfigurieren ihre Eigenschaften und schreiben die Behandler für die benötigten Ereignisse. Es ist allerdings eine leerreiche Übungsaufgabe, die Klasse `Introspector` einmal zu verwenden, um die Informationen über eine JavaBean anzuzeigen. Ein Beispiel:

```
//: gui/BeanDumper.java
// Introspecting a Bean.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.beans.*;
import java.lang.reflect.*;
import static net.mindview.util.SwingConsole.*;

public class BeanDumper extends JFrame {
    private JTextField query = new JTextField(20);
    private JTextArea results = new JTextArea();
    public void print(String s) { results.append(s + "\n"); }
    public void dump(Class<?> bean) {
        results.setText("");
        BeanInfo bi = null;
        try {
            bi = Introspector.getBeanInfo(bean, Object.class);
        } catch(IntrospectionException e) {
            print("Couldn't introspect " + bean.getName());
            return;
        }
        for(PropertyDescriptor d: bi.getPropertyDescriptors(){
            Class<?> p = d.getPropertyType();
            if(p == null) continue;
            print("Property type:\n " + p.getName() +
                  "Property name:\n " + d.getName());
            Method readMethod = d.getReadMethod();
            if(readMethod != null)
                print("Read method:\n " + readMethod);
            Method writeMethod = d.getWriteMethod();
```

```
            if(writeMethod != null)
                print("Write method:\n " + writeMethod);
            print("====================");
        }
        print("Public methods:");
        for(MethodDescriptor m : bi.getMethodDescriptors())
            print(m.getMethod().toString());
        print("=====================");
        print("Event support:");
        for(EventSetDescriptor e: bi.getEventSetDescriptors()){
            print("Listener type:\n " +
                  e.getListenerType().getName());
            for(Method lm : e.getListenerMethods())
                print("Listener method:\n " + lm.getName());
            for(MethodDescriptor lmd :
                    e.getListenerMethodDescriptors() )
                print("Method descriptor:\n " + lmd.getMethod());
            Method addListener= e.getAddListenerMethod();
            print("Add Listener Method:\n " + addListener);
            Method removeListener = e.getRemoveListenerMethod();
            print("Remove Listener Method:\n "+ removeListener);
            print("====================");
        }
    }
    class Dumper implements ActionListener {
        public void actionPerformed(ActionEvent e) {
            String name = query.getText();
            Class<?> c = null;
            try {
                c = Class.forName(name);
            } catch(ClassNotFoundException ex) {
                results.setText("Couldn't find " + name);
                return;
            }
            dump(c);
        }
    }
    public BeanDumper() {
        JPanel p = new JPanel();
        p.setLayout(new FlowLayout());
        p.add(new JLabel("Qualified bean name:"));
        p.add(query);
        add(BorderLayout.NORTH, p);
        add(new JScrollPane(results));
        Dumper dmpr = new Dumper();
        query.addActionListener(dmpr);
        query.setText("frogbean.Frog");
        // Force evaluation
        dmpr.actionPerformed(new ActionEvent(dmpr, 0, ""));
    }
    public static void main(String[] args) {
        run(new BeanDumper(), 600, 500);
    }
} ///:~
```

[199] Die `BeanDumper`-Methode `dump()` verrichtet die gesamte Arbeit. Zuerst versucht die Methode, ein *`BeanInfo`*-Objekt zu erzeugen und ruft im Erfolgsfalle diejenigen Methoden des *`BeanInfo`*-

Objektes auf, welche Informationen über die Eigenschaften, Methoden und Ereignisse liefern. Das zweite Argument der statischen `Introspector`-Methode `getBeanInfo()` (`Object.class`) gibt an, bis zu welchem Basistyp in der Ableitungshierarchie die Analyse durchgeführt werden soll. Im obigen Beispiel endet die Untersuchung eine Ebene vor der Klasse `Object`, deren Komponenten wir ausschließen wollen.

[200] Die Methode `getPropertyDescriptors()` gibt ein `PropertyDescriptor`-Array zurück. Zu jedem Element dieses Arrays liefert `getPropertyType()` die Klasse des Objektes, welches Sie der Eigenschaft mittels ihrer Änderungsmethode zuweisen beziehungsweise mittels ihrer Abfragemethode auswerten. Die Methode `getName()` gibt das Pseudonym der Eigenschaft zurück, welches den Namen der Abfrage- beziehungsweise Änderungsmethode (*property methods*) entnommen wird. Die Methoden `getReadMethod()` und `getWriteMethod()` liefern die Abfrage- beziehungsweise Änderungsmethode der Eigenschaft in Form von `Method`-Objekten, die Sie verwenden können, um die Methode aufzurufen (Teil des Reflexionsmechanismus').

[201] Die Methode `getMethodDescriptors()` gibt ein `MethodDescriptor`-Array zurück, welches die öffentlichen Methoden sowie die Abfrage- und Änderungsmethoden der Eigenschaften der JavaBean repräsentiert. Pro Element dieses Arrays wird eine Referenz auf das zugehörige `Method`-Objekt angefordert und die Signatur der Methode ausgegeben.

[202] Die Methode `getEventSetDescriptors()` gibt ein `EventSetDescriptor`-Array zurück. Jedes Element dieses Arrays liefert Auskünfte über die Ereignisbehandler, die Methoden der Behandlerklasse sowie die Methoden zum Registrieren eines Ereignisbehandlers beziehungsweise zum Löschen eines registrierten Ereignisbehandlers. Das Beispiel *BeanDumper.java* gibt alle diese Informationen aus.

[203] Das Programm analysiert die JavaBean `Frog`. Die Ausgabe des Programms lautet (um einige unnötige Einzelheiten gekürzt):

```
Property type:
  Color
Property name:
  color
Read method:
  public Color getColor()
Write method:
  public void setColor(Color)
====================
Property type:
  boolean
Property name:
  jumper
Read method:
  public boolean isJumper()
Write method:
  public void setJumper(boolean)
====================
Property type:
  int
Property name:
  jumps
Read method:
  public int getJumps()
Write method:
  public void setJumps(int)
====================
```

```
Property type:
  frogbean.Spots
Property name:
  spots
Read method:
  public frogbean.Spots getSpots()
Write method:
  public void setSpots(frogbean.Spots)
====================
Public methods:
public void croak()
public void removeActionListener(ActionListener)
public void setColor(Color)
public int getJumps()
public void addKeyListener(KeyListener)
public void setJumper(boolean)
public boolean isJumper()
public void addActionListener(ActionListener)
public void setSpots(frogbean.Spots)
public Color getColor()
public frogbean.Spots getSpots()
public void setJumps(int)
public void removeKeyListener(KeyListener)
======================
Event support:
Listener type:
  ActionListener
Listener method:
  actionPerformed
Method descriptor:
  public abstract void actionPerformed(ActionEvent)
Add Listener Method:
  public void addActionListener(ActionListener)
Remove Listener Method:
  public void removeActionListener(ActionListener)
====================
Listener type:
  KeyListener
Listener method:
  keyPressed
Listener method:
  keyReleased
Listener method:
  keyTyped
Method descriptor:
  public abstract void keyPressed(KeyEvent)
Method descriptor:
  public abstract void keyReleased(KeyEvent)
Method descriptor:
  public abstract void keyTyped(KeyEvent)
Add Listener Method:
  public void addKeyListener(KeyListener)
Remove Listener Method:
  public void removeKeyListener(KeyListener)
====================
```

[204] Die Ausgabe zeigt den größten Teil dessen, was das `Introspector`-Objekt „sieht“, während es

das *BeanInfo*-Objekt zu Ihrer JavaBean erzeugt. Sie sehen, daß Typ und Name einer Eigenschaft voneinander unabhängig sind. Beachten Sie, daß der Eigenschaftsname in Kleinbuchstaben geschrieben wird. (Die einzige Ausnahme sind Eigenschaftsnamen, die mit mehr als einem Großbuchstaben beginnen.) Beachten Sie, daß die Methodennamen, die Sie in der obigen Ausgabe sehen (zum Beispiel bei den Abfrage- und Änderungsmethoden) eigentlich von den `Method`-Objekten zurückgegeben werden, über die Sie auch die entsprechende Methode aufrufen können.

[205] Die Liste öffentlicher Methoden beinhaltet auch Methoden, die nicht im Zusammenhang mit einer Eigenschaft oder einem Ereignis stehen, zum Beispiel `croak()`. Jede dieser Methoden kann programmatisch auf einem Objekt einer JavaBean aufgerufen werden und die IDE kann Ihnen alle diese Methoden anbieten, um Ihnen die Arbeit zu erleichtern.

[206] Schließlich sind sämtliche Ereignisse komplett mit Behandlern, deren Methoden sowie den `addXXXListener()`- und `removeXXXListener()`-Methoden angegeben. Wenn Sie das *BeanInfo*-Objekt zur Verfügung haben, können Sie alles wichtige über die entsprechende JavaBean herausfinden. Insbesondere können Sie die Methoden der JavaBean aufrufen, auch wenn Sie außer einem Objekt dieser Klasse keine weiteren Informationen haben (eine durch den Reflexionsmechanismus implementierte Fähigkeit).

23.11.3 Eine etwas kompliziertere JavaBean

[207] Das folgende Beispiel ist etwas anspruchsvoller. Die JavaBean ist von `JPanel` abgeleitet und zeichnet einen kleinen Kreis um die Position des Mauszeigers. Wenn Sie eine der Maustasten drücken, wird der Text „Bang!“ in der Mitte des Darstellungsbereiches ausgegeben und ein Ereignisbehandler ausgelöst.

[208] Die Eigenschaften, die Sie ändern können sind die Größe des Kreises, seine Farbe und der Text der beim Drücken eine Maustaste angezeigt wird. Die Klasse `BangBean` hat außerdem eigene `addActionListener()`- und `removeActionListener()`-Methoden, so daß Sie einen eigenen Ereignisbehandler registrieren können, der ausgelöst wird, wenn der Benutzer auf die `BangBean`-Komponente klickt. Sie sollten die Unterstützung von Eigenschaften und Ereignissen erkennen können:

```
//: bangbean/BangBean.java
// A graphical Bean.
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;

public class BangBean extends JPanel implements Serializable {
    private int xm, ym;
    private int cSize = 20; // Circle size
    private String text = ”Bang!”;
    private int fontSize = 48;
    private Color tColor = Color.RED;
    private ActionListener actionListener;
    public BangBean() {
        addMouseListener(new ML());
        addMouseMotionListener(new MML());
    }
    public int getCircleSize() { return cSize; }
    public void setCircleSize(int newSize) {
        cSize = newSize;
```

```
    }
    public String getBangText() { return text; }
    public void setBangText(String newText) {
        text = newText;
    }
    public int getFontSize() { return fontSize; }
    public void setFontSize(int newSize) {
        fontSize = newSize;
    }
    public Color getTextColor() { return tColor; }
    public void setTextColor(Color newColor) {
        tColor = newColor;
    }
    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
    }
    // This is a unicast listener, which is
    // the simplest form of listener management:
    public void addActionListener(ActionListener l)
        throws TooManyListenersException {
        if(actionListener != null)
            throw new TooManyListenersException();
        actionListener = l;
    }
    public void removeActionListener(ActionListener l) {
        actionListener = null;
    }
    class ML extends MouseAdapter {
        public void mousePressed(MouseEvent e) {
            Graphics g = getGraphics();
            g.setColor(tColor);
            g.setFont(new Font(''TimesRoman'', Font.BOLD, fontSize));
            int width = g.getFontMetrics().stringWidth(text);
            g.drawString(text, (getSize().width - width) /2, getSize().height/2);
            g.dispose();
            // Call the listener's method:
            if(actionListener != null)
                actionListener.actionPerformed(
                    new ActionEvent(BangBean.this,
                    ActionEvent.ACTION_PERFORMED, null));
        }
    }
    class MML extends MouseMotionAdapter {
        public void mouseMoved(MouseEvent e) {
            xm = e.getX();
            ym = e.getY();
            repaint();
        }
    }
    public Dimension getPreferredSize() {
        return new Dimension(200, 200);
    }
} ///:~
```

[209] Als erstes fällt auf, daß die Klasse `BangBean` das Interface *`Serializable`* implementiert. Da-

durch ist die IDE in der Lage, alle in der JavaBean gespeicherten Informationen per Serialisierung zu sichern, nachdem der Programmierer die Eigenschaften bewertet hat. Wenn die JavaBean beim nächsten Programmstart neu erzeugt wird, werden die gespeicherten Eigenschaften wiederhergestellt, so daß Sie genau den Zustand zum Entwurfszeitpunkt erhalten.

[210] Ihrer Signatur nach, kann die Methode `addActionListener()` eine Ausnahme vom Typ `TooManyListenersException` auswerfen. Dieser Ausnahmetyp weist die Registrierung eines Ereignisses im „Unicast-Modus“ aus. In der Regel werden Ereignisse im „Multicast-Modus“ bekannt gegeben, damit möglichst viele Behandler von einem Ereignis Kenntnis erhalten. Der „Multicast-Modus“ hat allerdings mit Threads zu tun und gehört in den folgenden Unterabschnitt 23.11.4. Bis dahin umgehen wir das Problem mit einem Ereignis im „Unicast-Modus“.

[211] Wenn Sie eine Maustaste drücken wird der Inhalt des `text`-Feldes in der Mitte des Darstellungsbereichs der `BangBean`-Komponente angezeigt. Sofern das `actionListener`-Feld nicht `null` enthält wird die `actionPerformed()`-Methode des Ereignisbehandlers mit einem neu erzeugten `ActionEvent`-Objekt aufgerufen. Bei jeder Bewegung des Mauszeigers werden die neuen Koordinaten abgerufen und der Darstellungsbereich neu gezeichnet (wobei der Text gelöscht wird).

[212] Die Klasse `BangBeanTest` testet `BangBean`:

```
//: bangbean/BangBeanTest.java
// {Timeout: 5} Abort after 5 seconds when testing
package bangbean;
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBeanTest extends JFrame {
    private JTextField txt = new JTextField(20);
    // During testing, report actions:
    class BBL implements ActionListener {
        private int count = 0;
        public void actionPerformed(ActionEvent e) {
            txt.setText("BangBean action "+ count++);
        }
    }
    public BangBeanTest() {
        BangBean bb = new BangBean();
        try {
            bb.addActionListener(new BBL());
        } catch(TooManyListenersException e) {
            txt.setText("Too many listeners");
        }
        add(bb);
        add(BorderLayout.SOUTH, txt);
    }
    public static void main(String[] args) {
        run(new BangBeanTest(), 400, 500);
    }
} ///:~
```

Wenn Sie eine JavaBean in einer IDE verwenden, brauchen Sie diese Testklasse nicht, aber sie ist praktisch, um schnell eine Ihrer JavaBeans zu testen. `BangBeanTest` platziert eine `BangBean`-Komponente in einer `JFrame`-Komponente, verknüpft einen einfachen *`ActionListener`*-Ereignisbehandler mit der `BangBean`-Komponente, um den Zähler im Texteingabefeld zu inkrementieren, wenn

ein Ereignis vom Typ `ActionEvent` ausgelöst wird. Die IDE würde natürlich den größten Teil der zum Einsatz der JavaBean erforderlichen Anweisungen generieren.

[213] Wenn Sie die Klasse `BangBean` mit dem Hilfsprogramm *BeanDumper.java* oder mit einer IDE untersuchen, werden Sie viel mehr Eigenschaften, Methoden und Ereignisse vorfinden, als im obigen Quelltext. Das liegt daran, daß `BangBean` von `JPanel` abgeleitet und `JPanel` selbst wiederum eine JavaBean ist, so daß auch die Eigenschaften, Methoden Ereignisse der Basisklasse angezeigt werden.

Übungsaufgabe 35: (6) Laden Sie eine oder mehrere der frei verfügbaren GUI-Builder aus dem Internet herunter oder verwenden Sie ein kommerzielles Produkt, wenn Sie eines besitzen. Finden Sie heraus, wie Sie `BangBean` mit einer solchen Umgebung verwenden können und schreiben Sie ein Testprogramm. ■

23.11.4 JavaBeans und Synchronisierung

[214] Sie müssen beim Entwickeln einer JavaBean davon ausgehen, daß sie von mehr als einem Thread zugleich beansprucht werden kann. Das bedeutet:

- Sofern möglich, sollten alle öffentlichen Methoden einer JavaBean synchronisiert werden. Dadurch nehmen Sie natürlich die entsprechenden Unkosten zu Laufzeit in Kauf (die allerdings bei den jüngeren Versionen des Java Development Kits deutlich nachgelassen haben). Falls diese Unkosten inakzeptabel sind, können Methoden, die keine Probleme durch kritische Anweisungen verursachen können, unsynchronisiert bleiben. Beachten Sie aber, daß derartige Methoden nicht immer offensichtlich sind. Geeignete Kandidaten bestehen tendentiell nur aus wenigen Anweisungen (siehe `getCircleSize()` im folgenden Beispiel) und/oder sind „atomar“, das heißt, daß der Methodenaufruf so schnell verarbeitet wird, daß das Objekt zwischenzeitlich nicht verändert werden kann (sehen Sie aber Kapitel 22 durch, um Ihr Verständnis des Begriffs „Atomizität“ zu überprüfen). Die Synchronisierung solcher Methoden zu entfernen, hat eventuell keine sichtbare Auswirkung auf Ihr Programm. Es ist besser, alle öffentlichen Methoden einer JavaBean zu synchronisieren und das Schlüsselwort `synchronized` nur dann zu entfernen, wenn Sie sicher wissen, daß Sie dadurch eine sichtbare Wirkung hervorrufen und die Methode trotz fehlender Synchronisierung sicher ist.
- Bei Multicast-Ereignissen, die an mehr als einen Ereignisbehandler gesendet werden, müssen Sie davon ausgehen, daß Ereignisbehandler hinzugefügt oder entfernt werden, während Sie die Behandlerliste verarbeiten.

