Performant Java programmieren

Die Reihe Programmer's Choice

Von Profis für Profis
Folgende Titel sind bereits erschienen:

Bjarne Stroustrup
Die C++-Programmiersprache
1072 Seiten, ISBN 3-8273-1660-X

Elmar Warken
Kylix – Delphi für Linux
1018 Seiten, ISBN 3-8273-1686-3

Don Box, Aaron Skonnard, John Lam
Essential XML
320 Seiten, ISBN 3-8273-1769-X

Elmar Warken
Delphi 6
1334 Seiten, ISBN 3-8273-1773-8

Bruno Schienmann
Kontinuierliches Anforderungsmanagement
392 Seiten, ISBN 3-8273-1787-8

Damian Conway
Objektorientiertes Programmieren mit Perl
632 Seiten, ISBN 3-8273-1812-2

Ken Arnold, James Gosling, David Holmes
Die Programmiersprache Java
628 Seiten, ISBN 3-8273-1821-1

Kent Beck, Martin Fowler
Extreme Programming planen
152 Seiten, ISBN 3-8273-1832-7

Jens Hartwig
PostgreSQL – professionell und praxisnah
456 Seiten, ISBN 3-8273-1860-2

Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides
Entwurfsmuster
480 Seiten, ISBN 3-8273-1862-9

Heinz-Gerd Raymans
MySQL im Einsatz
618 Seiten, ISBN 3-8273-1887-4

Dusan Petkovic, Markus Brüderl
Java in Datenbanksystemen
424 Seiten, ISBN 3-8273-1889-0

Joshua Bloch
Effektiv Java programmieren
250 Seiten, ISBN 3-8273-1933-1

Hendrik Schreiber

Performant Java programmieren

An imprint of Pearson Education

München • Boston • San Francisco • Harlow, England
Don Mills, Ontario • Sydney • Mexico City
Madrid • Amsterdam

Die Deutsche Bibliothek – CIP-Einheitsaufnahme

Ein Titeldatensatz für diese Publikation ist bei
Der Deutschen Bibliothek erhältlich.

Umwelthinweis:
Dieses Produkt wurde auf chlorfrei gebleichtem Papier gedruckt.
Die Einschrumpffolie – zum Schutz vor Verschmutzung – ist aus umweltverträglichem und recyclingfähigem PE-Material.

5 4 3 2 1

05 04 03 02

ISBN 3-8273-2003-8

ein Imprint der Pearson Education Deutschland GmbH,
Martin-Kollar-Straße 10–12, D-81829 München/Germany

Einbandgestaltung: Christine Rechl, München
Titelbild: Cirsium oleraceum, Kohldistel. © Karl Blossfeldt Archiv
Ann und Jürgen Wilde, Zülpich/ VG Bild-Kunst Bonn, 2002
Lektorat: Christiane Auf, cauf@pearson.de; Tobias Draxler, tdraxler@pearson.de
Herstellung: Monika Weiher, mweiher@pearson.de
CD-Mastering: Gregor Kopietz, gkopietz@pearson.de
Satz: reemers publishing services gmbh, Krefeld, www.reemers.de
Druck und Verarbeitung: Bercker Graphischer Betrieb, Kevelaer
Printed in Germany

Erfahrung Karpfen, Neugier Hecht.
(Manfred Hinrich)

Inhalt

Vorwort

Seit ich 1996 mit Java in Berührung kam, war Performance in der ein oder anderen Form immer ein wichtiges Thema. Zunächst war es essentiell, die Größe von Applets zu verringern, dann eine in Java verfasste Skriptsprache zu optimieren und schließlich musste kleinen sowie großen Anwendungen der CPU-Hunger abgewöhnt werden.

Währenddessen wurden die Virtuellen Maschinen immer schneller, Just-in-Time-Compiler lösten Interpreter ab und Ahead-of-Time-Compiler sowie HotSpot betraten das Spielfeld. Mit den unterschiedlichen VMs änderten sich auch die Tricks und Kniffe, die mit der letzten VM noch zu großen Performance-Gewinnen geführt hatten. Schließlich entwickelte Sun mit großer Unterstützung der Industrie Enterprise-APIs, die zu J2EE (Java 2 Enterprise Edition) gebündelt wurden. Die daraus resultierenden Produkte, die Applikationsserver, sind der bisher größte Erfolg, den Java erzielt hat. Mittlerweile ist Java jenseits des Hypes. Java ist etabliert.

Doch natürlich hat Java auch Kritiker, hat sie immer gehabt. Und gerade die sehen in der Performance Javas größten Schwachpunkt. Dass diese Kritik nicht ganz unberechtigt ist, zeigen die Probleme, die auch Java-Jünger gelegentlich haben. Oftmals ist es dabei Unwissenheit, die zu Schwierigkeiten führt, und nicht Java an sich.

Doch trotz Javas Erfolg und akuter Performance-Probleme gab es bis dato kaum deutsche Bücher über Java-Performance, darüber, wie man Programme schreibt, so dass sie performant werden. Dies hat mich dazu motiviert, dieses Buch zu schreiben.

Zum Buch

Das Buch ist grob in zwei Teile geteilt. In den ersten vier Kapiteln gehe ich auf Grundlagen ein, die meiner Ansicht nach als Hintergrundwissen unentbehrlich sind. Hier werden Fragen beantwortet wie: Was ist Performance? Welche Bordmittel brauche ich, um effizient programmieren zu können? Wie sieht ein Entwicklungsprozess aus, der zu leistungsfähiger Software führen kann? Wie funktionieren Java VMs und welche Werkzeuge stehen zum Testen, Messen und Optimieren zur Verfügung?

Im zweiten Teil fokussiert jedes Kapitel einen Themenbereich aus dem Programmiereralltag. Dies sind allgemeine Entwicklungstechniken, aber auch Spezialthemen wie beispielsweise Thread-Programmierung oder XML. Dabei wird anhand von Beispielen plastisch vorgeführt, was zu einer Performance-Verbesserung führen kann und was eher nicht. Dabei lege ich Wert darauf, guten Stil nicht auf dem Geschwindigkeitsaltar zu opfern. Einige der Beispiele sind zum besseren Verständnis zudem mit UML-Diagrammen illustriert.

Es wäre vermessen zu behaupten, dass die Ideen zu allen Tipps und Hinweisen zwischen meinen eigenen Ohren entstanden. Tatsächlich fußen viele der beschriebenen Optimierungstechniken auf Ideen anderer Entwickler, Autoren und Kollegen. Es handelt sich also um *Best Practices*, wie es so schön auf Neudeutsch heißt.

Viele der Beispiele sind Listings, die Sie auch in elektronischer Form auf der beiliegenden CD-ROM bzw. im WWW unter der Adresse *http://www.tagtraum.com/performance/* finden und so besser nachvollziehen können. Soweit nicht anders angegeben, habe ich übrigens für alle Messungen ein Dell Inspiron 7500 Notebook mit Intel Pentium III 500 Mhz und Microsoft Windows 2000 Professional benutzt.

Selbstverständlich habe ich mir die allergrößte Mühe gegeben, Fehler zu vermeiden. Doch bekanntlich steckt der Teufel im Detail und genau wie jedes größere Programm hat jedes Buch Fehler. Zudem werden sicher einige der angegebenen URLs mit der Zeit ungültig. Wenn Sie einen Fehler finden, schreiben Sie mir (*hs@tagtraum.com*) oder dem Verlag, so dass ich ihn auf der Errata-Seite der Website richtig stellen kann.

Danksagungen

Zuallererst möchte ich mich bei Ihnen bedanken. Es gibt so viele gute Bücher, daher betrachte ich es als Kompliment, dass Sie ausgerechnet mein Buch erworben haben. Danke! Ich hoffe dieses Buch erweist sich für Sie als nützlich und endet nicht als Staubfänger im Regal der ungelesenen Bücher.

Ganz abgesehen davon ist es eine Ehre, sich über rund 300 Seiten verbreiten zu dürfen und dafür auch noch bezahlt zu werden. Diese Ehre wurde mir zuteil durch das Vertrauen von Addison-Wesley und die Tatsache, dass mich die innoQ Deutschland GmbH großzügigerweise für vier Monate freistellte. Genau aus diesem Grund muss ich mich an dieser Stelle nicht für unzählige durchgearbeitete Nächte und Wochenenden bei meinen Nächsten entschuldigen, wie das in anderen Büchern so häufig der Fall ist. Stattdessen möchte ich mich für die Unterstützung bei der innoQ im Allgemeinen und Stefan Tilkov im Besonderen herzlich bedanken.

Dank auch an Christiane Auf, Tobias Draxler und Philipp Burkart von Addison-Wesley für die unkomplizierte Zusammenarbeit und an Michael Neumann von Line Information GmbH für detailliertes Feedback und einen charmanten Hang zum Perfektionismus. Und schließlich gilt mein Dank all jenen, die mich während des Schreibens ermutigten und mit Rat, Zuneigung und anderen wichtigen Dingen unterstützten. Danke Jennifer Fuller, Barbara, Rolf und Marc Schreiber, Phillip Ghadir, Enke Eisenberg und Jason Sullivan.

Hendrik Schreiber

Raleigh, North Carolina, Mai 2002

1 Java ist zu langsam

Seit es Java gibt, gibt es Kritiker, die behaupten, Java sei zu langsam für dieses und jenes. Die automatische Speicherbereinigung (Garbage Collection) fresse die gesamte Rechenzeit auf und überhaupt, interpretierte Sprachen seien die Wurzel allen Übels.

Zu leugnen, dass Java in den ersten Versionen ein Performance-Problem hatte, hieße zu behaupten, dass Sanduhr-Mauszeiger ein Symbol für Produktivität sind und ein Repaint auch an schnellen Rechnern noch mit bloßem Auge nachvollziehbar sein muss. Denn so viel steht fest: Java überzeugte bestimmt nicht durch Geschwindigkeit. Es waren andere Eigenschaften wie Sicherheit, Netzwerkfähigkeit und Portabilität, die die Massen verführten.

Nun sind seit der ersten Java-Version einige Jahre vergangen und die Hersteller von Java Ausführungsumgebungen, den Java Virtuellen Maschinen (Java VM), hatten Zeit diese zu optimieren und die ein oder andere Finesse einzubauen. In der Zwischenzeit sind zudem die Rechner sehr viel schneller geworden (Abbildung 1.1).

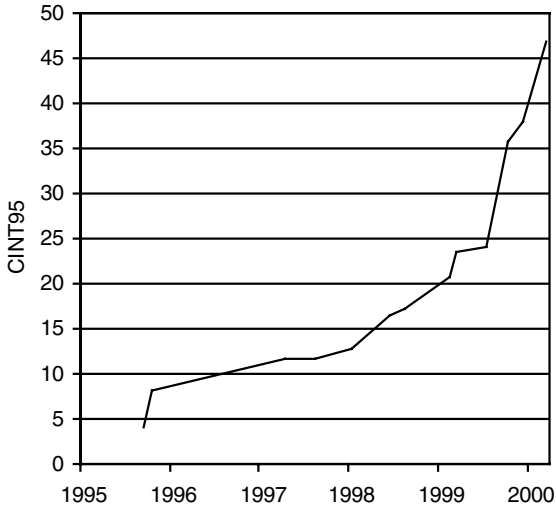

Abbildung 1.1: Von Mitte 1995 bis Anfang 2000 hat sich die Leistung bei Integer-Operationen von Intel-Desktop-Prozessoren mehr als verzehnfacht. Quelle: http://www.spec.org.

Und trotzdem hat Java immer noch den Ruf langsam zu sein. Gerade Programmierer, die aus anderen Sprachen zu Java wechseln (müssen!), stimmen gerne in den Chor der Nörgler ein. Java sei nicht performant, und *das würde mit C/C++/Delphi/Perl/Assembler/etc. viel schneller laufen*. Zugegeben, dies mag in einigen Fällen stimmen. Aber abgesehen davon, dass solch unreflektierte Pauschal-Kritik ganz erheblich nerven kann, ist sie unproduktiv und bringt den Kritisierenden in Verruf. Denn wenn der Bauer nicht schwimmen kann, liegt's bekanntlich an der Badehose.

Studien belegen, dass Performance-Unterschiede von Programmen, die in der gleichen Sprache, aber von verschiedenen Entwicklern verfasst wurden, mindestens so groß sind wie die Performance-Unterschiede von Programmen, die in unterschiedlichen Sprachen geschrieben wurden [Prechelt00, S.29].

1.1 Was ist Performance?

Bevor wir uns damit beschäftigen, wie wir die Performance unserer Programme verbessern, wollen wir zunächst einmal klären, was unter Performance zu verstehen ist. Leider ist Performance einer jener Anglizismen, die ihre volle Bedeutung nicht auf den ersten Blick entfalten.

Performance ist nicht nur reine Rechengeschwindigkeit. Zur Performance eines Programms gehören außerdem die Geschwindigkeit von Ein-/Ausgabe-Operationen sowie der Speicherverbrauch. So kann, wenn nur wenig schneller Speicher zur Verfügung steht, ein schnelles Programm mit hohem Speicherbedarf offensichtlich weniger performant sein als ein langsames Programm mit geringem Speicherbedarf. An diesem Beispiel wird schon klar, dass das Inanspruchnehmen von Rechenleistung, Speicher und Ein-/Ausgabe-Operationen oft in einem gespannten Verhältnis zueinander stehen. Verbraucht ein Programm wenig Speicher, so benötigt es oft eine hohe Rechenleistung. Ist die benötigte Rechenleistung gering, so ist häufig eine hohe Ein-/Ausgabe-Geschwindigkeit vonnöten. Wenn jedoch die Ein-/Ausgabe-Geschwindigkeit irrelevant ist, erweist sich unter Umständen der Speicherverbrauch als sehr hoch.

Überspitzt gesehen drängt sich der Eindruck auf, dass das Produkt der benötigten Ein-/Ausgabe- (`EA`) und Rechenleistung (`R`) sowie des Speicherverbrauchs (`M`) von `n` verschiedenen Problemlösungen konstant ist:

$$EA_n * R_n * M_n = konst$$

Nun ist dies leider keine erwiesene Tatsache, sondern allenfalls eine interessante Hypothese. Jedoch eine, die uns weiterbringt. Einmal angenommen, die Formel wäre eine anerkannte Tatsache und gültig für alle Programme dieser Welt. Dann müssten

wir nur noch die genauen technischen Daten der Zielplattform beim Hersteller erfragen und könnten für eben diese Zielplattform die optimale Version des Programms schreiben. Ein fähiges Team vorausgesetzt, wäre der schwierigste Teil der Aufgabe vermutlich, die korrekten Daten vom Hersteller zu bekommen.