[215] Der erste Punkt leuchtet ein, während der zweite Punkt etwas Überlegung voraussetzt. Das Beispiel *BangBean.java* umgeht das Threadproblem durch Vermeiden der Synchronisierung und Verwendung eines Unicast-Ereignisses. Das folgende Beispiel ist eine modifizierte Version, die von mehreren Threads zugleich beansprucht werden kann und Multicasting-Ereignisse versendet:

```
//: gui/BangBean2.java
// You should write your Beans this way so they
// can run in a multithreaded environment.
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.util.*;
import static net.mindview.util.SwingConsole.*;

public class BangBean2 extends JPanel
    implements Serializable {
    private int xm, ym;
```

```
private int cSize = 20; // Circle size
private String text = ”Bang!”;
private int fontSize = 48;
private Color tColor = Color.RED;
private ArrayList<ActionListener> actionListeners =
    new ArrayList<ActionListener>();
public BangBean2() {
    addMouseListener(new ML());
    addMouseMotionListener(new MM());
}
public synchronized int getCircleSize() { return cSize; }
public synchronized void setCircleSize(int newSize) {
    cSize = newSize;
}
public synchronized String getBangText() { return text; }
public synchronized void setBangText(String newText) {
    text = newText;
}
public synchronized int getFontSize(){ return fontSize; }
public synchronized void setFontSize(int newSize) {
    fontSize = newSize;
}
public synchronized Color getTextColor(){ return tColor;}
public synchronized void setTextColor(Color newColor) {
    tColor = newColor;
}
public void paintComponent(Graphics g) {
    super.paintComponent(g);
    g.setColor(Color.BLACK);
    g.drawOval(xm - cSize/2, ym - cSize/2, cSize, cSize);
}
// This is a multicast listener, which is more typically
// used than the unicast approach taken in BangBean.java:
public synchronized void
    addActionListener(ActionListener l) {
    actionListeners.add(l);
}
public synchronized void
    removeActionListener(ActionListener l) {
    actionListeners.remove(l);
}
// Notice this isn’t synchronized:
public void notifyListeners() {
    ActionEvent a = new ActionEvent(BangBean2.this,
                                    ActionEvent.ACTION_PERFORMED, null);
    ArrayList<ActionListener> lv = null;
    // Make a shallow copy of the List in case
    // someone adds a listener while we’re
    // calling listeners:
    synchronized(this) {
        lv = new ArrayList<ActionListener>(actionListeners);
    }
    // Call all the listener methods:
    for(ActionListener al : lv)
        al.actionPerformed(a);
}
class ML extends MouseAdapter {
```

```
        public void mousePressed(MouseEvent e) {
            Graphics g = getGraphics();
            g.setColor(tColor);
            g.setFont(
                      new Font("TimesRoman", Font.BOLD, fontSize));
            int width = g.getFontMetrics().stringWidth(text);
            g.drawString(text, (getSize().width - width) /2,
                         getSize().height/2);
            g.dispose();
            notifyListeners();
        }
    }
    class MM extends MouseMotionAdapter {
        public void mouseMoved(MouseEvent e) {
            xm = e.getX();
            ym = e.getY();
            repaint();
        }
    }
    public static void main(String[] args) {
        BangBean2 bb2 = new BangBean2();
        bb2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println("ActionEvent" + e);
                }
            });
        bb2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println("BangBean2 action");
                }
            });
        bb2.addActionListener(new ActionListener() {
                public void actionPerformed(ActionEvent e) {
                    System.out.println("More action");
                }
            });
        JFrame frame = new JFrame();
        frame.add(bb2);
        run(frame, 300, 300);
    }
} ///:~
```

[216] Die Synchronisierung der Methoden ist schnell erledigt. Beachten Sie in den Methoden `addActionListener()` und `removeActionListener()`, daß die Ereignisbehandler nun einem `ArrayList`-Objekt hinzugefügt werden, das heißt Sie können beliebig viele Ereignisbehandler registrieren.

[217] Wie Sie sehen, ist die Methode `notifyListeners()` *nicht* synchronisiert, kann also von mehr als einem Thread zugleich aufgerufen werden. Insbesondere können die Methoden `addActionListener()` und `removeActionListener()` aufgerufen werden, während `notifyListeners()` gerade verarbeitet wird. Das ist problematisch, da `notifyListeners()` die Liste der Ereignisbehandler traversiert. Zur Abschwächung dieses Problems wird in einem synchronisierten Block mit Hilfe des `ArrayList`-Konstruktors, der die Elemente seines Arguments kopiert, ein Duplikat dieser Liste angelegt und `notifyListeners()` arbeitet mit dieser Kopie. Dadurch kann die eigentliche Behandlerliste modifiziert werden, ohne daß sich eventuelle Änderungen auf `notifyListeners()` auswirken.

[218] Die Methode `paintComponent()` ist ebenfalls nicht synchronisiert. Die Entscheidung, ob eine überschriebene Methode synchronisiert werden muß oder nicht, ist weniger klar, als bei gewöhnli-

chen Methoden. Im obigen Beispiel hat sich gezeigt, daß `paintComponent()` stets zu funktionieren scheint, unabhängig davon, ob die Methode synchronsiert ist oder nicht. Sie müssen folgende Punkte beachten:

- Ändert die Methode den Zustand „kritischer“ Felder des Objektes? Ob ein Feld „kritisch“ ist oder nicht, hängt davon ab, ob es von anderen Threads im Programm abgefragt oder geändert wird. (In diesem Fall geschieht das Abfragen beziehungsweise Ändern fast immer über synchronisierte Methoden, das heißt es genügt, sich auf diese zu konzentrieren.) Die Methode `paintComponent()` ändert keine Felder.
- Hängt die Methode vom Zustand dieser „kritischen“ Felder ab? Ändert eine synchronisierte Methode ein von Ihrer Methode verwendetes Feld, so ist es sinnvoll, auch Ihre Methode zu synchronisieren. Da das `cSize`-Feld von der synchronisierten Methode `setCircleSize()` geändert wird, müßte `paintComponent()` nach der gerade formulierten Faustregel ebenfalls synchronisiert werden. Was kann im ungünstigsten Fall geschehen, wenn `cSize` während der Verarbeitung von `paintComponent()` verändert wird? Sind die Auswirkungen vertretbar und darüber hinaus flüchtig, so können Sie sich entscheiden, die `paintComponent()`-Methode unsychronisiert zu belassen, um die Unkosten durch das Aufrufen einer synchronisierten Methode zu vermeiden.
- Ein dritter Anhaltspunkt ist, ob die Basisklassenversion der überschriebenen Klasse synchronisiert ist. Bei `paintComponent()` ist dies nicht der Fall. Dies ist kein hieb- und stichfestes Argument, sondern nur ein Hinweis. Im obigen Beispiel enthält die Formel in `paintComponent()` ein Feld, das von einer synchronisierten Methode geändert wird (`cSize`), wodurch sich die Situation ändern kann. Beachten Sie, daß die Synchronisierung nicht vererbt wird, eine in der Basisklasse synchronisierte Methode beim Überschreiben in einer abgeleiteten Klasse also *nicht* automatisch synchronisiert ist.
- Die Methoden `paint()` und `paintComponent()` müssen so schnell wie möglich verarbeitet werden. Jeder Eingriff, der bei diesen Methoden Unkosten vermeidet, ist nachdrücklich zu empfehlen. Wenn Sie in Betracht ziehen, `paint()` und `paintComponent()` zu synchronisieren, könnte dies ein Anzeichen für schlechten Entwurf sein.

Die Anweisungen in der `main()`-Methode wurden verglichen mit *BangBeanTest.java* dahingehend verändert, daß einige zusätzliche Ereignisbehandler registriert werden, um die Multicast-Fähigkeit von `BangBean2` zu zeigen.

23.11.5 Archivieren einer JavaBean

[219] Bevor Sie eine JavaBean in eine IDE importieren können, müssen Sie sie in einem Container verpacken, genauer, einer *.jar* Datei, welche alle benötigten Klassen und eine sogenannte Manifest-Datei enthält, die die JavaBean als solche identifiziert. Die Manifest-Datei für die `BangBean`-JavaBean lautet:

```
Manifest-Version: 1.0

Name: bangbean/BangBean.class
Java-Bean: True
```

Die erste Zeile gibt die Version des Manifest-Dateiformats (*manifest schema*) an, bis auf weiteres 1.0. Die zweite Zeile (Leerzeilen werden nicht beachtet) bezeichnet die Datei *BangBean.class* und die dritte Zeile kennzeichnet diese Klasse als JavaBean. Ohne die dritte Zeile ist die IDE nicht in der Lage, die Klasse als JavaBean zu erkennen.

[220] Die einzige verzwickte Stelle ist die Pfadangabe beim `Name`-Feld. Wenn Sie zum Beispiel *BangBean.java* zurückblättern (Seite 1082), sehen Sie, daß die Klasse im Package `bangbean` liegt, also in einem Unterverzeichnis des Klassenpfades namens *bangbean*. Der Klassenname in der Manifest-Datei muß diese Packagezuordnung beinhalten. Außerdem müssen Sie die Manifest-Datei eine Ebene über dem Wurzelverzeichnis dieses Packageverzeichnisses deponieren, das heißt in diesem Fall in der Ebene, in der das Verzeichnis *bangbean* selbst liegt. Anschließend rufen Sie das `jar`-Kommando in dem Verzeichnis auf, in dem die Manifest-Datei und das Unterverzeichnis *bangbean* liegen:

```
jar cfm BangBean.jar BangBean.mf bangbean
```

[221] Diese Zeile setzt voraus, daß Sie die resultierende *.jar* Datei *BangBean.jar* nennen möchten und daß Ihre Manifest-Datei *BangBean.mf* heißt.

[222] Eventuell fragen Sie sich, wo die übrigen Klassen sind, die beim Übersetzen von *BangBean.java* generiert wurden. Nun, alle diese Klassen liegen im Unterverzeichnis *bangbean* und das letzte Argument des obigen `jar`-Kommandos lautet ebenfalls `bangbean`. Wenn Sie dem `jar`-Kommando den Namen eines Verzeichnisses übergeben, verpackt es das gesamte Verzeichnis in der *.jar* Datei (inklusive der ursprünglichen Quelltextdatei *BangBean.java*; wobei Sie sich vielleicht entscheiden, den Quelltext nicht in Ihre JavaBean aufzunehmen). Wenn Sie Ihre eben erzeugte *.jar* Datei wieder auspacken, werden Sie feststellen, daß `jar` eine eigene Manifest-Datei namens *MANIFEST.MF* generiert hat, die auf den Informationen aus Ihrer Version aufbaut. *MANIFEST.MF* befindet sich in einem Verzeichnis namens *META-INF* („Meta-Informationen“). Wenn Sie *MANIFEST.MF* öffnen, werden Sie sehen, daß `jar` für jede Datei eine digitale Signatur im folgenden Format gespeichert hat:

```
Digest-Algorithms: SHA MD5
SHA-Digest: pDpEAG9NaeCx8aFtqPI4udSX/OO=
MD5-Digest: O4NcSlhE3Smnzlp2hj6qeg==
```

Im allgemeinen brauchen Sie sich hierüber keine Gedanken zu machen. Wenn Sie Ihre JavaBean ändern, modifizieren Sie einfach Ihre ursprüngliche Manifest-Datei und rufen das `jar`-Kommando auf, um eine neue *.jar* Datei zu erzeugen. Sie können weitere JavaBeans in die *.jar* Datei aufnehmen, indem Sie einfach die entsprechenden Dateien in Ihre Manifest-Datei eintragen.

[223] Es ist sinnvoll, jede JavaBean in einem eigenen Verzeichnis zu deponieren, da Sie das `jar`-Kommando mit einem Unterverzeichnisnamen aufrufen, woraufhin es den gesamten Verzeichnisinhalt in einer *.jar* Datei archiviert. Die JavaBeans `Frog` und `BangBean` liegen jeweils in einem eigenen Unterverzeichnis.

[224] Nachdem Sie Ihre JavaBean korrekt in einer *.jar* Datei verpackt haben, können Sie sie in eine IDE importieren. Der Importvorgang ist bei jeder IDE verschieden. Sun Microsystems bietet eine frei verfügbare Testumgebung für JavaBeans an, nämlich den „Bean Builder“ (*http://java.sun.com/beans*). Beim „Bean Builder“ importieren Sie eine JavaBean dadurch, daß Sie sie in ein bestimmtes Unterverzeichnis kopieren.

Übungsaufgabe 36: (4) Tragen Sie die JavaBean *Frog.class* in die obige Manifest-Datei ein und rufen Sie das `jar`-Kommando auf, um eine *.jar* Datei zu erzeugen, die sowohl `Frog` als auch `BangBean` enthält. Laden Sie entweder den „Bean Builder“ herunter und installieren Sie das Programm oder verwenden Sie eine Entwicklungsumgebung Ihrer Wahl, um die *.jar* Datei zu importieren und die beiden JavaBeans zu testen. ■

Übungsaufgabe 37: (5) Schreiben Sie eine JavaBean namens `Valve` mit zwei Eigenschaften: Eine `boolean`-Eigenschaft `on` und eine `int`-Eigenschaft `level`. Schreiben Sie eine Manifest-Datei, archivieren Sie Ihre JavaBean per `jar` und importieren Sie sie in den „Bean Builder“ oder verwenden Sie eine Entwicklungsumgebung Ihrer Wahl, um sie testen zu können. ■

23.11.6 Unterstützung komplexer JavaBeans

[225] Sie haben gesehen, wie bemerkenswert einfach es ist, eine JavaBean zu schreiben. Sie sind keineswegs auf den in diesem Abschnitt dargestellten Umfang beschränkt. Die JavaBeans-Architektur liefert einen einfachen Einstiegspunkt, eignet sich aber auch für komplexere Anforderungen. Derartige Anwendungsfälle gehen über den Rahmen dieses Buches hinaus, werden aber in diesem Unterabschnitt kurz vorgestellt. Zu Einzelheiten siehe *http://java.sun.com/beans*.

[226] Eine Verfeinerungsmöglichkeit besteht beispielsweise bei den Eigenschaften. Bei den Beispielen in diesem Abschnitt kamen nur Eigenschaften mit Einzelwerten vor, es ist aber möglich, viele Werte in Gestalt eines Arrays darzustellen. Solche Eigenschaften heißen *indizierte Eigenschaften*. Es genügt, die passenden Abfrage- und Änderungsmethoden zur Verfügung zu stellen (wobei wiederum die Konvention für Methodennamen gilt), damit die Klasse `Introspector` die indizierte Eigenschaft erkennen und in Ihre IDE importieren kann.

[227] Eigenschaften können *gebunden* werden, das heißt sie benachrichtigen andere Objekte über Ereignisse vom Typ `PropertyChangeEvent`. Die anderen Objekte können somit ihren Zustand ändern, wenn sich der Zustand der JavaBean geändert hat.

[228] Eigenschaften können *beschränkt* werden, das heißt andere Objekte können einer inakzeptablen Änderung eines Eigenschaftswertes widersprechen. Die anderen Objekt werden mittels einer Ausnahme vom Typ `PropertyChangeEvent` benachrichtigt und können ihrerseits eine Ausnahme vom Typ `PropertyVetoException` auswerfen, um das Geschehen einer Änderung zu verhindern und den vorigen Eigenschaftswert zu rekonstruieren.

[229] Sie können die Darstellung Ihrer JavaBean bereits während der Designphase ändern:

- Sie können eine individuelle Übersicht über die Eigenschaften Ihrer JavaBean zur Verfügung stellen. Bei allen anderen JavaBeans wird die normale Übersicht verwendet, während bei Ihrer automatisch die individuelle Übersicht angezeigt wird.
- Sie können einen individuellen Editor für eine bestimmte Eigenschaft anlegen. Im allgemeinen wird die gewöhnliche Übersicht über die Eigenschaften verwendet. Wenn dagegen die bestimmte Eigenschaft geändert wird, so wird automatisch der individuelle Editor aufgerufen.
- Sie können zu Ihrer JavaBean eine individuelle *`BeanInfo`*-Klasse zur Verfügung stellen, die andere Informationen enthält, als die von der `Introspector`-Klasse erzeugte Version.
- Sie können bei allen `FeatureDescriptor`-Objekten einen „Expertenmodus“ ein- beziehungsweise ausschalten, um zwischen grundlegenden und fortgeschrittenen Eigenschaften unterscheiden zu können.

23.11.7 Weiterführende Informationen über JavaBeans

[230] Es gibt viele Bücher über JavaBeans, zum Beispiel Elliotte Rusty Harolds *JavaBeans* (IDG, 1998).

23.12 Alternativen zu Swing

[231] Obwohl die von Sun Microsystems bestätigte GUI-Bibliothek, ist Swing keinesfalls die einzige Möglichkeit, um graphische Benutzeroberflächen zu entwickeln. Zwei wichtige Alternativen sind

Macromedia Flash (basierend auf dem Flex-System von Macromedia) für die Clientseite von Webapplikationen und die quelloffene Eclipse-Bibliothek *Standard Widget Toolkit* (SWT) für Desktop-Anwendungen.