Nun ist die obige Formel aber leider keine Tatsache. Rechenleistung und Ein-/Ausgabe-Geschwindigkeit gleichberechtigt in einer Formel zu verewigen erscheint höchst riskant, und auch nur einen der drei Werte verlässlich zu messen, ist ein mehr als heikles Unterfangen. Wir erkennen jedoch, dass ein Programm an sich nicht performant ist. Es kann lediglich in einer bestimmten Umgebung performant sein. Daraus folgt:

Performant sind solche Programme, die die vorhandenen Ressourcen effizient nutzen.

Nehmen wir zum Beispiel einen Webserver. Vereinfacht betrachtet ist es seine Aufgabe, Dateien von der Festplatte zu einem Netzwerkadapter zu kopieren. Auf den ersten Blick sind also ein möglichst schneller Netzwerkadapter, eine möglichst schnelle Festplatte und eine möglichst schnelle Verbindung zwischen beiden Geräten nötig. Mit anderen Worten: Ein schneller Webserver zeichnet sich durch besonders effiziente Ein-/Ausgabe-Operationen aus.

Nun ist eine Festplatte verglichen mit dem Hauptspeicher in der Regel nicht besonders schnell. Es macht also Sinn, die Dateien im Hauptspeicher zu halten statt jedes Mal von der langsamen Festplatte zu lesen (siehe auch *Kapitel 10.4 Dateicache*). Leider ist jedoch der Hauptspeicher meist nicht groß genug für alle Dateien. Es ist also erstrebenswert, nur solche Dateien im Speicher zu halten, die besonders häufig nachgefragt werden.

Anstatt nur die offensichtlich notwendigen Ressourcen zu nutzen, bedient sich der Webserver aller ihm zur Verfügung stehenden Ressourcen, die der Leistungssteigerung dienen: des Netzwerkadapters, der Festplatte *und* des Hauptspeichers. Und zwar mit Betonung auf *ihm zur Verfügung stehenden*. Keinesfalls mehr!

Versucht der Webserver mehr Dateien im Hauptspeicher zu halten als realer und somit schneller Speicher vorhanden ist, macht das keinen Sinn. Im Gegenteil, Dateien würden in den Hauptspeicher geladen, um anschließend vom Betriebssystem wieder auf die Festplatte ausgelagert zu werden. Dateien wären also unnötigerweise doppelt auf der Festplatte vorhanden und belegten wertvollen Speicherplatz. Hinzu käme der erhebliche Aufwand, den das Betriebssystem betreiben muss, um Speicherseiten ein- und auszulagern. Hierfür gibt es einen Namen: Ressourcenverschwendung.

Nun liegt es in der Natur der Sache, dass optimale Ressourcennutzung und Verschwendung nahe beieinander liegen. Geschickt haushalten ist daher die halbe Miete.

1.2 Empfundene Performance

Es gibt jedoch Programme, die nach allen Regeln der Kunst optimiert wurden, gemäß obiger Definition performant sind und dennoch von Benutzern als langsam bezeichnet werden. Und leider lassen sich Nutzer nur selten durch technische Argumente von der Performance eines Programms überzeugen.

Empfundene Performance ist letztlich, was zählt.

Die gilt insbesondere, wenn der Nutzer direkt mit dem Programm interagiert. So macht es einen großen Unterschied, ob ein Programm Benutzeraktionen einfach ignoriert oder ob es mittels eines Sanduhrzeigers signalisiert, dass es gerade beschäftigt ist. Ein Positiv-Beispiel sind die gängigen Webbrowser. Während sie Daten über das Netz laden, zeigen sie dem Nutzer durch eine Animation an, dass sie beschäftigt sind. Üblicherweise befindet sich in der Statusleiste zudem eine Fortschrittsanzeige, die dem Benutzer ein Gefühl dafür vermittelt, wie lange er noch zu warten hat. Meist werden sogar Teilergebnisse unmittelbar dargestellt. Darüber hinaus versetzt ein Abbruch-Knopf den Nutzer in eine psychologisch wichtige Machtposition. Er kann selbst entscheiden, ob er noch länger warten möchte oder nicht, d.h. er muss sich nicht dem Programm unterordnen. Und Nutzer hassen nichts mehr, als sich einem Programm unterzuordnen.

Grundsätzlich gilt: Wenn Wartepausen unvermeidbar sind, muss der Nutzer möglichst über den Fortschritt des Prozesses informiert werden. Ist dies nicht möglich, sollte ihm beispielsweise durch eine Animation signalisiert werden, dass der Prozess noch im Gange ist. Kommt auch das nicht in Frage, so muss dem Nutzer zumindest vermittelt werden, dass das Programm gerade beschäftigt ist.

Ende 2001 lief im deutschen Fernsehen ein Werbespot der Deutschen Bundesbahn, in dem sie auf ihre Pünktlichkeit im Vergleich zu anderen Verkehrsmitteln hinwies. Im Spot wurden wartende Passagiere per Anzeigetafel und Durchsage über die dreiminütige Verspätung eines Zuges unterrichtet, was zu einer Großdemonstration führte. Einmal abgesehen davon, dass viele sich über jede nur dreiminütige Verspätung eines Bundesbahnzuges freuen würden, illustriert der Spot, wie wichtig es ist, den Nutzer auf dem Laufenden zu halten. Stellen Sie sich nur einmal vor, die Bahn hätte ihre Kunden nach zweiminütigem Warten noch immer nicht über die Verspätung informiert. Was wären dann wohl die Folgen gewesen ...

1.3 Gesunder Menschenverstand

Wir wissen nun, worauf es ankommt: Effizient mit den vorhandenen Ressourcen haushalten und Rücksicht auf die nicht-funktionalen Bedürfnisse der Benutzer nehmen.

Bloß – wie erreichen wir das?

Ich behaupte, die lösbaren Performance-Probleme bekommen Sie mit Wissen, Neugierde und ein wenig gesundem Menschenverstand in den Griff. Für die unlösbaren müssen Sie vermutlich etwas Zeit investieren, denn dank Moores Gesetz[1] hat Zeit bislang noch die meisten Performance-Probleme gelöst. Nun ist Zeit jedoch knapp, weshalb wir uns lieber mit den lösbaren Problemen auseinander setzen.

Rufen Sie sich ins Gedächtnis zurück, dass Java eine Hochsprache ist. Java bietet exzellente Abstraktionen für Betriebssystemspezialitäten und die darunter liegende Hardware. So müssen Sie sich beispielsweise nie mit den Eigenheiten einer Prozessorarchitektur herumschlagen. Die Besonderheiten des ausführenden Systems werden so weit es geht wegabstrahiert. An seine Stelle treten die VM und die Klassenbibliotheken der Java-Plattform.

Nun ist eine Hochsprache wie Java weit mehr als die Abstraktion von Betriebssystem und Hardware. Java ist objektorientiert, besitzt eine automatische Speicherbereinigung, bietet ein Sicherheitskonzept, verfügt über ausgereifte Netzwerkunterstützung etc. All dies hat einen Preis. Und manchmal besteht dieser Preis darin, dass etwas einfach aussieht, es in Wirklichkeit aber nicht ist. Oder anders gesagt, dass der Schein trügt. Ihre Neugierde und Ihr Verstand helfen Ihnen die Wahrheit herauszufinden.