[232] Warum sollten Sie Alternativen in Betracht ziehen? Im Hinblick auf Webapplikationen haben Sie ein starkes Argument: Applets sind gescheitert. Eine Webapplikation mit Applets ist noch immer eine Überraschung, auch wenn Applets von Anfang zu Java gehörten sowie angesichts des anfänglichen Rummels und der Verheißungen rund um Applets. Auch Sun Microsystems selbst scheint Applets auszuweichen, siehe beispielsweise

`http://java.sun.c0m/developer/onlineTraining/new2java/javamap/intro.html`

Eine interaktive Karte der Eigenschaften und Fähigkeiten von Java auf der Internetseite von Sun Microsystems wäre ein wahrscheinlicher Kandidat für ein Java-Applet. Dennoch wurde die Karte in Flash programmiert. Dies scheint eine stillschweigende Bestätigung dafür zu sein, daß sich die Applets nicht durchgesetzt haben. Wichtiger ist aber, daß der Flash-Player auf fast jedem Rechner installiert ist, also als akzeptierter Standard betrachtet werden kann. Wie Sie sehen werden, stellt das Flex-System eine sehr mächtige clientseitige Programmierumgebung zur Verfügung, mit Sicherheit mächtiger als JavaScript und mit einem häufig angenehmeren Look-and-Feel als bei Applets. Wenn Sie Applets verwenden wollen, müssen Sie noch immer den Benutzer überzeugen, sich die Laufzeitumgebung herunterzuladen, während der Flash-Player vergleichsweise klein ist und sich schnell herunterladen läßt.

[233] Bei Desktop-Anwendungen mit Swing ist problematisch, daß die Benutzer wahrnehmen, daß sie eine andersartige Anwendung benutzen, da sich das Look-and-Feel von Swing-Anwendungen vom normalen Desktop unterscheidet. Im allgemeinen interessieren sich die Benutzer nicht für neue Look-and-Feels, sondern versuchen, ihre Arbeit zu erledigen und bevorzugen eine Anwendung, die so aussieht und sich so verhält, wie sie es von ihren anderen Anwendungen gewohnt sind. SWT liefert Anwendungen, die wie native Anwendungen aussehen und da die Bibliothek soviele native Komponenten wie möglich verwendet, sind SWT-Anwendung tendentiell schneller als Swing-Anwendungen.

23.13 Webbrowserbasierte Flex-Clients

[234] Aufgrund der Allgegenwart des Flash-Players von Macromedia, sind sie meisten Benutzer in der Lage, Flash-basierte Anwendungen zu betreiben, ohne etwas installieren zu müssen und die Anwendungen haben unabhängig von Rechnerarchitektur und Betriebssystem stets dasselbe Look-and-Feel.[9]

[235] Das Flex Software Development Kit (Flex SDK) von Macromedia gestattet Ihnen Flash-Benutzerschnittstellen für Java-Anwendungen zu entwickeln. Flex besteht aus einer Kombination aus XML und einer Skriptsprache (wie HTML und JavaScript) und einer robusten Komponentenbibliothek. Die MXML-Syntax wird verwendet, um Layout und Komponenten zu deklarieren. Die Skriptsprache ergänzt MXML um Ereignisbehandlung und Aufrufe von Diensten, welche die Benutzerschnittstelle mit Java-Klassen, Datenmodellen, Webservices und so weiter verbinden. Der Flex-Compiler übersetzt Ihr MXML-Skript in Bytecode. Die clientseitige Flash-Laufzeitumgebung verhält sich wie eine Java-Laufzeitumgebung und interpretiert den übersetzten Bytecode. Das Flash-Byteformat heißt Shockwave Flash (SWF) und wird vom Flex-Compiler erzeugt.

[236] Es gibt eine quelloffene Alternative zu Flex unter der Webadresse *http://openlaszlo.org*. Die Struktur ähnelt Flex und mag für den einen oder anderen Leser interessant sein. Es gibt noch weitere Werkzeuge, um auf verschiedene Weise Flash-Anwendungen zu erzeugen.

[9] Der Kern dieses Abschnitts stammt von Sean Neville.

23.13.1 „Hello Flex“

[237] Das folgende MXML-Skript definiert eine Benutzerschnittstelle:

```
<?xml version=''1.0'' encoding=''utf-8''?>
<mx:Application
  xmlns:mx=''http://www.macromedia.com/2003/mxml''
  backgroundColor=''#ffffff''>
  <mx:Label id=''output'' text=''Hello, Flex!'' />
</mx:Application>
```

MXML-Skripte sind XML-Dokumente und beginnen daher mit einer XML-Deklaration. Das Wurzelelement des MXML-Skriptes ist `<mx:Application>`, welches zugleich den obersten visuellen und logischen Container einer Flex-Benutzerschnittstelle repräsentiert. Innerhalb des `<mx:Application>`-Elementes werden Elemente angelegt, die visuelle Komponenten darstellen, zum Beispiel das `<mx:Label>`-Element. Komponenten werden stets in einem Container angelegt. Container kapseln unter anderem den Layoutmanager, überwachen also die Anordnung der in ihnen enthaltenen Komponenten. Im einfachsten Fall übernimmt `<mx:Application>` die Funktion eines Containers, wie im obigen Beispiel. Der voreingestellte Layoutmanager des `<mx:Application>`-Containers platziert die Komponenten der Schnittstelle in der Reihenfolge ihrer Deklaration vertikal untereinander.

[238] ActionScript ist (wie JavaScript) eine Ausprägung des ECMAScript-Standards, ähnelt Java und unterstützt Klassen, strenge Typisierung und dynamic scripting. Durch ein Skript können wir dem Beispiel Verhalten hinzufügen. Das Element `<mx:Script>` verwendet, um ein ActionScript direkt im MXML-Skript zu platzieren:

```
<?xml version=''1.0'' encoding=''utf-8''?>
<mx:Application
  xmlns:mx=''http://www.macromedia.com/2003/mxml''
  backgroundColor=''#ffffff''>
  <mx:Script>
    <![CDATA[
    function updateOutput() {
      output.text = ''Hello! '' + input.text;
    }
    ]]>
  </mx:Script>
  <mx:TextInput id=''input'' width=''200''
    change=''updateOutput()'' />
  <mx:Label id=''output'' text=''Hello!'' />
</mx:Application>
```

Die `<mx:TextInput>`-Komponente erwartet eine Benutzereingabe und die `<mx:Label>`-Komponente gibt die vom Benutzer eingegebenen Zeichen wieder aus. Beachten Sie, daß der Wert eines `id`-Attribut eines MXML-Elementes dem Skript als Variablenname zur Verfügung steht, so daß ein MXML-Skript seine Elemente referenzieren kann. Das `change`-Attribute des `<mx:TextInput>`-Elementes ist mit der `updateOutput()`-Funktion verbunden, die jedesmal aufgerufen wird, wenn sich der Zustand des Texteingabefeldes ändert.

23.13.2 Übersetzen eines MXML-Skriptes

[239] Die kostenlose Flex-Probeversion von Macromedia, die Sie unter der Webadresse *http://www-.adobe.com/products/flex* herunterladen können, ist der einfachste Einstieg.[10] Es gibt verschiedene

[10] Achten Sie darauf, daß Sie das Flex Software Development Kit (Flex SDK) herunterladen, nicht den FlexBuilder. Letzterer ist eine Entwicklungsumgebung.

Versionen von der kostenlosen Probeversion bis zum Enterprise-Server. Macromedia bietet darüber hinaus Entwicklungswerkzeuge für Flex-Anwendungen an. Der genaue Inhalt der einzelnen Pakete ändert sich von Zeit zu Zeit. Auf der Website von Macromedia finden Sie alle benötigten Informationen. Beachten Sie, daß Sie eventuell die Datei *jvm.config* im Verzeichnis *bin* Ihrer Flex-Installation anpassen müssen.

[240] Es gibt zwei Möglichkeiten, um ein MXML-Skript in Flash-Bytecode zu übersetzen:

- Sie können das MXML-Skript in einer Java-Webapplikation deponieren, neben JSP- und HTML-Seiten in einer *.war* Datei. Das MXML-Skript kann bei Bedarf zur Laufzeit übersetzt werden, wenn eine browserseitige Anfrage an seine URL eingeht.

- Sie können das MXML-Skript mit Hilfe des Compilers `mxmlc` auf der Kommandozeile übersetzen.

Option 1, die Übersetzung zur Laufzeit der Webapplikation, erfordert neben Flex einen Servletcontainer, wie Tomcat (Apache). Die Webapplikationen (*.war* Dateien) im Deploymentverzeichnis des Servletcontainers müssen um die benötigten Informationen über die Flex-Konfiguration ergänzt werden, etwa durch Servletabbildungen im Deployment-Deskriptor (*web.xml*). Der Servletcontainer benötigt außerdem die *.jar* Dateien zu Flex. Nach der Konfiguration der Webapplikationen können Sie die MXML-Skripte in deren Verzeichnisbaum platzieren und anschließend in einem beliebigen Browser die URL zu Ihrem MXML-Skript aufrufen. Flex übersetzt das MXML-Skript analog zum JSP-Modell beim ersten Zugriff und gibt anschließend den übersetzten Bytecode (*.swf* Datei) in eine HTML-Seite eingebettet zurück.

[241] Option 2 benötigt keinen Server. Sie rufen den Flex-Compiler `mxmlc` auf der Kommandozeile auf, um *.swf* Dateien zu erzeugen, die Sie nach Ihren Wünschen deployen können. Die ausführbaren Datei `mxmlc` befindet sich im Verzeichnis *bin* Ihrer Flex-Installation und gibt, ohne Argumente aufgerufen, eine Liste der gültigen Kommandozeilenschalter und -optionen zurück. Typischerweise geben Sie den Pfad zur clientseitigen Bibliothek der Flex-Komponenten über die Option `-flexlib` an. Bei einfachen Beispielen, wie in den beiden obigen Fällen, nimmt der Compiler an, daß sich die Komponentenbibliothek im voreingestellten Verzeichnis befindet. Sie können die beiden ersten Beispiele wie folgt übersetzen:

```
mxmlc helloflexl.mxml
mxmlc helloflex2.mxml
```

Das zweite Kommando liefert die Datei *helloflex2.swf*, die Sie mittels Flash oder in eine HTML-Seite eingebettet und von einem HTTP-Server heruntergeladen, starten können. Wenn der Webbrowser die Flash-Laufzeitumgebung geladen hat, genügt häufig ein Doppelklick auf die *.swf* Datei, um sie im Browser zu staren.

[242] Das Beispiel *helloflex2.swf* liefert die folgende Benutzerschnittstelle im Flash-Player: Abbildung Bei aufwändigeren Anwendungen können Sie die MXML- und die ActionScript-Anweisungen voneinander trennen, indem Sie Funktionen in externen ActionScript-Dateien (Endung *.as*) referenzieren. In einem MXML-Skript verwenden Sie dazu das folgende Element:

```
<mx:Script source=''MyExternalScript.as'' />
```

Diese Anweisung gestattet einer MXML-Komponente eine Funktion aus der Datei *MyExternalScript.as* aufzurufen, als ob sie im MXML-Skript selbst definiert wäre.

23.13.3 MXML und ActionScript

[243] MXML ist eine abgekürzte Schreibweise für ActionScript-Klassen. Zu jedem MXML-Element existiert eine gleichnamige ActionScript-Klasse. Der Flex-Compiler wandelt beim Parsen zunächst die XML-Elemente in ActionScript um, lädt die referenzierten Klassen und übersetzt schließlich ActionScript in SWF.

[244] Sie können eine Flex-Anwendung vollständig in ActionScript schreiben, ohne MXML-Syntax zu verwenden. MXML ist eine komfortable Schnittstelle zu ActionScript. Die Komponenten der Benutzerschnittstelle wie Container und Steuerelemente, werden typischerweise per MXML deklariert, während Ereignisbehandlung und sonstige clientseitige Logik in ActionScript oder Java implementiert wird.

[245] Sie können eigene MXML-Steuerelemente entwickeln und referenzieren, indem Sie eigene ActionScript-Klassen schreiben. Sie können auch existierende MXML-Container und Steuerelemente in einem separaten MXML-Dokument zusammenfassen und von einem anderen MXML-Dokument aus per Tag referenzieren. Auf der Website von Macromedia finden Sie Informationen zu diesem Thema.

23.13.4 Container und Steuerelemente

[246] Der visuelle Kern der Komponentenbibliothek von Flex besteht aus Containern, die für das Layout zuständig sind und Steuerelementen, die in diesen Containern platziert werden. Beispiele für Container sind Darstellungsbereiche (*panels*), vertikale und horizontale Kästen, Kacheln (*tiles*), *accordions*, *divided boxes* und Gitter. Beispiele für Steuerelemente sind Schaltfläche, Textbereiche, Schieberegler, Kalender und *data grids*.

[247] Der Rest dieses Abschnitts ist einer Flex-Anwendung gewidmet, die eine Liste von Audiodateien anzeigt und sortiert. Diese Anwendung führt Container, Steuerelemente und die Anbindung einer Flex-Anwendung an Java vor.

[248] Wir beginnen das MXML-Skript damit, ein `<mx:DataGrid>`-Element (eines der anspruchsvolleren Steuerelemente) in einem `<mx:Panel>`-Element (Container) zu deklarieren:

```
<?xml version=''1.0'' encoding=''utf-8''?>
<mx:Application
  xmlns:mx=''http://www.macromedia.com/2003/mxml''
  backgroundColor=''#B9CAD2'' pageTitle=''Flex Song Manager''
  initialize=''getSongs()''>
  <mx:Script source=''songScript.as'' />
  <mx:Style source=''songStyles.css''/>
  <mx:Panel id=''songListPanel''
    titleStyleDeclaration=''headerText''
    title=''Flex MP3 Library''>
    <mx:HBox verticalAlign=''bottom''>
      <mx:DataGrid id=''songGrid''
        cellPress=''selectSong(event)'' rowCount=''8''>
        <mx:columns>
          <mx:Array>
            <mx:DataGridColumn columnName=''name''
              headerText=''Song Name'' width=''120'' />
            <mx:DataGridColumn columnName=''artist''
              headerText=''Artist'' width=''180'' />
            <mx:DataGridColumn columnName=''album''
              headerText=''Album'' width=''160'' />
```

```
            </mx:Array>
          </mx:columns>
        </mx:DataGrid>
        <mx:VBox>
          <mx:HBox height=''100'' >
            <mx:Image id=''albumImage'' source=''''
              height=''80'' width=''100''
              mouseOverEffect=''resizeBig''
              mouseOutEffect=''resizeSmall'' />
            <mx:TextArea id=''songInfo''
              styleName=''boldText'' height=''100%'' width=''120''
              vScrollPolicy=''off'' borderStyle=''none'' />
          </mx:HBox>
          <mx:MediaPlayback id=''songPlayer''
            contentPath=''''
            mediaType=''MP3''
            height=''70''
            width=''230''
            controllerPolicy=''on''
            autoPlay=''false''
            visible=''false'' />
        </mx:VBox>
      </mx:HBox>
      <mx:ControlBar horizontalAlign=''right''>
        <mx:Button id=''refreshSongsButton''
          label=''Refresh Songs'' width=''100''
          toolTip=''Refresh Song List''
          click=''songService.getSongs()'' />
      </mx:ControlBar>
    </mx:Panel>
    <mx:Effect>
      <mx:Resize name=''resizeBig'' heightTo=''100''
        duration=''500''/>
      <mx:Resize name=''resizeSmall'' heightTo=''80''
        duration=''500''/>
    </mx:Effect>
    <mx:RemoteObject id=''songService''
      source=''gui.flex.SongService''
      result=''onSongs(event.result)''
      fault=''alert(event.fault.faultstring, 'Error')''>
      <mx:method name=''getSongs''/>
    </mx:RemoteObject>
  </mx:Application>
```

[249] Das `<mx:DataGrid>`-Element enthält geschachtelte Elemente, welche die Spaltenköpfe deklarieren. Hat das MXML-Element eines Steuerelementes ein Attribut oder enthält es selbst Elemente, dann wissen Sie, daß dieses MXML-Element zu einer Eigenschaft, einem Ereignis oder einem referenzierten Objekt der unterliegenden ActionScript-Klasse gehört. Das `<mx:DataGrid>`-Element hat beispielsweise ein `id`-Attribut mit dem Wert `songGrid`. Somit können ActionScript und die Elemente des MXML-Skripts das Gitter programmatisch referenzieren, indem sie `songGrid` als Variablennamen verwenden. Das `<mx:DataGrid>`-Element exponiert viel mehr Eigenschaften, als im obigen Beispiel gezeigt sind. Sie finden die vollständige API-Dokumentation der MXML-Container und Steuerelemente unter der Webadresse http://livedocs.macromedia.com/flex/is/asdocs_en/index.html.

[250] Dem `<mx:DataGrid>`-Element folgt ein `<mx:VBox>`-Element, welches ein `<mx:Image>`-Element

(Vorderseite des Albums mit Informationen zum Musiktitel) und ein `<mx:MediaPlayback>`-Element (Abspielen der *.mp3* Dateien) enthält. Das Beispiel streams the content, um die Größe der übersetzten *.swf* Datei zu begrenzen. Wenn Sie Graphiken, Audio- und Videodateien in eine Flex-Anwendung einbetten statt sie bei Bedarf herunterzuladen, werden die Dateien in die übersetzte *.swf* Datei integriert und zusammen mit der Benutzerschnittstelle ausgeliefert.

[251] Der Flash-Player enthält eingebettete Codecs zum Abspielen vieler verschiedener Audio- und Videoformate. Flash und Flex unterstützen die meisten gängigen Graphikformate und Flex kann außerdem Scalable Vector Graphics (SVG) Dateien in *.swf* Dateien umwandeln, die sich in Flex-Clients einbetten lassen.

23.13.5 Effekte und Formatierungsmöglichkeiten

[252] Der Flash-Player stellt Graphiken mit Hilfe von Vektoren dar, wodurch zur Laufzeit sehr ausdrucksstarke Transformationen ausgeführt werden können. Flex-Effekte bieten einen Vorgeschmack dieser Art von Animation. Effekte sind Transformationen, die Sie auf Steuerelemente und Container in MXML-Syntax anwenden können.

[253] Das `<mx:Effect>`-Element in *song.mxml* hat zwei Auswirkungen: Das erste geschachtelte `<mx:Resize>`-Element vergrößert eine Abbildung dynamisch, wenn der Mauszeiger in der Abbildung steht, das zweite Elemente verkleinert die Abbildung wieder, wenn der Mauszeiger die Abbildung verläßt. Beide Effekte sind mit den Mausereignissen des `<mx:Image>`-Elementes mit dem Attribut `id=''albumImage''` verknüpft.

[254] Flex liefert auch Effekte für häufige Animationen wie Übergänge (*transitions*), Verwischen (*wipes*) und modulierte Alphakanäle. Neben den eingebauten Effekten unterstützt Flex die Drawing-API von Flash für wahrhaft erfinderische Animationen. Eine tiefere Darstellung dieses Themas stützt sich auf Graphikdesign und Animation und geht über den Rahmen dieses Abschnitts hinaus.

[255] Standardformatierungsmöglichkeiten sind verfügbar, da Flex Cascading Style Sheets (CSS) unterstützt. Wenn Sie eine *.css* Datei mit einem MXML-Skript verknüpfen, gehorchen die Flex-Komponenten diesen Formatierungsangaben. Die Datei *songStyles.css* enthält die folgenden CSS-Einstellungen für unser Beispiel:

```
.headerText {
  font-family: Arial, ''_sans'';
  font-size: 16;
  font-weight: bold;
}

.boldText {
  font-family: Arial, ''_sans'';
  font-size: 11;
  font-weight: bold;
}
```

Diese *.css* Datei wird über das `<mx:Style>`-Element im MXML-Skript importiert und in der Anwendung genutzt. Nach dem Importieren der *.css* Datei können die dortigen Deklarationen auf die Flex-Komponenten im MXML-Skript angewendet werden. Beispielsweise verwendet das `<mx:TextArea>`-Element mit dem Attribut `id=''songInfo''` die `boldText`-Deklaration aus der *.css* Datei.

23.13.6 Ereignisse

[256] Eine Benutzerschnittstelle ist ein Zustandsautomat (*state machine*). Sie führt Aktionen aus, wenn Zustandsänderungen eintreten. Bei Flex werden Zustandänderungen durch Ereignisse ausgelöst. Die Klassenbibliothek von Flex enthält eine Vielzahl von Steuerelementen die eine große Bandbreite von Ereignissen erfassen, darunter alle Arten von Mausbewegungen und Tastaturereignissen.

[257] Das `click`-Attribut des `<mx:Button>`-Elementes (Schaltfläche) ist ein Beispiel für ein Ereignis bei diesem Steuerelement. Der Attributwert von `click` kann eine Funktion oder ein kleines Skript sein. In unserem MXML-Skript enthält das `<mx:ControlBar>`-Element ein `<mx:Button>`-Element mit dem Attribut `id=''refreshSongsButton''`, um die Liste der Musiktitel zu aktualisieren. Wird die Schaltfläche betätigt, so wird die Funktion `songService.getSongs()` aufgerufen. Das `click`-Ereignis des `<mx:Button>`-Elementes wird per `<mx:RemoteObject>`-Element auf die entsprechende Java-Methode abgebildet.

23.13.7 Anbindung an Java

[258] Das `<mx:RemoteObject>`-Element am unteren Ende des MXML-Skripts konfiguriert die Verbindung mit der externen Java-Klasse `gui.flex.SongService`. Der Flex-Client ruft die `getSongs()`-Methode der Klasse `SongService` auf, um die Daten für das `<mx:DataGrid>`-Element anzufordern. Zu diesem Zweck muß sich `SongService` wie ein Dienst verhalten, also wie ein Endpunkt mit dem der Client Nachrichten austauschen kann. Das `source`-Attribut des `<mx:RemoteObject>`-Elementes gibt den vollqualifizierten Namen der Java-Klasse an und legt eine ActionScript Rückruffunktion namens `onSongs()` fest, die aufgerufen wird, wenn die Java-Methode von ihrer Verarbeitung zurückkehrt. Das Unterelement `<mx:Method>` deklariert die Methode `getSongs()` und macht die Java-Methode somit in der Flex-Anwendung erreichbar.

[259] Aufrufe von Diensten kehren bei Flex asynchron zurück, nachdem die Rückruffunktionen durch Ereignisse benachrichtigt worden sind. Das `<mx:RemoteObject>`-Element zeigt ein Dialogfenster mit einer Warnung an, wenn ein Fehler auftritt.

[260] Nun kann die `getSongs()`-Methode aufgerufen werden:

```
songService.getSongs();
```

Aufgrund der MXML-Konfiguration wird hierdurch die Methode `getSongs()` in der Klasse `SongService` aufgerufen:

```
//: gui/flex/SongService.java
package gui.flex;
import java.util.*;

public class SongService {
    private List<Song> songs = new ArrayList<Song>();
    public SongService() { fillTestData(); }
    public List<Song> getSongs() { return songs; }
    public void addSong(Song song) { songs.add(song); }
    public void removeSong(Song song) { songs.remove(song); }
    private void fillTestData() {
        addSong(new Song(''Chocolate'', ''Snow Patrol'',
                         ''Final Straw'', ''sp-final-straw.jpg'',
                         ''chocolate.mp3''));
        addSong(new Song(''Concerto No. 2 in E'', ''Hilary Hahn'',
                         ''Bach: Violin Concertos'', ''hahn.jpg'',
```

```
                            ''bachviolin2.mp3''));
        addSong(new Song(''Round Midnight'', ''Wes Montgomery'',
                            ''The Artistry of Wes Montgomery'',
                            ''wesmontgomery.jpg'', ''roundmidnight.mp3''));
    }
} ///:~
```

Ein `Song`-Objekt ist lediglich ein Datencontainer (JavaBean):

```
//: gui/flex/Song.java
package gui.flex;

public class Song implements java.io.Serializable {
    private String name;
    private String artist;
    private String album;
    private String albumImageUrl;
    private String songMediaUrl;
    public Song() {}
    public Song(String name, String artist, String album,
                String albumImageUrl, String songMediaUrl) {
        this.name = name;
        this.artist = artist;
        this.album = album;
        this.albumImageUrl = albumImageUrl;
        this.songMediaUrl = songMediaUrl;
    }
    public void setAlbum(String album) { this.album = album;}
    public String getAlbum() { return album; }
    public void setAlbumImageUrl(String albumImageUrl) {
        this.albumImageUrl = albumImageUrl;
    }
    public String getAlbumImageUrl() { return albumImageUrl;}
    public void setArtist(String artist) {
        this.artist = artist;
    }
    public String getArtist() { return artist; }
    public void setName(String name) { this.name = name; }
    public String getName() { return name; }
    public void setSongMediaUrl(String songMediaUrl) {
        this.songMediaUrl = songMediaUrl;
    }
    public String getSongMediaUrl() { return songMediaUrl; }
} ///:~
```

Nach dem Initialisieren der Anwendung oder wenn Sie die Schaltfläche mit dem Attribut `id=''refreshSongsButton''` betätigen, wird die Java Methode `getSongs()` und nach ihrer Rückkehr die ActionScript-Funktion `onSongs(event.result)` aufgerufen, um das `<mx:DataGrid>`-Element mit dem Attribut `id=''songGrid''` zu füllen.

[261] Das mit Hilfe des `<mx:Script>`-Elementes im MXML-Skript eingebundene Skript *songScript.as* enthält die ActionScript-Funktionen:

```
//: gui/flex/songScript.as
function getSongs() {
  songService.getSongs();
}

function selectSong(event) {
```

```
    var song = songGrid.getItemAt(event.itemIndex);
    showSongInfo(song);
  }

  function showSongInfo(song) {
    songInfo.text = song.name + newline;
    songInfo.text += song.artist + newline;
    songInfo.text += song.album + newline;
    albumImage.source = song.albumImageUrl;
    songPlayer.contentPath = song.songMediaUrl;
    songPlayer.visible = true;
  }

  function onSongs(songs) {
    songGrid.dataProvider = songs;
  } ///:~
```

Das Ereignisattribut `cellPress` des `<mx:DataGrid>`-Elementes in unserem MXML-Skript definiert, was beim Selektieren einer Gitterzelle geschieht:

```
cellPress = ''selectSong(event)'';
```

Klickt der Benutzer im `<mx:DataGrid>`-Element einen Musiktitel an, so wird die `selectSong()`-Funktion im obigen ActionScript aufgerufen.

23.13.8 Datenmodelle und Datenbindung

[262] Ein Steuerelement kann Dienste direkt ansprechen und die ereignisgetriebenen Rückruffunktion von ActionScript gestattet Ihnen, das visuelle Steuerelement programmatisch zu aktualisieren, wenn ein Dienst Daten zurückgibt. Ein Skript zur Aktualisierung eines Steuerelementes kann einfach sein, aber auch langatmig und sperrig werden. Die Funktionalität tritt so häufig auf, daß Flex dieses Verhalten automatisch behandelt, nämlich über die sogenannte Datenbindung (*data binding*).

[263] In ihrer einfachsten Form gestattet Datenbindung Steuerelementen, Daten unmittelbar zu referenzieren, im Gegensatz zu „Zwischenanweisungen“ (*glue code*), um die Daten an das Steuerelement zu übergeben. Wurden die Daten aktualisiert, so wird das Steuerelement, welches diese Daten referenziert, automatisch aktualisiert, ohne daß ein programmatischer Eingriff erforderlich ist. Die Flex-Infrastruktur reagiert korrekt auf Ereignisse, die veränderte Daten anzeigen und aktualisiert alle an diese Daten gebundenen Steuerelemente.

[264] Ein einfaches Beispiel für die Syntax der Datenbindung:

```
<mx:Slider id=''mySlider''/>
<mx:Text text=''{mySlider.value}'' />
```

Datenbindung wird implementiert, indem Sie Referenzen in geschweife Klammern (`{}`) setzen. Der gesamte Inhalt zwischen den geschweiften Klammern gilt als Ausdruck, der von Flex ausgewertet wird.

[265] Der Wert des ersten Steuerelementes, eines `<mx:Slider>`-Elementes (Schieberegler), wird vom zweiten Steuerelement angezeigt, einem `<mx:Text>`-Element (Texteingabefeld). Wenn sich der Wert des Schiebereglers ändert, wird die `text`-Eigenschaft des `<mx:Text>`-Element automatisch aktualisiert. Dadurch muß der Entwickler das Änderungsereignis des `<mx:Slider>`-Elementes nicht behandeln, um das `<mx:Text>`-Element aktualisieren zu können.

[266] Einige Steuerelemente, wie `<mx:Tree>` und `<mx:DataGrid>` sind anspruchsvoller und verfügen über eine `dataprovider`-Eigenschaft, um die Anbindung an Datenkollektionen zu vereinfachen. Die

ActionScript-Funktion `onSongs()` zeigt, wie die `SongService`-Methode `getSongs()` an die `dataprovider`-Eigenschaft des `<mx:DataGrid>`-Elementes gebunden wird. Die `onSongs()`-Funktion ist die Rückruffunktion, die ActionScript aufruf, wenn die Java-Methode zurückkehrt (siehe Deklaration im `<mx:RemoteObject>`-Element des MXML-Skripts).

[267] Bei einer anspruchsvolleren Anwendung mit komplexerem Datenmodell, etwa einer Enterprise-Applikation mit Transferobjekten oder einer messaging application mit Daten, die einem komplizierten Schema gehorchen, ist eventuell eine weiterreichende Entkopplung der Datenquelle von der Steuerelementen erwünscht. Bei Flex erreichen wir diese Entkopplung durch Deklarieren eines „Modellobjektes“, genauer eines generischen MXML-Containers für Daten. Das Datenmodell enthält keinerlei Logik. Es ist viel mehr das Äquivalent des Transferobjektes aus dem Entwicklungsumfeld der Enterprise-Applikationen beziehungsweise anderer Programmiersprachen. Ein solches Datenmodell wird einerseits mit der Steuerkomponente verbunden und andererseits werden seine Eigenschaften zugleich mit den Ein- und Ausgaben der Dienste verknüpft. Die Datenquellen (die Dienste) sind somit von den visuellen Verbrauchern der Daten entkoppelt und das Entwurfsmuster *Model-View-Controller* wird unterstützt. Bei großen und komplexen Anwendungen ist der anfängliche Komplexitätszuwachs durch Einführen eines Datenmodells häufig ein geringer Preis im Vergleich mit dem Gewinn durch eine klar entkoppelte Implementierung auf der Grundlage des Entwurfsmusters *Model-View-Controller*.

[268] Neben Java-Objekten kann Flex auch mit SOAP-basierten Webservices (`<mx:WebService>`) und RESTful HTTP-basierten Diensten (`<mx:HttpService>`) komnunizieren. Der Zugriff auf alle Dienste unterliegt der Authorisierung.

23.13.9 Übersetzen und Deployment der Beispielanwendung

[269] Die beiden ersten Beispiele ließen sich ohne die Option `-flexlib` auf der Kommandozeile übersetzen. Zum Übersetzen des Beispiels *songs.mxml* müssen Sie dagegen per `-flexlib` den Pfad zur Datei *flex-config.xml* angeben. Bei meiner Installation lautet das Kommando wie folgt (Sie müssen den Pfad an Ihre Konfiguration anpassen. Das Kommando ist einzeilig.):

```
mxmlc -flexlib C:/''Program Files''/Macromedia/Flex/jrun4/servers/de-
                fault/flex/WEB-INF/flex songs.mxml
```

Dieses Kommando übersetzt die Anwendung in eine *.swf* Datei, die Sie mit Ihrem Webbrowser aufrufen können. Die Quelltextdistribution des Buches enthält keine *.mp3* oder *.jpg* Dateien, das heißt Sie sehen nur das Gerüst, wenn Sie die Anwendung aufrufen.

[270] Sie müssen darüber hinaus einen Server konfigurieren, um die Java-Dateien der Flex-Anwendung ansprechen zu können. Die Probeversion von Flex enthält den Server „JRun“, den Sie nach der Flex-Installation über die Menüs oder von der Kommandozeile aus starten können:

```
jrun -start default
```

[271] Sie können verifizieren, daß der Server erfolgreich gestartet wurde, indem Sie in einem Webbrowser die Adresse *http://localhost:8700/samples* aufrufen und die verschiedenen Beispiele betrachten (ein guter Einstiegspunkt, um sich mit den Fähigkeiten von Flex vertraut zu machen).

[272] Anstelle der Übersetzung der Anwendung auf der Kommandozeile, können Sie den Vorgang an den Server delegieren. Deponieren Sie die Quelltextdateien im Verzeichnis *jrun4/servers/default/flex* und rufen Sie die Anwendung in einem Browser mit der folgenden Adresse auf: *http://localhost:8700-/flex/songs.mxml*.

[273] Damit die Anwendung läuft, müssen Sie sowohl Java als auch Flex konfigurieren.

[274] Konfiguration von Java: Deponieren Sie die *.class* Dateien zu *Song.java* und *SongService.java* im Verzeichnis *WEB-INF/classes*. Gemäß der J2EE-Spezifikation werden Klassen innerhalb einer *.war* Datei in diesem Unterverzeichnis platziert. Alternativ können Sie die Dateien per `jar` archivieren und die *.jar* Datei im Verzeichis *WEB-INF/lib* ablegen. Für den JRun-Server lauten die Pfade *jrun4/servers/default/flex/WEB-INF/classes/gui/flex/Song.class* und *jrun4/servers/default/flex/WEB-INF/classes/gui/flex/SongService.class*. Auch die Graphiken und *.mp3* Dateien müssen in der Webapplikation verfügbar sein.

[275] Konfiguration von Flex: Aus Sicherheitsgründen kann Flex nicht auf Java-Objekte zugreifen, außer Sie erteilen die Erlaubnis dazu in der Konfigurationsdatei *flex-config.xml*. Für den JRun-Server lautet der Pfad zu dieser Datei *jrun4/servers/default/flex/WEB-INF/flex/flex-config.xml*. Suchen Sie den Eintrag `<remote-objects>` und darin das Unterelement `<whitelist>`. Dort finden Sie die folgende Notiz:

> For security, the whitelist is locked down by default. Uncomment the source element below to enable access to all classes during development.
>
> We strongly recommend not allowing access to all source files in production, since this exposes Java and Flex system classes. <source>*</source>

[276] Entfernen Sie die Kommentarzeichen des `<source>`-Einrags, so daß die Zeile `<source>*</source>` lautet. Die Bedeutung dieses und der übrigen Einträge finden Sie in der Flex-Dokumentation.