Betrachten wir ein Beispiel:

```
public void allocate() {
   {
      byte[] a = new byte[1000000];
   }
   {
      byte[] a = new byte[1000000];
   }
   {
      byte[] a = new byte[1000000];
   }
   ...
}
```

Listing 1.1: Hoffen auf die Müllabfuhr

Man würde erwarten, dass die Methode `allocate()` problemlos ausführbar ist – wird doch der Array `a` jeweils in einem eigenen Block deklariert und dann dieser Block sofort verlassen. Nach dem Verlassen ist `a` somit nicht mehr sichtbar; jetzt sollte die Speicherbereinigung zum Zuge kommen. Doch weit gefehlt. Mit den meisten VMs resultiert das Ausführen der Methode nach einigen `byte`-Array-Allokationen in einem `OutOfMemoryError`. Der Grund dafür ist einfach: die `byte`-Arrays sind außerhalb ihres

1 Gemäß Moores Gesetz verdoppelt sich die Transistordichte integrierter Schaltkreise alle 18 Monate. Entsprechend erhöht sich auch die Leistungsfähigkeit von Mikroprozessoren. Das Gesetz wurde Mitte der 60er Jahre vom späteren Intel-Gründer Gordon Moore aufgestellt und bezog sich zunächst auf einen Zeitraum von je 12 Monaten. Seit 1970 hat es sich alle 18 Monate bewahrheitet.

Blocks zwar nicht mehr sichtbar, d.h. man kann auf `a` nicht mehr zugreifen, aus Gründen der Effizienz werden die auf dem Stack allozierten Referenzen jedoch erst beim Verlassen der Methode vom Garbage Collector eingesammelt. Denkt man kurz darüber nach, leuchtet dieses Verhalten sofort ein. Würde der Garbage Collector nach jedem Block oder gar Statement aufgerufen, wäre die VM vollauf mit sich selbst beschäftigt. Der auszuführende Code verkäme zur Nebensache.

1.4 Wissen ist Performance

Zugegeben, obiges Beispiel hat viel mit Wissen über die Interna von VMs und Garbage Collectoren zu tun. Aber gerade das macht es so geeignet. Wissen ist eines Ihrer wichtigsten Werkzeuge zum Optimieren von Programmen.

Nun kann man Wissen erwerben, nur leider nicht im Supermarkt um die Ecke. Man muss es sich aneignen. Beispielsweise indem man ein Buch liest, mal eine Stunde in der Java-Sprachspezifikation stöbert oder sich mit Kollegen austauscht. Wichtige Ressourcen müssen zudem unmittelbar für alle Entwickler verfügbar sein.

Entsprechendes gilt für viele Berufe, in denen Menschen hauptsächlich fürs Denken bezahlt werden. So gibt es im Journalismus die so genannten *Bordmittel*, die jeder gute Journalist an seinem Arbeitsplatz haben sollte. Für einen Politikredakteur gehören dazu beispielsweise der so genannte Oeckl, ein Buch mit Telefonnummern und Adressen aller wichtigen Personen und Organisationen der Bundesrepublik Deutschland, sowie das Munzinger Archiv, eine Sammlung von Lebensläufen aller wichtigen Personen des öffentlichen Lebens. Ohne diese beiden Nachschlagewerke ist es quasi unmöglich, guten politischen Journalismus zu betreiben. Ein Telefon und ein Computer mit einem Redaktionssystem allein reichen einfach nicht.

Das Gleiche gilt für Java-Entwickler.

Man kann keine guten Java-Programme schreiben, wenn man nicht direkten, lokalen Zugriff auf die Java-API-Dokumentation und den Java-Quellcode hat.

Das bedeutet, dass Sie nach dem Herunterladen des Java Development Kits (JDK) auch noch die zugehörige Dokumentation herunterladen müssen. Entpacken Sie die Dokumentation und setzen Sie in Ihrem Browser ein Lesezeichen auf den API-Teil. Anschließend – und dies ist der Schritt, den die meisten leider vergessen – entpacken Sie die Datei *src.jar* bzw. im Fall von JDK 1.4 die Datei *src.zip*, die sich üblicherweise im Basisverzeichnis des JDKs befindet.

```
C:\>cd jdk1.3.1
C:\jdk1.3.1>bin\jar xf src.jar
```

Listing 1.2: Entpacken des Sun JDK 1.3.1 Java-Quellcodes auf einem Windows-System

```
C:\>cd j2sdk1.4.0
C:\j2sdk1.4.0>mkdir src
C:\j2sdk1.4.0>cd src
C:\j2sdk1.4.0\src>..\bin\jar -xf ..\src.zip
```

Listing 1.3: Entpacken des Sun JDK 1.4.0 Java-Quellcodes auf einem Windows-System

Der Java-Quellcode befindet sich nun im Verzeichnis *C:\jdk1.3.1\src* bzw. *C:\j2sdk1.4.0\src*. Konfigurieren Sie Ihre Entwicklungsumgebung so, dass Sie problemlos auf den Code zugreifen können. Denn obgleich die API-Dokumentation von Java vorbildlich ist, gilt die alte Programmiererweisheit:

Die Wahrheit steht im Code.

Und kein ernst zu nehmender Entwickler kann es sich leisten, die Wahrheit zu ignorieren.

Leider liegen oft nicht alle verwendeten Bibliotheken im Quellcode vor. Sofern es die Lizenzbedingungen zulassen, erweisen hier Decompiler wie *jad* (*http://kpdus.tripod.com/jad.html*) wertvolle Dienste.

Einige Entwicklungsumgebungen[2] erlauben es dem Entwickler, per Mausklick zum Quellcode einer Methode oder einer Klasse zu navigieren. Kaufen Sie Ihren Entwicklern eine solche IDE oder überzeugen Sie Ihren Projektleiter davon, Ihnen und Ihren Kollegen eine solche IDE zu kaufen. Ihre Produktivität wird sich steigern und dank Ihrer Neugierde werden Sie sich langfristig mehr Wissen aneignen. Wissen darüber, was tatsächlich passiert, wenn Sie diese oder jene unscheinbare Methode aufrufen.

Dieses Wissen über die verwendeten Klassenbibliotheken wird Ihnen letztendlich helfen, performanten Code zu schreiben.

Natürlich müssen Sie nicht den Quellcode *jeder* Klasse vor Gebrauch studieren. Es reicht vollkommen, sich bei Bedarf die kritischen Stellen anzuschauen. Darüber hinaus gibt es jedoch Dinge, die Sie auf jeden Fall wissen sollten. Diese will dieses Buch vermitteln.

2 Z.B. IntelliJIDEA, *http://www.intellij.com/*.

2 Entwicklungsprozess

Wenn Sie kurz vor der Übergabe Ihres zweijährigen, performancekritischen Projektes zum ersten Mal die Leistungsfähigkeit Ihrer Software messen und feststellen, dass Sie Ihr Ziel um einige Größenordnungen verfehlt haben, werden Sie schnell merken, dass Sie etwas falsch gemacht haben. Und zwar von Anfang an. Denn jetzt, kurz vor der Übergabe, sind die Kosten, die Performance noch zu erhöhen, gewöhnlich exorbitant. Im Allgemeinen wird davon ausgegangen, dass, wenn Sie Software nach dem oben beschriebenen Modell entwickeln, die Kosten für Änderungen mit dem Projektfortschritt exponentiell steigen (Abbildung 2.1).