Übungsaufgabe 38: (3) Probieren Sie das einfache Beispiel für Datenbindung auf Seite 1099 aus. ■

Übungsaufgabe 39: (4) Die Quelltextdistribution zu diesem Buch enthält die in *SongService.java* genannten *.mp3* und *.jpg* Dateien nicht. Nehmen Sie einige Ihre eigenen *.mp3* und *.jpg* Dateien, ändern Sie *SongService.java* entsprechend, installieren Sie die Flex-Probeversion und übersetzen Sie die Beispielanwendung. ■

23.14 Das Standard Widget Toolkit (SWT)

[277] Der Ansatz von Swing besteht darin, alle Komponenten pixelweise aufzubauen, um jede gewünschte Komponente anbieten zu können, unabhängig davon, ob das unterliegende Betriebssystem diese Komponente zur Verfügung stellt oder nicht (siehe ~~früher in diesem Kapitel~~). Das Standard Widget Toolkit (SWT) des Eclipse-Projektes wählt den Mittelweg, native Komponenten zu verwenden, sofern sie vom Betriebssystem definiert werden und andernfalls künstlich herstellen. Die SWT-Anwendung wirkt dadurch auf den Benutzer wie eine native Anwendung und ist häufig deutlich performanter als ein äquivalentes Swing-Programm. Verglichen mit Swing, hat SWT außerdem ein einfacheres Programmiermodell, was bei vielen Anwendungen erstrebenswert ist.[11]

[278] Da SWT das native Betriebssystem einbezieht, um so viel Arbeit wie möglich auszulagern, kann es automatisch von Eigenschaften und Fähigkeiten des Betriebssystems Gebrauch machen, die Swing nicht zur Verfügung stehen. Beispielsweise implementiert Windows sogenannte Teilpixelrasterung (*subpixel rendering*), wodurch Zeichensätze auf LCD-Bildschirmen schärfer dargestellt werden.

[279] Sie können sogar Applets mit SWT entwickeln.

[280] Dieser Abschnitt ist nicht als umfassende Einführung in SWT gedacht, sondern soll Ihnen eine Anregung und einen Einblick vermitteln, um die Unterschiede zwischen SWT und Swing deutlicher erkennen zu können. Sie werden entdecken, daß es viele SWT-Widgets gibt, die sich alle relativ

[11] Chris Grindstaff war sehr hilfsbereit beim Umschreiben der SWT-Beispiele und hat mir viele Informationen über das Standard Widget Toolkit zur Verfügung gestellt.

einfach verwenden lassen. Die Einzelheiten entnehmen Sie der vollständigen Dokumentation und den vielen Beispielen unter der Webadresse *http://www.eclipse.org*. Es gibt bereits eine Anzahl von Büchern über das Programmieren mit SWT und weitere werden folgen.

23.14.1 Installieren des Standard Widget Toolkits

[281] Vor dem Programmieren mit SWT müssen Sie die SWT-Bibliothek des Eclipse-Projekts herunterladen und installieren. Wählen Sie unter der Webadresse *http://www.eclipse.org/swt* ein stabiles Release für Ihr Betriebssystem und einen Server aus. Folgen Sie den Verweisen zur aktuellen Eclipse-Version und suchen Sie dort nach einer komprimierten Datei, deren Name mit „swt“ beginnt und den Namen Ihres Betriebssystems angibt (zum Beispiel „win32“). Die komprimierte Datei enthält ein Archiv namens *swt.jar*, welches Sie am einfachsten installieren, indem Sie es in Ihrem *jre/lib/ext*-Verzeichnis ablegen (auf diese Weise brauchen Sie Ihren Klassenpfad nicht anzupassen). Wenn Sie die SWT-Bibliothek auspacken, stoßen Sie unter Umständen auf weitere Dateien, die Sie an verschiedenen Stellen Ihres Betriebssystems installieren müssen. Die Win32-Distribution enthält beispielsweise *.dll* Dateien, die Sie in Ihrem Java-Bibliothekspfad (`java.library.path`) installieren müssen. (Der Bibliothekspfad stimmt in der Regel Ihrer `$PATH`-Variablen überein. Sie können aber das Hilfsprogramm *object/ShowProperties.java* aus Abschnitt 3.7, Seite 67 benutzen, um den Wert Ihrer `java.library.path`-Eigenschaft anzuzeigen). Anschließend sollten Sie in der Lage sein, SWT-Anwendungen übersetzen und ausführen, wie jedes andere Java-Programm.

[282–283] Die SWT-Dokumentation muß separat heruntergeladen werden. Alternativ können Sie die Eclipse-Laufzeitumgebung installieren, die sowohl die SWT-Bibliothek als auch die SWT-Dokumentation enthält und die Dokumentation anschließend über das Hilfesystem von Eclipse lesen.

23.14.2 „Hello SWT“

[284] Wir beginnen mit einem „Hello World“-Beispiel:

```
//: swt/HelloSWT.java
// {Requires: org.eclipse.swt.widgets.Display; You must
// install the SWT library from http://www.eclipse.org }
import org.eclipse.swt.widgets.*;

public class HelloSWT {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Hi there, SWT!"); // Title bar
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} ///:~
```

Nach dem Herunterladen der Quelltextdistribution zu diesem Buch werden Sie entdecken, daß die Direktive `// {Requires: ...` eine notwendige Bedingung für die Übersetzung der *.java* Dateien im Verzeichnis *swt* der Quelltextdistribution beschreibt, die zusammen mit *build.xml* von Ant ausgewertet wird. Alle Dateien, die das Package `org.eclipse.swt` importieren, setzen voraus, daß die SWT-Bibliotek von *http://www.eclipse.org* installiert ist.

[285] Die Klasse `Display` repräsentiert die Verbindung zwischen SWT und dem unterliegenden Betriebssystem, ist also ein Teil der Brücke zwischen SWT und Betriebssystem. Die Klasse `Shell` repräsentiert das Hauptfenster, in dem alle übrigen Komponenten angelegt werden. Das Argument der `setText()`-Methode liefert die Beschriftung der Titelleiste des Hauptfensters. Damit das Fenster und somit die ganze Anwendung angezeigt werden, müssen Sie die `open()`-Methode des `Shell`-Objektes aufrufen.

[287] Während Swing die Ereignisbehandlerschleife vor dem Programmierer versteckt, sind Sie bei SWT gezwungen, die Schleife explizit auszuschreiben. Am oberen Ende des Schleifenkörpers prüfen Sie, ob das Hauptfenster geschlossen wurde. Beachten Sie, daß Sie an dieser Stelle die Möglichkeit haben, Anweisungen zum Aufräumen einzusetzen. Das bedeutet allerdings, daß der `main`-Thread die Benutzerschnittstelle steuert. Bei Swing wird hinter den Kulissen ein separater Ereignisbehandlerthread erzeugt, während sich bei SWT der `main`-Thread um die Benutzerschnittstelle kümmert. Da es standardmäßig nur einen und nicht zwei Threads gibt, ist es etwas unwahrscheinlicher, daß Sie die Benutzerschnittstelle [to clobber: verprügeln] mit Threads.

[288] Beachten Sie, daß Sie sich nicht darum kümmern müssen, dem user interface thread Aufgaben zu übergeben, wie bei Swing. SWT kümmert sich nicht nur an Ihrer Stelle darum, sondern wirft eine Ausnahme aus, wenn Sie versuchen, nicht über den richtigen Thread auf ein Widget zuzugreifen. Falls Sie dennoch neue Threads abspalten müssen, um Operationen mit langer Laufzeit zu verarbeiten, können Sie Änderungen nach wie vor auf dieselbe Weise abwickeln, wie bei Swing. SWT stellt über die Klasse `Display` drei Methoden für diesen Zweck zur Verfügung: `asyncExec(Runnable)`, `syncExec(Runnable)` und `timerExec(Runnable)`.

[289] Der `main`-Thread ruft an dieser Stelle die `readAndDispatch()`-Methode des `Display`-Objektes auf (das heißt es kann pro Anwendung nur ein `Display`-Objekt existieren). Die Methode `readAndDispatch()` gibt `true` zurück, wenn die Ereigniswarteschlange noch Ereignisse zur Verarbeitung enthält. In diesem Fall rufen Sie die Methode sofort noch einmal auf. Andernfalls rufen Sie die `sleep()`-Methode des `Display`-Objektes auf und prüfen die Warteschlange nach einer kurzen Wartezeit erneut.

[290] Sie müssen beim Programmende die `dispose()`-Methode Ihres `Display`-Objektes explizit aufrufen. Es ist bei SWT häufig erforderlich, Resourcen explizit freizugeben, da es sich in der Regel betriebssystemseitige Resourcen handelt, die andernfalls aufgebraucht werden können.

[291] Das folgende Programm zeigt durch Erzeugen mehrerer `Shell`-Objekte, daß die Klasse `Shell` das Hauptfenster repräsentiert:

```
//: swt/ShellsAreMainWindows.java
import org.eclipse.swt.widgets.*;

public class ShellsAreMainWindows {
    static Shell[] shells = new Shell[10];
    public static void main(String[] args) {
        Display display = new Display();
        for(int i = 0; i < shells.length; i++) {
            shells[i] = new Shell(display);
            shells[i].setText("Shell #" + i);
            shells[i].open();
        }
        while(!shellsDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
    static boolean shellsDisposed() {
```

```
        for(int i = 0; i < shells.length; i++)
            if(shells[i].isDisposed())
                return true;
        return false;
    }
} ///:~
```

Wenn Sie das Programm aufrufen, bekommen Sie zehn Hauptfenster. Das Programm ist so geschrieben, daß alle Fenster geschlossen werden, wenn Sie eines schließen.

[292–293] Auch SWT verwendet Layoutmanager, zwar andere als Swing, aber die Idee ist dieselbe. Das nächste Beispiel ist etwas komplizierter und setzt die Ausgabe der statischen `System`-Methode `getProperties()` ins Hauptfenster ein:

```
//: swt/DisplayProperties.java
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
import java.io.*;

public class DisplayProperties {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText("Display Properties");
        shell.setLayout(new FillLayout());
        Text text = new Text(shell, SWT.WRAP | SWT.V_SCROLL);
        StringWriter props = new StringWriter();
        System.getProperties().list(new PrintWriter(props));
        text.setText(props.toString());
        shell.open();
        while(!shell.isDisposed())
            if(!display.readAndDispatch())
                display.sleep();
        display.dispose();
    }
} ///:~
```

Bei SWT muß jedes Widget ein Elternobjekt des allgemeinen Typs `Composite` haben und Sie müssen dem Konstruktor des Widgets eine Referenz auf das Elternobjekt als erstes Objekt übergeben. Das erste Argument des Konstruktors der Klasse `Text` ist beispielsweise `shell`. Nahezu jeder Konstruktor akzeptiert auch einen Schalter als Argument, über den Sie beliebig viele Stildirektiven übergeben können, je nachdem, welche Direktiven das entsprechende Widget akzeptiert. Zwei oder mehr Direktiven werden bitweise `OR`-verknüpft (siehe oben).

[294] Das obige `Text`-Objekt ist so konfiguriert, daß es seinen Inhalt umbricht und bei Bedarf automatisch einen senkrechten Rollbalken hinzufügt. Die werden feststellen, daß SWT sehr konstruktorbasiert ist, das heißt viele Attribute von Widgets können nur schwierig oder überhaupt nicht bewertet werden, außer per Konstruktor. Lesen Sie stets in der Dokumentation nach, welche Schalter der Konstruktor eines Widgets akzeptiert. Beachten Sie, daß bei einigen Konstruktoren ein Schalter zwar erforderlich, in der Liste der akzeptierten Schalter aber nicht dokumentiert ist. Dies gestattet zukünftige Erweiterungen, ohne die Schnittstelle ändern zu müssen.

23.14.3 Die Hilfsklasse SWTConsole

[295] Beachten Sie, bevor wir fortfahren, daß es, analog wie bei Swing-Anwendungen, einige Anweisungen gibt, die Sie bei jeder SWT-Anwendung schreiben. Sie erzeugen zum Beispiel bei SWT stets ein `Display`-Objekt, übergeben dem Konstruktor der Klasse `Shell` ein Referenz auf dieses Objekt, legen eine Ereignisbehandlerschleife für die `readAndDispatch()`-Methode an und so weiter. Es gibt natürlich Spezialfälle, bei denen Sie von diesem Schema abweichen, es aber so häufig vor, damit sich eine Hilfsklasse wie `net.mindview.util.SwingConsole` lohnt (siehe Unterabschnitt 23.2.1).

[296] Wir setzen voraus, daß jede SWT-Anwendung dem folgenden Interface genügt:

```
//: swt/util/SWTApplication.java
package swt.util;
import org.eclipse.swt.widgets.*;

public interface SWTApplication {
    void createContents(Composite parent);
} ///:~
```

Die `createContents()`-Methode der Anwendung erwartet eine Referenz vom Typ `Composite` (`Shell` ist von `Composite` abgeleitet) und muß den Inhalt ihres Hauptfensters vollständig in `createContents()` aufbauen. Die statische `SWTConsole`-Methode `run()` ruft an der richtigen Stelle die Methode `createContents()` auf, legt die Abmessungen des Hauptfensters anhand der beim Aufruf der `run()`-Methode übergebenen Argumente fest, öffnet das Hauptfenster, steuert die Ereignisbehandlerschleife und schließt das Hauptfenster bei Programmende:

```
//: swt/util/SWTConsole.java
package swt.util;
import org.eclipse.swt.widgets.*;

public class SWTConsole {
    public static void
        run(SWTApplication swtApp, int width, int height) {
        Display display = new Display();
        Shell shell = new Shell(display);
        shell.setText(swtApp.getClass().getSimpleName());
        swtApp.createContents(shell);
        shell.setSize(width, height);
        shell.open();
        while(!shell.isDisposed()) {
            if(!display.readAndDispatch())
                display.sleep();
        }
        display.dispose();
    }
} ///:~
```

Die Methode `createContents()` trägt außerdem den Namen der Klasse, die das Interface *`SWTApplication`* implementiert, in die Titelleiste des Hauptfensters ein und konfiguriert die Breite und Höhe des Hauptfensters. Das folgende Beispiel ist eine Abwandlung des Beispiels *DisplayProperties.java* von Seite 1104 und verwendet die Hilfsklasse `SWTConsole`, um die Umgebungseigenschaften anzuzeigen:

```
//: swt/DisplayEnvironment.java
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.layout.*;
```

```
import java.util.*;

public class DisplayEnvironment implements SWTApplication {
    public void createContents(Composite parent) {
        parent.setLayout(new FillLayout());
        Text text = new Text(parent, SWT.WRAP | SWT.V_SCROLL);
        for(Map.Entry entry: System.getenv().entrySet()) {
            text.append(entry.getKey() + ": " + entry.getValue() + "\n");
        }
    }
    public static void main(String[] args) {
        SWTConsole.run(new DisplayEnvironment(), 800, 600);
    }
} ///:~
```

Die Hilfsklasse `SWTConsole` gestattet uns die Konzentration auf die interessanten Aspekte eine Anwendung, statt sich immer wiederholender Anweisungen.

Übungsaufgabe 40: (4) Ändern Sie das Beispiel *DisplayProperties.java* (Seite 1104), so daß es die Hilfsklasse `SWTConsole` verwendet. ■

Übungsaufgabe 41: (4) Ändern Sie das Beispiel *DisplayEnvironment.java*, so daß es die Hilfsklasse `SWTConsole` *nicht* verwendet. ■

23.14.4 Menüs

[297] Das folgenden Beispiel demonstriert einfache Menüs, indem es seinen eigenen Quelltext einliest, in einzelne Wörter trennt und aus diesen Wörtern Menüpunkte erzeugt:

```
//: swt/Menus.java
// Fun with menus.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import java.util.*;
import net.mindview.util.*;

public class Menus implements SWTApplication {
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        Menu bar = new Menu(shell, SWT.BAR);
        shell.setMenuBar(bar);
        Set<String> words =
            new TreeSet<String>(new TextFile("Menus.java", "\W+"));
        Iterator<String> it = words.iterator();
        while(it.next().matches("[0-9]+"))
            ; // Move past the numbers.
        MenuItem[] mItem = new MenuItem[7];
        for(int i = 0; i < mItem.length; i++) {
            mItem[i] = new MenuItem(bar, SWT.CASCADE);
            mItem[i].setText(it.next());
            Menu submenu = new Menu(shell, SWT.DROP_DOWN);
            mItem[i].setMenu(submenu);
        }
        int i = 0;
        while(it.hasNext()) {
```

```
                addItem(bar, it, mItem[i]);
                i = (i + 1) % mItem.length;
            }
        }
        static Listener listener = new Listener() {
                public void handleEvent(Event e) {
                    System.out.println(e.toString());
                }
            };
        void addItem(Menu bar, Iterator<String> it, MenuItem mItem) {
            MenuItem item = new MenuItem(mItem.getMenu(),SWT.PUSH);
            item.addListener(SWT.Selection, listener);
            item.setText(it.next());
        }
        public static void main(String[] args) {
            SWTConsole.run(new Menus(), 600, 200);
        }
    } ///:~
```

Eine Menüleiste (`Menu`-Objekt) muß dem Hauptfenster (`Shell`-Objekt) zugeordnet werden. Die `Composite`-Methode `getShell()` fordert eine Referenz auf das Hauptfenster an. Die Hilfsklasse `net.mindview.util.TextFile` wurde in Unterabschnitt 19.7.1 beschrieben und wird hier verwendet, um einen Container vom Typ `TreeSet` mit den Wörter aus *Menus.java* zu füllen (der Containertyp `TreeSet` speichert seine Elemente in sortierter Reihenfolge). Die ersten Elemente sind Zahlen und werden verworfen. Sieben Top-Level-Menüpunkte werden aus der Wörterliste benannt und anschließend Untermenüpunkte angelegt, bis keine Wörter mehr übrig sind.

[298] Als Reaktion auf das Selektieren eines Menüpunktes, gibt der Ereignisbehandler (*`Listener`*) einfach das abgefangene Ereignisobjekt aus, so daß Sie die darin enthaltenen Informationen sehen können. Wenn Sie das Programm aufrufen, sehen Sie, daß die Ausgabe des Ereignisobjektes auch die Beschriftung des Menüpunktes enthält. Sie können das Verhalten des Ereignisbehandlers somit von der Beschriftung abhängig machen. Alternativ können Sie für jedes Menü einen separaten Ereignisbehandler anlegen (sicherer Ansatz im Hinblick auf Internationalisierung).

23.14.5 Dialogbereiche mit Reitern, Schaltflächen und Ereignisse

[299] Das Standard Widget Toolkit bietet eine reichhaltige Auswahl von Komponenten, die im SWT-Jargon als *Widgets* bezeichnet werden. Sehen Sie sich die einfacheren Widgets in der Dokumentation des Packages `org.eclipse.swt.widgets` an. Das Package `org.eclipse.swt.custom` beinhaltet die etwas ausgefalleneren Exemplare.

[300–301] Das folgende Beispiel zeigt einen Ordner oder Karteikasten mit Reitern zur Demonstration einiger einfacher Widgets. Jede Ebene („Dialogbereich“) unter einem Reiter zeigt ein anderes Widget. Sie sehen außerdem, wie `Composite`-Objekte (im Großen und Ganzen das Äquivalent von `JPanel`) erzeugt werden, um Komponenten in Komponenten zu platzieren:

```
//: swt/TabbedPane.java
// Placing SWT components in tabbed panes.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import org.eclipse.swt.browser.*;
```

```
public class TabbedPane implements SWTApplication {
    private static TabFolder folder;
    private static Shell shell;
    public void createContents(Composite parent) {
        shell = parent.getShell();
        parent.setLayout(new FillLayout());
        folder = new TabFolder(shell, SWT.BORDER);
        labelTab();
        directoryDialogTab();
        buttonTab();
        sliderTab();
        scribbleTab();
        browserTab();
    }
    public static void labelTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText(''A Label''); // Text on the tab
        tab.setToolTipText(''A simple label'');
        Label label = new Label(folder, SWT.CENTER);
        label.setText(''Label text'');
        tab.setControl(label);
    }
    public static void directoryDialogTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText(''Directory Dialog'');
        tab.setToolTipText(''Select a directory'');
        final Button b = new Button(folder, SWT.PUSH);
        b.setText(''Select a Directory'');
        b.addListener(SWT.MouseDown, new Listener() {
                public void handleEvent(Event e) {
                    DirectoryDialog dd = new DirectoryDialog(shell);
                    String path = dd.open();
                    if(path != null)
                        b.setText(path);
                }
            });
        tab.setControl(b);
    }
    public static void buttonTab() {
        TabItem tab = new TabItem(folder, SWT.CLOSE);
        tab.setText(''Buttons'');
        tab.setToolTipText(''Different kinds of Buttons'');
        Composite composite = new Composite(folder, SWT.NONE);
        composite.setLayout(new GridLayout(4, true));
        for(int dir : new int[] { SWT.UP, SWT.RIGHT, SWT.LEFT, SWT.DOWN }) {
            Button b = new Button(composite, SWT.ARROW | dir);
            b.addListener(SWT.MouseDown, listener);
        }
        newButton(composite, SWT.CHECK, ''Check button'');
        newButton(composite, SWT.PUSH, ''Push button'');
        newButton(composite, SWT.RADIO, ''Radio button'');
        newButton(composite, SWT.TOGGLE, ''Toggle button'');
        newButton(composite, SWT.FLAT, ''Flat button'');
        tab.setControl(composite);
    }
    private static Listener listener = new Listener() {
            public void handleEvent(Event e) {
```

```
            MessageBox m = new MessageBox(shell, SWT.OK);
            m.setMessage(e.toString());
            m.open();
        }
    };
private static void newButton(Composite composite, int type, String label) {
    Button b = new Button(composite, type);
    b.setText(label);
    b.addListener(SWT.MouseDown, listener);
}
public static void sliderTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText(''Sliders and Progress bars'');
    tab.setToolTipText(''Tied Slider to ProgressBar'');
    Composite composite = new Composite(folder, SWT.NONE);
    composite.setLayout(new GridLayout(2, true));
    final Slider slider = new Slider(composite, SWT.HORIZONTAL);
    final ProgressBar progress = new ProgressBar(composite, SWT.HORIZONTAL);
    slider.addSelectionListener(new SelectionAdapter() {
            public void widgetSelected(SelectionEvent event) {
                progress.setSelection(slider.getSelection());
            }
        });
    tab.setControl(composite);
}
public static void scribbleTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText(''Scribble'');
    tab.setToolTipText(''Simple graphics: drawing'');
    final Canvas canvas = new Canvas(folder, SWT.NONE);
    ScribbleMouseListener sml= new ScribbleMouseListener();
    canvas.addMouseListener(sml);
    canvas.addMouseMoveListener(sml);
    tab.setControl(canvas);
}
private static class ScribbleMouseListener
    extends MouseAdapter implements MouseMoveListener {
    private Point p = new Point(0, 0);
    public void mouseMove(MouseEvent e) {
        if((e.stateMask & SWT.BUTTON1) == 0)
            return;
        GC gc = new GC((Canvas)e.widget);
        gc.drawLine(p.x, p.y, e.x, e.y);
        gc.dispose();
        updatePoint(e);
    }
    public void mouseDown(MouseEvent e) { updatePoint(e); }
    private void updatePoint(MouseEvent e) {
        p.x = e.x;
        p.y = e.y;
    }
}
public static void browserTab() {
    TabItem tab = new TabItem(folder, SWT.CLOSE);
    tab.setText(''A Browser'');
    tab.setToolTipText(''A Web browser'');
    Browser browser = null;
```

```
            try {
                browser = new Browser(folder, SWT.NONE);
            } catch(SWTError e) {
                Label label = new Label(folder, SWT.BORDER);
                label.setText("Could not initialize browser");
                tab.setControl(label);
            }
            if(browser != null) {
                browser.setUrl("http://www.mindview.net");
                tab.setControl(browser);
            }
        }
        public static void main(String[] args) {
            SWTConsole.run(new TabbedPane(), 800, 600);
        }
    } ///:~
```

Die `createContents()`-Methode wählt einen Layoutmanager und ruft anschließend die Methoden auf, die jeweils einen Dialogbereich mit Reiter erzeugen. Die Beschriftung des Reiters wird per `setText()` definiert (Sie können auf einem Reiter aber auch eine Schaltfläche oder eine Graphik platzieren) und für jeden Reiter ein Tooltip festgelegt. Am Ende jeder Methode wird die `setControl()`-Methode aufgerufen, um die erzeugte Komponente im Dialogbereich unter dem jeweiligen Reiter zu platzieren.

[302–305] Die Methode `labelTab()` zeigt eine einfache Beschriftung. Die `directoryDialogTab()`-Methode beinhaltet eine Schaltfläche, die ein Verzeichnisauswahlfenster (`DirectoryDialog`) öffnet, in dem der Benutzer ein Verzeichnis selektieren kann. Name und Pfad des gewählten Verzeichnisses werden anschließend als Beschriftung der Schaltfläche verwendet. Die Methode `buttonTab()` zeigt verschiedene einfache Schaltflächen. Die `sliderTab()`-Methode wiederholt das Beispiel *Progress.java* aus Unterabschnitt 23.8.19 und verknüpft einen Schieberegler mit einem Fortschrittsbalken. Die Methode `scribbleTab()` zeigt ein einfaches „Malprogramm“, das nur aus wenigen Zeilen besteht. Schließlich demonstriert die Methode `browserTab()` das Leistungsvermögen der SWT-Komponente `Browser`: Einen vollausgestatteten Webbrowser in einer einzigen Komponente.

23.14.6 Graphics

[306–308] Das folgende Beispiel zeigt das Swing-Beispiel *SineWave.java* aus Unterabschnitt 23.8.15, umgeschrieben in SWT:

```
//: swt/SineWave.java
// SWT translation of Swing SineWave.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.layout.*;

class SineDraw extends Canvas {
    private static final int SCALEFACTOR = 200;
    private int cycles;
    private int points;
    private double[] sines;
    private int[] pts;
    public SineDraw(Composite parent, int style) {
        super(parent, style);
        addPaintListener(new PaintListener() {
```

```
                    public void paintControl(PaintEvent e) {
                        int maxWidth = getSize().x;
                        double hstep = (double)maxWidth / (double)points;
                        int maxHeight = getSize().y;
                        pts = new int[points];
                        for(int i = 0; i < points; i++)
                            pts[i] = (int)
                                ((sines[i] * maxHeight / 2 * .95) + (maxHeight / 2));
                        e.gc.setForeground(e.display.getSystemColor(SWT.COLOR_RED));
                        for(int i = 1; i < points; i++) {
                            int x1 = (int)((i - 1) * hstep);
                            int x2 = (int)(i * hstep);
                            int y1 = pts[i - 1];
                            int y2 = pts[i];
                            e.gc.drawLine(x1, y1, x2, y2);
                        }
                    }
                });
            setCycles(5);
        }
        public void setCycles(int newCycles) {
            cycles = newCycles;
            points = SCALEFACTOR * cycles * 2;
            sines = new double[points];
            for(int i = 0; i < points; i++) {
                double radians = (Math.PI / SCALEFACTOR) * i;
                sines[i] = Math.sin(radians);
            }
            redraw();
        }
    }

    public class SineWave implements SWTApplication {
        private SineDraw sines;
        private Slider slider;
        public void createContents(Composite parent) {
            parent.setLayout(new GridLayout(1, true));
            sines = new SineDraw(parent, SWT.NONE);
            sines.setLayoutData(new GridData(SWT.FILL, SWT.FILL, true, true));
            sines.setFocus();
            slider = new Slider(parent, SWT.HORIZONTAL);
            slider.setValues(5, 1, 30, 1, 1, 1);
            slider.setLayoutData(new GridData(SWT.FILL, SWT.DEFAULT, true, false));
            slider.addSelectionListener(new SelectionAdapter() {
                    public void widgetSelected(SelectionEvent event) {
                        sines.setCycles(slider.getSelection());
                    }
                });
        }
        public static void main(String[] args) {
            SWTConsole.run(new SineWave(), 700, 400);
        }
    } ///:~
```

Das Äquivalent von `JPanel` als Zeichenoberfläche ist bei SWT die Klasse `Canvas`. Im Vergleich mit der Swing-Version dieses Programmes sehen Sie, daß die Klasse `SineDraw` nahezu identisch ist. Bei SWT erhalten Sie den Graphikkontext über das Ereignisobjekt, das dem Ereignisbehandler (*Paint-*

Listener) übergeben wird. Bei Swing wird das `Graphics`-Objekt der Methode `paintComponent()` direkt übergeben. Die auf dem *Graphikobjekt* ausgeführten Operationen sind gleich. Die `setCycles()`-Methoden sind identisch, bis auf `repaint()` (Swing) beziehungsweise `redraw()` (SWT). Die `createContents()`-Methode ist etwas aufwändiger als bei der Swing-Version, *to lay things out* und um Schieberegler und Fortschrittsbalken zu konfigurieren, aber auch hier gilt, daß die grundlegenden Schritte im Großen und Ganzen dieselben sind.

23.14.7 Threads und SWT

[309] Obwohl AWT/Swing *single-threaded* sind, können Sie die *single-threadedness* mühelos verletzten und nicht-deterministisches Verhalten hervorrufen. Es ist grundsätzlich unerwünscht, daß mehrere Threads die Anzeige verändern können, da sie sich in überraschender Weise gegenseitig überschreiben.

[310] Bei SWT ist dies nicht erlaubt. Greift mehr als ein Thread auf die Anzeige zu, wird eine Ausnahme ausgeworfen. Dieser Mechanismus verhindert, daß ein unerfahrener Programmierer versehentlich diesen Fehler macht und schwer auffindbare Fehler in das Programm einschleppt.

[311–312] Das folgende Beispiel zeigt das Swing-Beispiel *ColorBoxes.java* aus Unterabschnitt 23.10.2, umgeschrieben in SWT:

```
//: swt/ColorBoxes.java
// SWT translation of Swing ColorBoxes.java.
import swt.util.*;
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
import org.eclipse.swt.events.*;
import org.eclipse.swt.graphics.*;
import org.eclipse.swt.layout.*;
import java.util.concurrent.*;
import java.util.*;
import net.mindview.util.*;

class CBox extends Canvas implements Runnable {
    class CBoxPaintListener implements PaintListener {
        public void paintControl(PaintEvent e) {
            Color color = new Color(e.display, cColor);
            e.gc.setBackground(color);
            Point size = getSize();
            e.gc.fillRectangle(0, 0, size.x, size.y);
            color.dispose();
        }
    }
    private static Random rand = new Random();
    private static RGB newColor() {
        return new RGB(rand.nextInt(255),
                       rand.nextInt(255), rand.nextInt(255));
    }
    private int pause;
    private RGB cColor = newColor();
    public CBox(Composite parent, int pause) {
        super(parent, SWT.NONE);
        this.pause = pause;
        addPaintListener(new CBoxPaintListener());
    }
    public void run() {
```

```
        try {
            while(!Thread.interrupted()) {
                cColor = newColor();
                getDisplay().asyncExec(new Runnable() {
                        public void run() {
                            try { redraw(); } catch(SWTException e) {}
                            // SWTException is OK when the parent
                            // is terminated from under us.
                        }
                    });
                TimeUnit.MILLISECONDS.sleep(pause);
            }
        } catch(InterruptedException e) {
            // Acceptable way to exit
        } catch(SWTException e) {
            // Acceptable way to exit: our parent
            // was terminated from under us.
        }
    }
}

public class ColorBoxes implements SWTApplication {
    private int grid = 12;
    private int pause = 50;
    public void createContents(Composite parent) {
        GridLayout gridLayout = new GridLayout(grid, true);
        gridLayout.horizontalSpacing = 0;
        gridLayout.verticalSpacing = 0;
        parent.setLayout(gridLayout);
        ExecutorService exec = new DaemonThreadPoolExecutor();
        for(int i = 0; i < (grid * grid); i++) {
            final CBox cb = new CBox(parent, pause);
            cb.setLayoutData(new GridData(GridData.FILL_BOTH));
            exec.execute(cb);
        }
    }
    public static void main(String[] args) {
        ColorBoxes boxes = new ColorBoxes();
        if(args.length > 0)
            boxes.grid = new Integer(args[0]);
        if(args.length > 1)
            boxes.pause = new Integer(args[1]);
        SWTConsole.run(boxes, 500, 400);
    }
} ///:~
```

Wie im vorigen Beispiel painting is controlled durch eine Klasse, die das Interface *PaintListener* implementiert, also eine `paintControl()`-Methode definiert, die aufgerufen wird, wenn der SWT-Thread bereit ist, um Ihre Komponente zu zeichnen. Der Konstruktor der Klasse `CBox` registriert einen solchen Ereignisbehandler.

[313] Die `run()`-Methode der Klasse `CBox` unterscheidet sich deutlich von der Swing-Version, da Sie die `redraw()`-Methode nicht direkt aufrufen können, sondern den Aufruf über die `asyncExec()`-Methode des `Display`-Objektes bewerkstelligen müssen. Diese Vorgehensweise entspricht im Großen und Ganzen der statischen `SwingUtilities`-Methode `invokeLater()`. Wenn Sie diesen Aufruf durch einen direkten Aufruf der `redraw()`-Methode ersetzen, bleibt das Programm einfach stehen.

[314] Wenn Sie das Programm aufrufen, sehen Sie kleine visuelle Artifakte, horizontale Linien, die gelegentlich durch eines der Kästchen laufen. Dieser Effekt entsteht, weil SWT, im Gegensatz zu Swing, nicht doppelt gepuffert ist. Starten Sie beide Versionen nebeneinander und Sie werden den Unterschied deutlicher sehen. Sie können auch bei SWT eine Doppelpufferung erwirken. Sie finden Beispiele hierfür unter der Webadresse *http://www.eclipse.org*.

Übungsaufgabe 42: (4) Ändern Sie das Beispiel *swt/ColorBoxes.java*, so daß es zunächst Punkte („Sterne“) auf dem Darstellungsbereich verstreut und anschließend die Farben dieser „Sterne“ zufällig ändert. ■

23.14.8 Vergleich zwischen SWT und Swing

[315] Eine so kurze Einführung kann schwerlich ein vollständiges Bild vermitteln. Sie sollten aber erkennen können, daß SWT in vielen Situationen eine einfachere Lösung gestattet als Swing. Andererseits kann das Entwickeln einer graphischen Benutzeroberfläche mit SWT durchaus kompliziert sein. Ihre Motivation für SWT sollte an erste Stelle darin bestehen, dem Benutzer ihrer Anwendung einen transparenteren Eindruck zu vermitteln (das Look-and-Feel Ihrer Anwendung entspricht den anderen Anwendungen unter der jeweiligen Plattform.) Die durch SWT gegebene Reaktionsfähigkeit steht an zweiter Stelle. Andernfalls ist Swing unter Umständen eine gangbare Alternative.

Übungsaufgabe 43: (6) Suchen Sie sich eines der Swing-Beispiele aus, das in diesem Abschnitt noch nicht in eine SWT-Version übersetzt wurde und schreiben Sie eine SWT-Version. (Anmerkung: Dies ist eine gute Hausaufgabe für den Unterricht, da die Lösungen *nicht* im *The Thinking in Java Annotated Solution Guide* stehen.) ■

23.15 Zusammenfassung

[316] Die GUI-Bibliotheken haben im Laufe des Lebenszyklus' von Java einige dramatische Änderungen erfahren. Die AWT-Bibliothek von Java 1.0 wurde aufgrund ihres ärmlichen Designs entschieden kritisiert und obwohl Sie mit AWT portable GUIs entwickeln konnten, wirkten die Ergebnisse auf allen Plattformen „gleich mittelmäßig“. AWT war hinsichtlich seiner Handhabung im Vergleich mit den nativen Entwicklungswerkzeugen für die verschiedenen Plattformen außerdem einschränkend, unpraktisch und unangenehm.