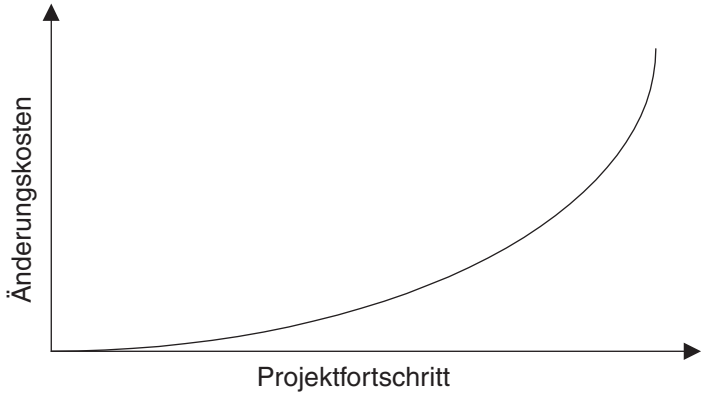

Abbildung 2.1: In traditionellen Wasserfall-Projekten steigen die Kosten von Änderungen exponentiell mit dem Projektfortschritt [vgl. Beck00, S. 21].

Es ist wichtig festzustellen, dass nicht die Entwickler einen lausigen Job abgeliefert haben, sondern diejenigen, die für den Entwicklungsprozess verantwortlich waren.

Performante Software lässt sich nur mit einem auf dieses Ziel abgestimmten Prozess entwickeln. Es reicht nicht, ein Programm kurz vor der Abgabe zum ersten Mal durchzumessen. Das ist so, als würden Sie ein Auto bauen, in das Sie am Auslieferungstag einsteigen, um zu sehen, wie schnell es denn fährt und ob die Geschwindigkeit der in der Werbebroschüre abgedruckten entspricht.

Es hat sich die Einsicht durchgesetzt, dass vielleicht Kleinstprogramme auf einen Schlag fertig gestellt werden können. Alle anderen müssen modularisiert sowie schrittweise verbessert und ausgebaut werden. Daher beruhen die meisten modernen Software-Entwicklungs-Methoden auf iterativer Verfeinerung und inkrementeller Erweiterung. So werden funktionierende Teilergebnisse geschaffen, die bereits einen Wert für den Kunden darstellen. Zudem können Probleme frühzeitig und somit rechtzeitig erkannt werden. Denn kurz vor der Abgabe ist es meistens zu spät und zu teuer, um das Projekt noch zu retten. Daraus folgt, dass die Performance eines Programms genau wie seine Funktion rechtzeitig und kontinuierlich getestet werden muss. Dies widerspricht gleich mehreren populären Weisheiten:

Rules of Optimization:
Rule 1: Don't do it.
Rule 2: (for experts only): Don't do it yet.

(M.A. Jackson)

More computing sins are committed in the name of efficiency (without necessarily achieving it) than for any other single reason – including blind stupidity.

(W.A. Wulf)

We should forget about small efficiencies, say about 97% of the time:
premature optimization is the root of all evil.

(Donald Knuth)

In allen drei Zitaten steckt ein Körnchen Wahrheit. Wir sollten jedoch nicht vergessen, dass die Performance eines Programms Teil der Anforderungen ist und daher nicht einfach unter den Tisch fallen darf oder *morgen* erledigt werden kann. Denn nachträglich optimierte Programme lehren einem häufig das Fürchten ...

Im Folgenden wird als Beispiel ein prototypischer Prozess umrissen. Halten Sie sich nicht sklavisch an diesen Prozess! Kein Prozess ist so gut, dass man ihn zur Bibel erklären sollte. Nutzen Sie ihn als Anregung und gebrauchen Sie ihn mit Verstand.[1]

Seine Essenz ist es, das Messen der Performance als Teil des funktionalen Testens und das Testen als Teil des Entwickelns zu begreifen sowie iterativ-inkrementell vorzugehen.

Messen ist Testen und Testen ist Entwickeln.

Bauen Sie in jeder Iteration kleine funktionierende Einheiten, die Sie in der nächsten Iteration erweitern oder verbessern. Beginnen Sie dabei mit den Einheiten, die für den Kunden den größten Wert darstellen.

1 Siehe auch *Manifesto for Agile Software Development – http://www.agilealliance.org/.*

Sie werden feststellen, dass im Folgenden keine Rollen oder Verantwortlichkeiten definiert werden, sondern nur Tätigkeiten bzw. Ziele. Das ist durchaus so gewollt. Gewöhnlich findet sich jemand für eine gegebene Aufgabe. Und vielleicht muss nicht jeder immer dasselbe machen. Wenn Sie das Know-how Ihrer Mitarbeiter als schützenswerte Investition betrachten, ist das sogar angebracht. Kommen Sie auf keinen Fall auf die Idee, Analysieren, Entwerfen und Kodieren strikt zu trennen und die Aufgaben verschiedenen Personen zuzuordnen, die sich evtl. nicht mögen oder gar räumlich und zeitlich voneinander getrennt arbeiten. Alle Projektphasen greifen ineinander und gehen ineinander über. Obwohl hier mehrere Schritte der Reihe nach vorgestellt werden, ist es eher hinderlich, sich immer genau an diese Reihenfolge zu halten. Stattdessen kann es durchaus Sinn machen, von Phase zwei direkt in Phase eins oder drei zu springen.

Folgendes sind die drei Hauptphasen des Prozesses:

- Analyse und Design
- Kodieren und Testen
- Integrieren und Testen

Könnte man Analysen und Designs verlässlich und mit vertretbarem Aufwand formal testen, ohne zuvor den Code schreiben zu müssen, so hieße die erste Phase *Analyse, Design und Testen*. Nun ist dem nicht so, was das Testen in den anderen beiden Phasen umso wichtiger macht. Daher gehe ich nach der Beschreibung der drei Prozess-Phasen noch einmal auf verschiedene Aspekte des Testens und Messens ein. Doch hier sind zunächst einmal die drei Phasen.

2.1 Analyse und Design

Je besser Sie Ihre Aufgabe verstehen, desto passender wird Ihr Design ausfallen. Finden Sie gemeinsam mit dem Kunden heraus, wie die Rahmenbedingungen aussehen (Betriebssystem, Leistung und Anzahl der Prozessoren, Bandbreite der Netzwerkanbindung, Bildschirmauflösung etc.). Fragen Sie nach allem, was irgendwie Einfluss auf die Leistungsfähigkeit der zu erstellenden Software haben könnte. Nur, wenn Sie die Laufzeitumgebung gut kennen, können Sie passende Software schreiben.

Versuchen Sie außer den regulären funktionalen Anforderungen möglichst genaue Performance-Anforderungen zu vereinbaren, beispielsweise indem Sie die Anwendungsfälle (Use-Cases) oder Stories[2] der regulären Analyse mit Laufzeit-Toleranzen für performancekritische Transaktionen versehen. Vergessen Sie diesen Schritt auf keinen Fall! Wenn Sie nicht wissen, was der Kunde wünscht, wissen Sie nicht, wann Sie

2 Stories sind das Extreme-Programming-Äquivalent zu Anwendungsfällen. Es handelt sich hierbei um die kurze Beschreibung eines Features, zu Papier gebracht auf einer Karteikarte.

fertig sind, somit ist Ärger vorprogrammiert. Betrachten Sie jedoch weder die funktionalen noch die Performance-Anforderungen als endgültig. Es liegt in der Natur der Anforderungen, dass sie sich ändern und Sie darauf reagieren müssen.