[317] Als mit Java 1.1 das neue Ereignismodell und JavaBeans eingeführt wurden, war die Bühne bereitet: Es war möglich geworden, GUI-Komponenten zu entwickeln, die mittels einer visuellen IDE ausgewählt und platziert werden konnten. Das Design des Ereignismodells und die JavaBeans ließen deutlich erkennen, daß erleichtere Programmierung und pflegeleichter Quelltext in Betracht gezogen worden waren (ein Gesichtspunkt, der bei der ersten AWT-Version alles andere als offensichtlich gewesen war). Die Wandlung war aber nicht eher vollzogen, als bis die Java Foundation Classes (JFC) beziehungsweise die Swing-Klassen ins Leben gerufen worden waren. Mit den Swing-Komponenten kann die Entwicklung einer plattformübergreifenden GUI zu einer zivilisierten Erfahrung werden.

[318] IDEs waren der eigentliche Auslöser der Revolution. Wenn Sie sich eine Verbesserung an einer kommerziellen IDE für eine propriertäre Sprache wünschen, müssen Sie die Daumen drücken und hoffen, daß der Hersteller das liefert, was Sie haben wollen. Als offene Umgebung gestattet Java nicht nur den Wettbewerb zwischen IDEs, sondern fördert ihn. Soll ein solches Werkzeug ernst genommen werden, so ist die Unterstützung von JavaBeans unerläßlich. Dadurch ist ein ebenes Spielfeld gewährleistet. Ist eine neue IDE besser als Ihre gegenwärtige IDE, so sind Sie nicht an Ihre IDE gebunden, sondern können auf die neue umsteigen und Ihre Produktivität verbessern. Ein

solcher Wettbewerb um IDEs mit dem Schwerpunkt GUI-Entwicklung war neuartig und der daraus entstandene Markt ist in der Lage sehr positive Ergebnisse im Hinblick auf die Produktivität der Programmierer hervorzubringen.

[319] Dieses Kapitel sollte Ihnen lediglich als Einführung in das Gebiet der GUI-Programmierung dienen und Sie auf den richtigen Weg bringen, indem Sie erkennen, daß es relativ leicht ist, sich in den Bibliotheken zurecht zu finden. Was Sie in diesem Kapitel gelernt haben, deckt voraussichtlich einen guten Teil Ihrer Bedürfnisse in der Entwicklung von Benutzerschnittstellen ab. Es gibt viel mehr über die Eigenschaften und Fähigkeiten von Swing, SWT und Flash/Flex zu sagen; schließlich sind sie als vollwertige Werkzeugkästen zur Entwicklung leistungsfähiger GUIs gedacht. Wahrscheinlich gibt es für alles was Sie sich vorstellen können einen Weg, um dieses Ziel zu erreichen.

23.15.1 Weiterführende Quellen

[320] Ben Galbraith's Online-Präsentation unter der Webadresse *www.galbraiths.org/presentations* liefern eine gute Übersicht über die Themen Swing und SWT.

Hinweis: Sie finden die Lösungen zu ausgewählten Übungsaufgaben im elektronischen *The Thinking in Java Annotated Solution Guide*, den Sie unter der Adresse *http://www.mindview.net* herunterladen können.

Teil V

Anhänge

Anhang A

Ergänzungen und Beilagen

Inhaltsübersicht

[0] Es gibt eine Reihe von Ergänzungen und Beilagen zu diesem Buch, darunter die Informationen, Schulungen und Dienste, die auf der Internetseite von Mindview angeboten werden (*http://www-.mindview.net*).

[1] Dieser Anhang beschreibt die Ergänzungen und Beilagen, damit Sie feststellen können, ob Ihnen etwas davon hilft.

[2] Beachten Sie, daß wir unsere Schulungen nicht nur öffentlich abhalten, sondern auch intern in Ihrem Unternehmen anbieten.

A.1 Quelltextdistribution und ausgelagerte Teile früherer Auflagen

[3] Die Quelltextdistribution zu diesem Buch steht unter der Webadresse *http://www.mindview-.net* zum Herunterladen zur Verfügung und enthält die *build.xml*-Dateien für Ant und alle übrigen Dateien, die zum erfolgreichen Übersetzen und Aufrufen der Beispiele in diesem Buch benötigt werden.

[4] Einige wenige Teile des Buches wurden in elektronischer Form ausgelagert, darunter

- Das Klonen von Objekten
- Über- und Rückgabe von Objekten
- Analyse und Design

- Teile anderer Kapitel aus der dritten Auflage von *Thinking in Java*, die für die vierte Auflage nicht wichtig genug waren.

A.2 Multimediasemiar Thinking in C: Grundlagen für Java

[5] Das Multimediaseminar *Thinking in C* steht zum kostenlosen Herunterladen zur Verfügung. Diese von Chuck Allison entworfene und bei Mindview entwickelte Flash-Präsentation ist eine Einführung in die Syntax, Operatoren und Funktionen von C, auf der die Java-Syntax basiert. Die Präsentation setzt voraus, daß Sie den Flash Player von *http://www.macromedia.com* (automatische Umleitung nach *http://www.adobe.com*) auf Ihrem Rechner installiert haben.

A.3 Die Schulung Thinking in Java

[6] Mindview Inc., meine Firma, bietet fünftägige öffentliche oder unternehmensinterne Schulungen mit praktischen Übungen an, basierend auf diesem Buch. Die frühere Schulung *Hands-On Java* ist unsere Einführungsschulung und vermittelt die Grundlagen für die fortgeschrittenen Schulungen. Eine Schulungslektion besteht aus ausgewählten Inhalten der einzelnen Kapitel und mündet in einen betreuten Übungsabschnitt, so daß jeder Teilnehmer persönliche Aufmerksamkeit erhält. Unter der Webadresse *http://www.mindview.net* finden Sie den Terminplan, die Schulungsorte, Empfehlungsschreiben und sonstige Einzelheiten.

A.4 Die CD zur Schulung Hands-On Java

[7] Die CD *Hands-On Java* beinhaltet eine erweiterte Version des Stoffumfangs der Schulung *Thinking in Java* und basiert auf diesem Buch. Die CD bietet zumindest ein wenig der Erfahrung einer Live-Schulung ohne Reiseaufwand und -kosten. Zu jedem Kapitel des Buches gibt es eine Audiolektion und entsprechende Folien. Ich habe die Schulung entworfen und spreche die Lektionen. Die Inhalte der CD haben Flash-Format und sollten somit auf jeder Plattform laufen, die den Flash-Player unterstützt. Die CD *Hands-On Java* wird auf der Internetseite von Mindview zum Verkauf angeboten, wo Sie auch Produktproben herunterladen können.

A.5 Die Schulung Thinking in Objects

[8] Diese Schulung stellt die Ideen der objektorientierten Programmierung aus der Perspektive des Designers vor. Sie untersucht den Prozeß der Entwicklung und des building eines Systems, und konzentiert sich hauptsächlich auf sogenannte „Agile Softwareentwicklung“ oder „Lightweight Methodologies“, inbesondere Extreme Programming (XP). Die Verfahren werden allgemein eingeführt, etwa die „Karteikarte“ (*index-card*)Planungsmethode, die in Kent Becks und Martin Fowlers *Planning Extreme Programming* (Addison-Wesley, 2001) beschrieben wird, Class-responsibility-collaboration (CRC) cards beim Objektdesign, Paarprogrammierung (*pair programming*), iteration planning, Modultests (*unit testing*), automated building, Versionsverwaltung und ähnliche Themen. Der Kurs beinhaltet ein XP-Projekt, welches während der Woche entwickelt wird.

[9] Wenn Sie gerade am Anfang eines Projektes stehen und objektorientierte Designansätze nutzen möchten, können wir Ihr Projekt als Beispiel verwenden und während der Schulungswoche einen ersten Entwurf erarbeiten.

[10] Unter der Webadresse *http://www.mindview.net* finden Sie den Terminplan, die Schulungsorte, Empfehlungsschreiben und sonstige Einzelheiten.

A.6 Thinking in Enterprise Java

[11] Dieses Buch ist aus einigen fortgeschrittenen Kapiteln der früheren Auflagen von *Thinking in Java* hervorgegangen. *Thinking in Enterprise Java* ist kein zweiter Band von *Thinking in Java*, sondern konzentriert sich auf die Abdeckung des fortgeschrittenen Gebiets der Java Enterprise Edition. Das Buch (in gewisser Weise nach wie vor im Entwicklungsprozeß) steht zur Zeit auf der Internetseite von Mindview Inc. zum kostenlosen Herunterladen zur Verfügung. Als separates Buch kann es ausgedehnt werden, um die benötigten Gebiete zu erfassen. *Thinking in Enterprise Java* hat, wie *Thinking in Java* das Ziel, eine verständliche Einführung in die Grundlagen und Technologien der Java EE zu geben, so daß der Leser auf eine fortgeschrittene Darstellung dieser Gebiete vorbereitet ist.

[12] Die folgenden Gebiete werden behandelt (die Liste kann erweitert werden):

- Einführung in die Java EE
- Netzwerkprogrammierung mit Sockets und ~~Channels~~
- Remote Method Invocation (RMI)
- Verbindung mit einer Datenbank
- Namens- und Verzeichnisdienste
- Servlets
- JavaServer Pages
- Tags, ~~JSP-Fragmente~~ und Expression Language
- Enterprise JavaBeans (EJBs)
- XML
- Webservices
- Automatisches Testen

Sie finden *Thinking in Enterprise Java* in der aktuellen Version auf der Internetseite von Mindview.

A.7 Thinking in Patterns (with Java)

[13] Die „Entwurfsmuster“-Bewegung, aufgezeichnet im Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995), ist einer der wichtigsten Vorwärtsschritte im objektorientierten Design, Das Buch dokumentiert 23 allgemeine Problemklassen und ihre Lösungen, hauptsächlich in C++ geschrieben. Das Buch ist die Quelle des wesentlichen, fast zwingenden Vokabulars in der objektorientierten Programmierung. Das Buch *Thinking in Patterns, Problem-Solving Techniques using Java* stellt die grundlegenden Konzept der Entwurfsmuster zusammen mit Beispielen in Java vor. Es ist nicht einfach als Übersetzung von *Design Patterns* gedacht, sondern als neue Perspektive durch die Java-Denkweise. *Thinking in Patterns, Problem-Solving Techniques using Java* ist nicht auf die traditionellen 23 Entwurfsmuster

beschränkt, sondern beinhaltet an passenden Stellen auch andere Ideen und Problemlösungsverfahren.

[14] *Thinking in Patterns, Problem-Solving Techniques using Java* hat als letztes Kapitel der ersten Auflage von *Thinking in Java* begonnen und wurde zu einem eigenen Buch, nachdem sich die Ideen fortfuhren, sich zu entwickeln. Zum Zeitpunkt der Drucklegung dieses Buches befindet sich *Thinking in Patterns, Problem-Solving Techniques using Java* noch immer im Entwicklungsprozeß, wurde aber inhaltlich aufgrund zahlreicher Präsentationen in der Schulung *Thinking in Objects* mehrfach überarbeitet. (Die Schulung *Thinking in Objects* wurde inzwischen zwei Schulungen *Designing Objects and Systems* und *Thinking in Patterns* aufgeteilt.) Mehr Informationen zu diesem Buch finden Sie auf der Internetseite von Mindview.

A.8 Die Schulung Thinking in Patterns

[15] Diese Schulung hat sich aus der Schulung *Thinking in Objects and Patterns* entwickelt, die Bill Venners und ich in den vergangenen Jahren durchgeführt haben. Die Schulung wurde irgendwann übervoll und wir haben sie in zwei Schulungen aufgeteilt: *Thinking in Patterns* und *Designing Objects and Systems*.

[16] Die Schulung richtet sich deutlich am Inhalt und der Darstellungsweise des Buches *Thinking in Patterns, Problem-Solving Techniques using Java* aus. Die beste Möglichkeit, um sich über das Seminar zu informieren ist, das Buch zu lesen, welches Sie auf der Internetseite von Mindview herunterladen können.

[17] Die Präsentation betont vielerorts den Entwicklungsvorgang beim Design: Es beginnt mit einer ersten Lösung und bewegt sich durch die Logik und den Entwicklungsprozeß der Lösung hin zu besser passenden Designs. Das letzte Projekt (die Simulation einer Abfallverwertungsanlage) hat sich über einen längeren Zeitraum hingezogen und Sie können die Entwicklung als Prototyp für den Weg betrachten, den eigenes Design von einer adäquaten Lösung eines bestimmten Problems hinzu einem flexiblen Ansatz für eine Klasse von Problemen nimmt.

[18] Diese Schulung hilft Ihnen

- Die Flexibilität Ihrer Entwürfe dramatisch zu verbessern.
- Erweiterbarkeit und Wiederverwendung zu implementieren.
- Eine engere Kommunikation über Entwürfe in der Sprache der Entwurfsmuster zu erwirken.

Nach jeder Lektion erhalten Sie einige Übungsaufgaben zu Entwurfsmustern, wobei Sie unter Anleitung bestimmte Entwurfsmuster zur Lösung von Programmierproblemen anwenden sollen. Unter der Webadresse *http://www.mindview.net* finden Sie den Terminplan, die Schulungsorte, Empfehlungsschreiben und sonstige Einzelheiten.

A.9 Beratung, Betreuung und Revision bei Design und Implementierung

[19] Darüber hinaus bietet Mindview auch Beratung, Betreuung und Revision bei Design und Implementierung, um Ihnen durch den Entwicklungszyklus Ihres Projektes zu helfen, insbesondere beim ersten Java-Projekt in Ihrem Unternehmen. Auf der Internetseite von Mindview finden Sie Informationen zur Verfügbarkeit und weitere Einzelheiten.

Anhang B

Quellen

Inhaltsübersicht

B.1 Compiler, Laufzeitumgebung, Standardbibliothek und Dokumentation

[0] Das Java Development Kit (JDK) auf der Java-Seite von Sun Microsystems (*http://java.sun-.com*): Auch wenn Sie sich für die Entwicklungsumgebung eines Drittanbieters entscheiden, ist es sinnvoll, das JDK zur Hand zu haben, falls Sie auf einen eventuellen Fehler des Compilers stoßen. Das JDK ist der Prüfstein. Wenn das JDK einen Fehler hat, stehen die Chancen gut, daß der Fehler bekannt und dokumentiert ist.

[1] Die JDK-Dokumentation im HTML-Format von Sun Microsystems (*http://java.sun.com*): Ich habe noch kein Buch über die Standardbibliothek von Java gefunden, das nicht veraltet oder unvollständig war. Auch wenn die JDK-Dokumentation von Sun Microsystems viele kleine Fehler aufweist und gelegentlich zu knapp ist, um brauchbar zu sein, sind wenigstens alle Klassen und Methoden erfaßt. Der eine oder andere fühlt sich am Anfang mit Online-Dokumentation statt eines gedruckten Buches nicht wohl, aber es lohnt sich, diese Abneigung hinter sich zu lassen und mit der HTML-Dokumentation zu arbeiten, damit Sie mindestens eine Übersicht bekommen. Wenn Sie noch nicht mit der JDK-Dokumentation zurecht kommen, greifen Sie zu einem Buch.

B.2 Editoren und integrierte Entwicklungsumgebungen

[2] Auf diesem Gebiet gibt es einen gesunden Wettbewerb. Viele Angebote sind kostenlos (und zu den kommerziellen Exemplaren wird in der Regel eine kostenlose Probeversion angeboten), so daß Sie am besten einfach ausprobieren, welche Editor beziehungsweise welche Entwicklungsumgebung am besten zu Ihnen paßt. Einige Beispiele:

- *JEdit*: Der kostenlose Editor von Slava Pestov ist in Java geschrieben. Sie bekommen also den Bonus, eine Java-Desktopanwendung in Aktion beobachten zu können. Der Editor basiert auf zahlreichen Plugins, die zu einem großen Teil von der aktiven Gemeinschaft entwickelt werden. Sie können JEdit unter der Webadresse *http://www.jedit.org* herunterladen.
- *NetBeans*: Eine kostenlose Entwicklungsumgebung von Sun Microsystem, entworfen für die Entwicklung graphischer Benutzeroberflächen per „Drag and Drop“, das Editieren von Quelltext, Fehlersuche per Debugger und vieles weitere. Sie können Netbeans unter der Webadresse *http://www.netbeans.org* herunterladen.
- *Eclipse*: Ein von IBM neben anderen Projekten unterstütztes quelloffenes Probjekt. Die Eclipse-Plattform wurde als erweiterbares Fundament entworfen, so daß Sie eigenständige Anwendungen auf der Basis von Eclipse entwickeln können. Das in Abschnitt 23.14 beschriebene Standard Widget Toolkit (SWT) wurde im Rahmen des Eclipse-Projektes entwickelt. Sie können Eclipse unter der Webadresse *http://www.eclipse.org* herunterladen.
- *IntelliJ IDEA*: Unter den kommerziellen Entwicklungsumgebungen der Favorit einer großen Fraktion der Java-Programmierer. Viele Mitglieder dieser Fraktion behaupten, daß IDEA stets einen oder zwei Schritte vor Eclipse liegt, vielleicht weil IntelliJ nicht versucht, eine Entwicklungsumgebung und eine Entwicklungsplattform zu entwickeln, sondern sich auf die Entwicklungsumgebung konzentriert. Sie können eine kostenlose Probeversion von IntelliJ IDEA unter der Webadresse *http://www.jetbrains.com* herunterladen.