Das Wissen über die Laufzeitumgebung und die Anforderungen des Kunden versetzt Sie und Ihr Team in die Lage, ein Design zu erstellen, von dem Sie annehmen, dass es der Aufgabe gerecht wird. Dank Ihrer und der Qualifikation Ihrer Mitarbeiter haben Sie berechtigten Grund an das Design zu *glauben*. Verfallen Sie jedoch nicht dem Irrtum, dass Sie zu diesem Zeitpunkt *wissen*, dass Ihr Design die gesetzten Erwartungen erfüllt. Sie sollten daher erwägen, die kritischsten Teile Ihres Designs zu verifizieren, indem Sie zunächst einen Prototypen entwerfen und diesen für eine Machbarkeitsstudie verwenden. Vergessen Sie nicht, dass das Wichtigste am Prototypen ist, ihn nach Gebrauch wegzuschmeißen. Fangen Sie noch einmal von vorne an und Sie werden langfristig Zeit gewinnen.

Analyse und Design sind in einem iterativ-inkrementellen Prozess niemals eine einmalige Angelegenheit. Daher ist es selbstverständlich, dass Erkenntnisse aus allen anderen Phasen in der nächsten Iteration in die Analyse-und-Design-Phase einfließen. Dies gilt genauso für alle folgenden Phasen.

2.2 Kodieren und Testen

Während der Kodierungsphase wird das Design in Code gegossen. Hierzu gehört auch der entsprechende Testcode. Während dieser für einzelne Klassen und für die Komponenten-Integration von den Entwicklern selbst geschrieben wird, sollten die System-Tests vom Kunden erstellt werden, und zwar mit Unterstützung aus Ihrem Team.

Kent Beck und andere Verfechter agiler Methoden propagieren das Schreiben von Tests vor dem Schreiben des zu testenden Codes (*Test First*). Auf diese Weise müssen die Entwickler sich zunächst mit einer Außensicht auf ihre Software beschäftigen, sie machen sich zunächst Gedanken über das Benutzen und dann erst über das Implementieren. Dies führt in der Regel zu besseren Tests und somit zu besserem Code. Zudem wissen die Entwickler genau, wann sie fertig sind – nämlich wenn alle Tests laufen, insbesondere auch die älterer Programmteile.

Spätestens, wenn Sie über ein System verfügen, zu dem Sie Performance-Anforderungen haben, sollten Sie entsprechende Testtreiber für das System fertig gestellt haben. Je früher Sie Ihre Software testen, desto besser. Blicken Sie der Wahrheit ins Auge: Nicht zu wissen, ob man einen Speicherfresser oder CPU-Zyklen-Verbrenner erstellt hat, ist weitaus schlimmer, als es zu wissen.

Wenn Sie dank der präzisen Anforderungen Ihres Kunden oder Ihrer eigenen Messdaten genau wissen, dass Klasse *A* die Methode *B* auf System *C* in *D* Sekunden ausführen können muss, schätzen Sie sich glücklich. Sie können auf unterster Ebene neben dem funktionalen Unit-Test auch einen Performance-Unit-Test schreiben. Noch früher können Sie nicht testen.

Vermischen Sie dabei auf keinen Fall funktionale Tests mit Performance-Tests. Sie wollen in der Lage sein, sehr schnell das korrekte Funktionieren Ihrer Klassen nachzuweisen. Performance-Tests benötigen aber gewöhnlich etwas länger als funktionale Tests. Ungeduldige Entwickler (also fast alle) würden daher entsprechend seltener testen. Das Vermischen der beiden Testarten ist also kontraproduktiv.

2.3 Integrieren und Testen

In der Integrations- und Testphase wird der bereits von den Entwicklern während des Kodierens getestete Code zusammengeführt und einem Integrationstest unterzogen. Hier kommen die bereits erstellten funktionalen und leistungsorientierten Testtreiber zum Einsatz. Beachten Sie, dass Sie immer funktionale Tests benötigen und diese auch bei jeder Integration ausführen sollten.

Leistungstests sind nur etwas wert, wenn sichergestellt ist, dass die erbrachten Leistungen auch den funktionalen Anforderungen genügen.

Die gemessenen Resultate fließen natürlich in folgende Analyse-und-Design- und Kodierungsphasen ein. Wichtig ist dies insbesondere, wenn die Resultate nicht akzeptabel sind. Evtl. müssen Sie die gesamte Architektur überdenken. Wenn Sie frühzeitig mit dem Testen begonnen haben, sollte dies jedoch kein Problem sein.

2.3.1 Mikro- und Makro-Benchmarks

Wie schon angedeutet, sollten Sie Ihre Software Performance-Tests auf verschiedenen Ebenen unterziehen (Abbildung 2.2). Auf unterster Ebene stehen Performance-Unit-Tests (Mikro-Benchmarks), auf oberster Ebene stehen System-Performance-Tests (Makro-Benchmarks).

Angenommen, Sie erstellen eine Applikation, die unter anderem große Datenmengen in eine Protokoll-Datei schreiben muss. Daher entschließen Sie sich einen dedizierten Protokoll-Dienst zu schreiben. Ein sinnvoller Performance-Unit-Test wäre beispielsweise zu messen, wie schnell der Protokoll-Writer Ihres Dienstes 10.000 Einträge ausgeben kann. Mit Hilfe des Tests können Sie entsprechende Daten sammeln und die Protokoll-Writer-Klasse kontinuierlich verbessern.

Auf Systemebene jedoch ist das Schreiben von Protokollen nur ein Teilaspekt der Gesamtperformance. Ihre Testtreiber sollten daher nicht nur isolierte Eigenschaften Ihres Programms testen, sondern möglichst reale Szenarien nachempfinden. Diese Art des Testens wird auch als Makro-Benchmarking bezeichnet.

Eine gute Ausgangsbasis für Testszenarien sind die in der Analyse erstellten Anwendungsfälle oder Stories. Fragen Sie Ihren Kunden nach realistischen Testdaten und der relativen Häufigkeit der verschiedenen Abläufe. Besser noch, nehmen Sie Testdaten an einem bestehenden System auf. Entsprechende Werkzeuge können dabei sehr hilfreich sein (siehe *Kapitel 4.6 Makro-Benchmarks*). Verfüttern Sie dann diese Daten an Ihren Testtreiber. So können Sie realistische Tests durchführen und aussagekräftige Ergebnisse erhalten.