B.3 Literaturempfehlungen

- Horstmann, Cay S. and Cornell, Gary: *Core Java 2*, Volumes 1 („Fundamentals“) und 2 („Advanced Features“), jeweils 8th ed., Prentice Hall International (2007 beziehungsweise 2008). Gewaltig, umfassend und mein erster Griff, wenn ich nach Antworten suche. Ich empfehle dieses Buch, wenn Sie *Thinking in Java* abgeschlossen haben und ein weiteres Netz knüpfen möchten.
- Chan, Patrick and Lee, Rosanna: *The Java Class Libraries, An Annotated Reference*, Addison Wesley (1997). Obwohl leider veraltet, liefert dieses Buch, was die JDK-Referenz hätte bieten sollen, nämlich genügend Beschreibung, um das JDK nutzen zu können. Ein technischer Rezensent von *Thinking in Java* schrieb: „Hätte ich nur ein Buch über Java, es wäre dieses (zusätzlich zu Ihrem)“. Ich bin weniger begeistert als dieser Rezensent. Das Buch ist umfangreich, teuer und ich finde die Qualität der Beispiele unbefriedigend. Dennoch ist das Buch eine Stelle zum Nachsschlagen, wenn Sie nicht weiterkommen und scheint mehr Tiefe (und schiere Größe) zu haben, als die meisten Alternativen. *Core Java 2* deckt allerdings die aktuelleren Eigenschaften und Fähigkeiten vieler Bibliothekskomponenten besser ab.
- Harold, Elliotte Rusty: *Java Network Programming*, 2nd ed., O'Reilly (2000).

  Ich habe erst angefangen, die Netzwerkprogrammierung mit Java (und Netzwerke im allgemeinen) zu verstehen, nachdem ich dieses Buch entdeckt hatte. Ich empfinde die Internetseite des Autors (*http://www.cafeaulait.org*) als anregend, eigenwillig und eine fortschrittliche Perspektive auf die Entwicklungen im Java-Umfeld, unbelastet vom Gehorsam gegenüber einem Anbieter. Seine regelmäßigen Aktualisierungen halten mit den sich schnell verändernden Neuigkeiten über Java Schritt.
- Gamma E., Helm R., Johnson R. E. and Vlissides J.: *Design Patterns, Elements of Reusable Object-Oriented Software*, Addison-Wesley (1995). Das wegweisende Buch, welches die

„Entwurfsmuster-Bewegung“ in der Programmierung ausgelöst hat und an vielen Stellen in *Thinking in Java* erwähnt wird.

- Kerievsky, Joshua: *Refactoring to Patterns*, Addison-Wesley (2005). Verbindet Refaktorierung und Entwurfsmuster. Das Wertvollste an diesem Buch ist, daß es Ihnen zeigt, wie Sie ein Design entwickeln, indem Sie bei Bedarf Entwurfsmuster unterziehen.
- Raymond, Eric S.: *The Art of UNIX Programming*, Addison-Wesley (2004). Obwohl Java eine plattformübergreifende Sprache ist, macht die Verbreitung von Java auf Servern Unix/Linux-Kenntnisse erforderlich. Erics Buch ist eine exzellente Einführung in die Geschichte und Philosophie dieses Betriebssystems und faszinierend zu lesen, wenn Sie einfach etwas über die Wurzeln der elektronischen Datenverarbeitung erfahren wollen.

B.3.1 Analyse und Design

- Beck, Kent and Andres, Cynthia: *Extreme Programming Explained*, 2nd ed., Addison-Wesley (2005). Ich hatte schon immer das Gefühl, daß es einen ganz anderen und viel besseren Softwareentwicklungsprozeß gibt und denke, daß Extreme Programming (XP) diesem Gefühl sehr nahe kommt. Das einzige Buch, das mich auf ähnliche Weise beeindruckt hat, war *Peopleware* (siehe unten) und setzt sich hauptsächlich mit der Umgebung und dem Umgang mit Unternehmenskultur auseinander. *Extreme Programming Explained* beschäftigt sich mit Programmierung und stellt die meisten Dinge, selbst neueste „Erkenntnisse“, auf den Kopf. Die Autoren sagen sogar, daß Bilder/Skizzen in Ordnung sind, solange Sie nicht zuviel Zeit damit zubringen und bereit sind, sie zu verwerfen. (Sie werden bemerken, daß das Buch kein „UML-Prüfsiegel“ auf dem Einband hat.) Ich haben Entscheidungen, für ein Unternehmen zu arbeiten, alleine aufgrund dessen fallen sehen, ob das Unternehmen mit XP arbeitet oder nicht. Ein kleines Buch mit kleinen Kapitel, mühelos zu lesen und aufregend, darüber nachzudenken. Sie beginnen, sich selbst unter diesen Bedingungen arbeiten zu sehen. Das Buch vermittelt Visionen einer völlig neuen Welt.
- Fowler, Martin: *UML Distilled*, 2nd ed., Addison-Wesley (2000). Der erste Kontakt mit UML ist aufgrund der vielen Diagramme und Einzelheiten abschreckend. Laut Fowler ist vieles davon unnötigt, so daß er alles bis auf das wesentliche wegschneidet. Bei den meisten Projekten brauchen Sie nur wenige Diagramme als Hilfsmittel und Fowlers Ziel besteht darin, ein gutes Design zu entwickeln, statt sich mit sämtlichen Artefakten auf dem Weg dorthin aufzuhalten. Die erste Hälfte des Buches beinhaltet vielmehr alles was Sie brauchen. Ein schönes, dünnes und lesbares Buch. Das erste Buch, das Sie sich besorgen sollten, wenn Sie UML verstehen müssen.
- Evans, Eric: *Domain-Driven Design*, Addison-Wesley (2004). Das Buch konzentriert sich auf das *Domänenmodell* als primäres Artefakt des Designprozesses. Ich bin zu der Ansicht gekommen, daß das Domänenmodell eine wichtige Verschiebung der Gewichtung ist, die dabei hilft, die Designer auf dem richtige Abstraktionsniveau zu halten.
- Jacobsen, Ivar; Booch, Grady and Rumbaugh, James: *The Unified Software Development Process*, Addison-Wesley (1990). Als ich anfing, war ich komplett darauf vorbereitet, das Buch nicht mögen. Es schien alle Eigenschaften eines langweiligen college textes zu haben. Ich wurde angenehm überrascht, obwohl einige Teile Erklärungen beinhalten, die vermuten lassen, daß den Autoren die Konzepte nicht klar sind. Der Großteil des Buches ist nicht nur klar, sondern unterhaltsam. Und das Beste von allem: Der Unified Software Development Process makes a lot of practical sense. Der Unified Software Development Process ist zwar kein Extreme Programming (und bietet nicht dessen Klarheit im Hinblick auf das Testen), ist

aber ebenfalls Teil der „UML-Bewegung“. Auch wenn es Ihnen nicht gelingt, XP unter Ihren Kollegen zu etablieren, sind die meisten Leute auf den „UML ist gut“-Zug aufgesprungen, so daß sie den Unified Software Development Process wahrscheinlich annehmen werden. Ich finde, dieses Buch sollte das Flaggschiff von UML und das Buch sein, das Sie lesen, wenn Sie nach Fowlers *UML Distilled* Einzelheiten suchen.

Bevor Sie sich für eine Methode entscheiden, hilft es, die Perspektive derer einzunehmen, die nicht versuchen, Ihnen einen Ansatz zu „verkaufen“. Es ist nicht schwer eine Methode zu adaptieren, ohne wirklich zu verstehen, was Sie von ihr erwarten oder was das Verfahren für Sie leisten wird. Daß andere die Methode ebenfalls nutzen, scheint ein zwingender Grund zu sein. Menschen haben allerdings eine seltsame kleine psychologische Eigenart: Glaubt der Mensch, etwas sei im Stande, sein Problem zu lösen, so probiert er es aus. (Dieses Experimentieren ist gut.) Konnte das Problem aber nicht gelöst werden, neigt der Mensch dazu, seine Anstrengungen zu verdoppeln und laut zu verkünden, welch ein großartiges Verfahren er entdeckt hat. (Diese Verweigerungshaltung ist nicht gut.) Eine sinnvolle Erwartungshaltung und Voraussetzung besteht darin, daß Sie nicht alleine sind, wenn es Ihnen gelingt weitere Kollegen ins Boot zu holen, auch wenn die Reise kein Ziel hat (oder das Boot untergeht).

Dies soll nicht heißen, daß jedes Verfahren scheitern muß, sondern daß Sie bis an die Zähne mit mentalen Hilfsmitteln gerüstet sein sollten, die Ihnen helfen, im Experimentiermodus zu bleiben („Es funktioniert nicht. Lassen Sie uns etwas anderes ausprobieren.“) und den Verweigerungsmodus zu vermeiden („Nein. Dies ist nicht wirklich problematisch. Alles ist bestens und wir brauchen nichts zu ändern.“) Die folgenden Bücher werden Sie, meiner Ansicht nach, mit diesen Hilfsmitteln ausstatten, wenn Sie sie *vor* der Entscheidung für ein Verfahren lesen:

- Glass, Robert L.: *Software Creativity*, Prentice Hall (1995).

  Dies ist das beste Buch, das discusses perspective on the whole methodology issue. Das Buch ist eine Sammlung kurzer Essays und Veröffentlichungen, die der Autor selbst geschrieben oder gelegentlich übernommen hat (ein Beitragender ist P. J. Plauger) und reflektiert die vielen Jahre die er über dieses Gebiet nachgedacht und es studiert hat. Die Texte sind unterhaltsam und gerade lang genug, um zu sagen was nötig ist. Glass bummelt und langweilig nicht, verbreitet aber auch keinen Nebel. Das Buch enthält Hunderte von Verweisen auf andere Veröffentlichungen und Studien. Jeder Programmierer und Projektleiter sollte dieses Buch gelesen haben, bevor er durch den Sumpf von Methoden und Verfahren watet.

- Glass, Robert L.: *Software Runaways: Monumental Software Disasters*, Prentice Hall (1998).

  Das Großartige an diesem Buch ist, daß es in den Vordergrund stellt, worüber wir nicht sprechen: Die Anzahl der Projekte, nicht nur mißlingen, sondern auf besonders spektakuläre Weise scheitern. Ich glaube, daß die meisten von uns nach wie vor glauben, „das kann mir nicht passieren“ (oder „das passiert mir nicht noch einmal“) und ich glaube, daß uns diese Denkweise in eine nachteilige Ausgangslage versetzt. Wenn Sie sich dessen bewußt sind, daß etwas schief gehen kann, sind Sie besser positioniert, um die Richtung korrigieren zu können.

- DeMarco, Tom and Lister, Timothy: *Peopleware*, 2nd ed., Dorset House (1999). Sie *müssen* dieses Buch lesen. Es macht nicht nur Spaß, es erschüttert auch Ihre Weltanschauung und zersetzt Ihre Vermutungen. Obwohl DeMarco und Lister ihren Hintergrund in der Softwareentwicklung haben, ist dieses Buch über Projekte und Teams im allgemeinen geschrieben. Der Schwerpunkt liegt aber auf den Menschen und ihren Bedürfnissen, statt auf der Technologie und ihren Anforderungen. Die Autoren sprechen über das Schaffen einer Umgebung, in der sich die Menschen wohlfühlen und produktiv sind, statt über die Entscheidung, welche Regeln die Menschen zu befolgen haben, um zweckmäßige Rädchen einer Maschine zu sein. Die letztere Haltung trägt am meisten dazu bei, daß die Programmierer beim Einführen einer neuen

Methode lächeln, nicken und stillschweigend weiterarbeiten wie zuvor.

- Weinberg, Gerald M.: *Secrets of Consulting: A Guide to Giving and Getting Advice Successfully*, Dorset House (1985). Ein hervorragendes Buch und eines meiner bisher unerreichten Lieblingsbücher. Es ist perfekt, wenn Sie sich selbst als Berater versuchen oder Berater in Anspruch nehmen, um Ihre Arbeitsweise zu verbessern. Kurze Kapitel mit Geschichten und Anekdoten, die Ihnen beibringen, wie Sie zum Kern der Sache vordringen und dabei so wenig wie möglich stolpern.

- Waldrop, M. Mitchell: *Complexity*, Simon & Schuster (1992). Dieses Buch dokumentiert das Zusammentreffen einer Gruppe von Wissenschaftlern verschiedener Disziplinen in Santa Fe (New Mexico), um reale Probleme zu diskutieren, die ihre individuellen Fächer nicht zu lösen vermögen (der Aktienmarkt in den Wirtschaftswissenschaften, die Entstehung des Lebens in der Biologie, die Gründe des menschlichen Verhaltens in der Soziologie und so weiter). Durch Kreuzung von Physik, Wirtschaftswissenschaften, Chemie, Mathematik, Informatik, Soziologie und weiteren Fächern entwickelt sich ein multidisziplinärer Ansatz zur Lösung dieser Probleme. Noch wichtiger ist aber, daß sich ein andere Denkweise hinsichtlich dieser ultrakomplexen Probleme herausbildet: Eine Denkweise, weg vom mathematischen Determinismus und der Illusion, eine Formel angeben zu können, die alles Verhalten vorhersagt und hin zu einem Beobachten und Suchen nach Mustern und dem Versuch, diese Muster mit den vorhandenen Mitteln zu emulieren. (Das Buch dokumentiert beispielsweise das Aufkommen der genetischen Algorithmen.) Ich halte diese Art zu Denken für wertvoll, da wir Wege beobachten können, um die zunehmend komplexen Softwareprojekte bewerkstelligen zu können.

B.3.2 Python

Lutz, Mark and Ascher, David: *Learning Python*, 2nd ed., O'Reilly (2003). Eine schöne Einführung für Programmierer in meine Lieblingssprache und einen exzellenten Gefährten für Java. Das Buch beinhaltet eine Einführung in Jython, eine reine Java-Implementierung von Python, die Ihnen gestattet, Java und Python in einem einzigen Programm zu kombinieren (der Jython-Interpreter ist in reinen Java-Bytecode übersetzt, so daß Sie nichts spezielles ergänzen müssen, ~~to accomplish this~~). Diese Sprachvereinigung verspricht großartige Möglichkeiten.

B.3.3 Meine eigenen Bücher

[5] Nicht alle Bücher sind zur Zeit lieferbar, können aber zum Teil über Antiquariate bezogen werden.

- *Computer Interfacing with Pascal & C*, 1988 im Selbstverlag unter dem Verlagszeichen von Eissys herausgegeben. Nur über die Internetseite von Mindview zum Kauf erhältlich. Eine Einführung in die Elektronik von der Zeit an, als CP/M noch regierte und DOS ein Emporkömmling war. Ich verwendete Hochsprachen und häufig den parallelen Port des Computers, um verschiedene elektronische Projekte zu steuern. Das Buch ging aus Kolumnen für die erste und beste Zeitschrift hervor, für ich jemals geschrieben habe: Micro Cornucopia. Bedauerlicherweise war die Zeitschrift verloren, bevor es das Internet gab. Die Arbeit an diesem Buch war eine außerordentlich befriedigende Veröffentlichungserfahrung.

- *Using C++*, Osborne/McGraw-Hill (1989). Eines der ersten Bücher über C++. Das Buch ist vergriffen und wurde durch seine zweite Auflage ersetzt, die unter dem Namen *C++ Inside & Out* erschienen ist.

- *C++ Inside & Out*, Osborne/McGraw-Hill (1993). Dieses Buch ist eigentlich die zweite Auflage von *Using C++* (siehe dort). Das C++ in diesem Buch ist einigermaßen genau, stammt aber von 1992 und *Thinking in C++* sollte *C++ Inside & Out* ersetzen. Auf der Internetseite von Mindview Inc. finden Sie weitere Informationen über das Buch und können den Quelltext herunterladen.
- *Thinking in C++*, erste Auflage, Prentice Hall (1995). Dieses Buch hat den Jolt-Award des *Software Development Magazine* für das beste Buch des Jahres gewonnen.
- *Thinking in C++*, Band 1, zweite Auflage, Prentice Hall (2000). Sie finden dieses Buch auf der Internetseite von Mindview Inc. zum Herunterladen. Das Buch wurde aktualisiert, um dem abgeschlossenen Sprachstandard zu folgen.
- *Thinking in C++*, Band 2, mit Chuck Allison, zweite Auflage, Prentice Hall (2003). Sie finden dieses Buch auf der Internetseite von Mindview Inc. zum Herunterladen.
- *Black Belt C++: The Masters Collection*, M&T Books (1994), herausgegeben von Bruce Eckel. Das Buch ist vergriffen. Eine Sammlung von Kapiteln verschiedener Koryphäen auf dem Gebiet C++, basierend auf dem C++-Zug der Software Development Conference, deren Vorsitz ich hatte. Der Einband dieses Buches hat mich angeregt, die Kontrolle über alle zukünftigen Einbandentwürfe zu übernehmen.
- *Thinking in Java*, erste Auflage, Prentice Hall (1998). Die erste Auflage dieses Buches hat den Productivity-Award des *Software Development Magazine*, den Editors-Choice-Award des *Java Developers Journal* und den Readers-Choice-Award der *JavaWorld* für das beste Buch gewonnen. Sie finden das Buch auf der Internetseite von Mindview Inc. zum Herunterladen.
- *Thinking in Java*, zweite Auflage, Prentice Hall (2000). Diese Auflage hat den den Editors-Choice-Award der *JavaWorld* für das beste Buch gewonnen. Sie finden das Buch auf der Internetseite von Mindview Inc. zum Herunterladen.
- *Thinking in Java*, dritte Auflage, Prentice Hall (2003). Diese Auflage hat den Jolt-Award des *Software Development Magazine* für das beste Buch des Jahres gewonnen, zusammen mit anderen Preisen, die auf dem Buchrücken verzeichnet sind. Sie finden das Buch auf der Internetseite von Mindview Inc. zum Herunterladen.

Stichwortverzeichnis

B

C

D

G

H

J

K

M

P

Q

R

S

U