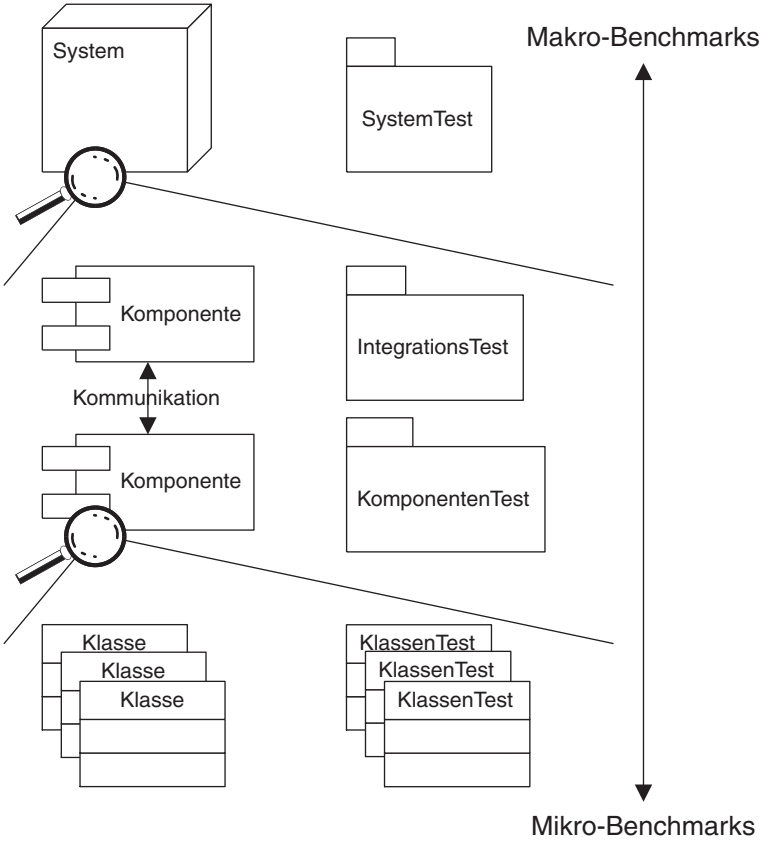

Abbildung 2.2: Verschiedene Testebenen in einem System. Die Paket-Ebene wurde ausgelassen, da ein Paket in Java nicht unbedingt eine semantische, sondern eher eine strukturelle Einheit ist und daher nicht direkt mit einem Funktions- oder Performance-Test assoziiert ist. In der Regel reicht es, alle Klassentests eines Pakets auszuführen, um das Paket zu testen.

Unter Umständen werden Sie beim Testen feststellen, dass Sie zwar verhältnismäßig schnell Protokolldaten schreiben können, dies aber leider immer in Momenten passiert, in denen der Benutzer gerade etwas eingeben muss, jedoch nicht kann, da das System mit den Protokolldaten beschäftigt ist.

Oder Sie stellen fest, dass Sie zwar in der Lage sind, sehr schnell sehr viele Protokoll-Ereignisse zu verarbeiten, diese jedoch über eine sehr langsame Schnittstelle angeliefert werden. Somit nützt es Ihnen wenig, die Daten schnell schreiben zu können. Der Flaschenhals sitzt an anderer Stelle.

Mikro-Benchmarks sind sehr nützlich zum Optimieren kleiner, isolierter Programmteile. Sie zeigen Ihnen aber immer nur einen Ausschnitt des Gesamtbildes. Makro-Benchmarks hingegen sind zum Optimieren Ihres Codes oft nutzlos, liefern Ihnen jedoch wichtige Daten, wenn es um das Optimieren des Systems inklusive VM und Hardware geht. Beide Testarten ergänzen sich also.

2.3.2 Testlänge

Sowohl für Mikro- als auch für Makro-Benchmarks gilt, dass Testläufe nicht zu kurz sein dürfen, um aussagekräftige Ergebnisse zu produzieren. Mikro-Benchmarks sollten mindestens fünf Sekunden laufen, Makro-Benchmarks einiges länger. Beachten Sie, dass Teile Ihrer Laufzeitumgebung wie Caches oder die Java VM erst nach einer gewissen Laufzeit ihre volle Leistung entfalten. Beispielsweise konnten frühe HotSpot-Versionen Methoden, die nur einmal ausgeführt werden, nur interpretieren. Immer noch gilt, dass der adaptive Compiler erst nach einer gewissen Laufzeit zum Zuge kommt.

2.3.3 Auswertung

Führen Sie außerdem immer mehrere Testläufe durch. So können Sie sich ein Bild von der Streuung der Messergebnisse machen. Statistische Werte, die Sie auf jeden Fall erheben sollten, sind:

- Bestes Ergebnis
- Schlechtestes Ergebnis
- Durchschnittliches Ergebnis

Darüber hinaus können Maßzahlen wie Standardabweichung, Median, geometrisches Mittel etc. das Vergleichen der Ergebnisse mehrerer Testsessions erheblich erleichtern. Ein handelsübliches Tabellenkalkulations-Programm kann bei der Analyse der gemessenen Werte nützliche Dienste erweisen.

2.3.4 Wie häufig testen?

Leistungstests sollten dem Aufwand angemessen häufig durchgeführt werden. Handelt es sich beispielsweise um ein verteiltes System und der Kunde kann oder will Ihnen kein geeignetes Testsystem zur Verfügung stellen, müssen Sie vielleicht die Entwicklermaschinen zu Testzwecken missbrauchen. Gewöhnlich sind Entwickler nicht besonders begeistert, wenn sie auf einer Maschine arbeiten müssen, die gerade an einem verteilten Lasttest teilnimmt. Also müssen Mittagspausen, Abende und Wochenenden herhalten. Niemand arbeitet gerne zu diesen Zeiten. Wenn Sie die Stimmung in Ihrem Team nicht allzu sehr strapazieren wollen, testen Sie nicht zu häufig und versuchen Sie stattdessen gemeinsam mit dem Kunden die Testbedingungen zu verbessern.

Anders sieht es aus, wenn Sie erkennen, dass Sie ein echtes Performance-Problem haben. Es kann sich durchaus lohnen, mehrere komplette Releasezyklen nur an der Performance zu arbeiten. Je besser Sie messen, desto größer sind die Chancen, dass Sie das Problem verstehen und eine gute Lösungs-Strategie entwickeln.

Es gilt also Augenmaß zu bewahren. Nochmals: Leistungstests machen nur bei funktionierenden Softwareeinheiten Sinn. Zu viele Leistungstests führen leicht dazu, dass die funktionale Qualität leidet, zu wenig Leistungstest führen zum späten Erkennen von Problemen. Es ist also der goldene Mittelweg gefragt.

3 Virtuelle Maschinen

Um zu wissen, wie man einen Motor frisiert, sollte man wissen, wie er funktioniert. Daher wollen wir uns in diesem Kapitel ein wenig mit Javas Virtueller Maschine beschäftigen.

Die Java VM ist eine abstrakte Maschine. Doch genau wie ein echter Prozessor hat sie einen Befehlssatz und manipuliert zur Laufzeit verschiedene Speicherbereiche. Sie setzt jedoch keinerlei spezifische Hardware voraus, sondern wird emuliert. Der emulierte Prozessortyp ist eine Kellermaschine mit mehreren Stacks.

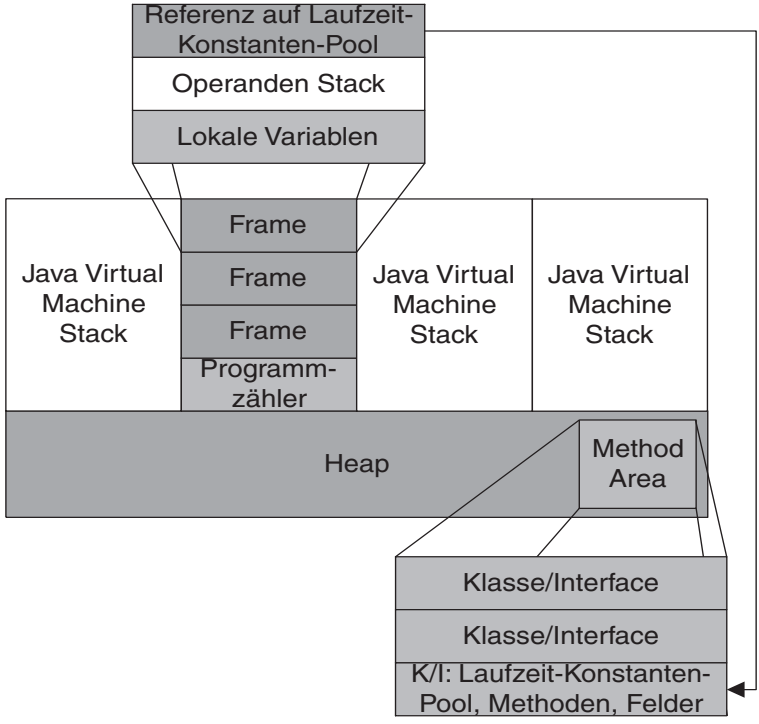

Abbildung 3.1: Schematischer Überblick über die Java VM

Die Grundbestandteile der VM sind jeweils ein Java Virtual Machine Stack pro Thread[1] sowie der von allen Threads geteilte Heap-Speicher (Abbildung 3.1). Im Heap werden sämtliche Objekt-Instanzen sowie Klassen und Interfaces gehalten. Letztere befinden sich dabei in einem speziellen Bereich namens Method-Area. Der Heap wird automatisch durch die Speicherbereinigung verwaltet.

Jeder Stack enthält eine Reihe von so genannten Frames. Während der Ausführung eines Java-Programms wird für jeden Methodenaufruf ein Frame angelegt, auf den Stack gelegt und nach Ausführung der Methode wieder heruntergenommen. Ein Frame enthält bzw. verweist auf alle wichtigen Daten, die zur Ausführung der Methode nötig sind. Zu jedem Stack existiert zudem ein Programmzähler, der auf die Adresse des gerade ausgeführten Codes zeigt.

Vereinfacht dargestellt wird beim Start der VM die zu startende Klasse geladen, ein Thread initialisiert und dessen Programmzähler auf den Beginn der statischen `main()`-Methode der Klasse gesetzt. Dann werden nacheinander die im Bytecode der `main()`-Methode enthaltenen VM-Befehle und somit das Programm ausgeführt.

Absichtlich beschreibt die Java-VM-Spezifikation kaum Implementierungsdetails und lässt VM-Herstellern viele Freiheiten. Die Hauptunterscheidungsmerkmale zwischen heute erhältlichen VMs liegen in der Art und Weise, wie der Java-Bytecode ausgeführt und wie der Heap verwaltet wird. Im Folgenden werden verschiedene Strategien für beides erläutert. Darüber hinaus werden wir kurz auf verschiedene Leistungstests eingehen, die bei der Entscheidung für oder gegen eine VM hilfreich sein können.

3.1 Bytecode-Ausführung

Java-Bytecode lässt sich auf verschiedene Weisen ausführen. Dazu gehören:

- Interpretieren
- Unmittelbar vor der Ausführung in Binärcode übersetzen (Just-in-Time, JIT) und dann den Binärcode ausführen
- Sofort nach dem Erstellen in Binärcode übersetzen und diesen später ausführen (Ahead-of-Time, AOT)
- Teilweise interpretieren und erst wenn notwendig dynamisch in Binärcode übersetzen (Dynamic-Adaptive-Compilation, DAC) und dann diesen ausführen
- Von einer in Silizium gegossenen VM ausführen lassen

Im Folgenden gehen wir kurz auf die einzelnen Möglichkeiten ein.

1 Für `native` Methodenaufrufe kann es zudem einen Native-Stack pro Thread geben.

3.1.1 Interpreter

In Javas frühen Tagen wurde Java ausschließlich interpretiert. Heute werden reine Interpreter für die Java 2 Standard Edition (J2SE) kaum noch verwendet. Der Grund liegt in ihrer Langsamkeit. Dennoch können Interpreter sehr nützlich sein, da sie bei Ausnahmen in Stacktraces gewöhnlich die Klasse, Methode und Zeilenzahl angeben. Das macht sie zu einem wertvollen Werkzeug für die Fehlersuche.

Ein weiterer Vorteil von Java-Interpretern ist ihr geringer Speicherverbrauch. Daher sind sie insbesondere für Kleingeräte wie PDA (Personal Digital Assistent) und Mobiltelefone interessant.

3.1.2 Just-in-Time-Compiler

Nachdem man merkte, dass mit interpretiertem Java nicht viel zu gewinnen war, kamen JIT-Compiler in Mode. JIT-Compiler übersetzen Bytecode unmittelbar vor der Ausführung in plattformspezifischen Binärcode. VMs mit JIT sind in der Regel sehr viel schneller als VMs, die nur über einen Interpreter verfügen. Jedoch führen JITs gegenüber Interpretern auch zu einem höheren Speicherverbrauch. Zudem ist die Qualität des erzeugten Binärcodes nicht mit der Qualität von Binärcode vergleichbar, der von einem optimierenden, statischen Compiler erzeugt wird. Dies liegt daran, dass die Übersetzung zur Laufzeit und somit sehr schnell erfolgen muss. Viel Zeit für langwierige Analysen und Optimierungen bleibt da nicht. Außerdem ist aufgrund der Übersetzung die Startzeit von VMs mit JITs in der Regel länger als die von VMs mit Interpretern.

Eine sehr erfolgreiche VM mit JIT-Technologie wird von IBM produziert.

- IBM Java VM: *http://www.ibm.com/java/*

3.1.3 Dynamisch angepasste Übersetzung

Suns aktuelle Java VMs werden mit HotSpot-Technologie ausgeliefert. Die Idee von HotSpot beruht auf dem 80-20-Prinzip. Dieses besagt, dass während 80% der Laufzeit eines Programms gewöhnlich 20% des Codes ausgeführt werden. Daraus folgt, dass es sich lohnt, genau jene 20% besonders schnell auszuführen, während Optimierungen in den restlichen 80% des Codes keinen großen Effekt haben.

Dementsprechend interpretiert HotSpot zunächst den Bytecode und analysiert zur Laufzeit, welche Programmteile in Binärcode übersetzt und optimiert werden sollten. Diese Teile werden dann übersetzt und mehr oder minder aggressiv optimiert. Im Englischen heißen die besonders kritischen Programmteile auch Hotspots – daher der Name. Das gesamte Verfahren heißt Dynamic-Adaptive-Compilation, kurz DAC.

Verfechter von DAC behaupten gerne, dass durch die adaptive Kompilierung bessere Resultate erzielt werden können als durch statisches Übersetzen. In der Theorie ist das

auch durchaus richtig. In der Praxis zeigt sich jedoch, dass außer der reinen Ausführung von Code noch viele andere Faktoren auf die Geschwindigkeit einer Java VM Einfluss haben. Insbesondere ist das die automatische Speicherbereinigung. Daher hinken Vergleiche mit statisch kompilierten Programmen ohne automatische Speicherbereinigung meist.

Zudem ist die adaptive, optimierende Übersetzung mit vorheriger Laufzeitanalyse ein recht aufwändiger Prozess, der sich nur für lang laufende Programme lohnt. Aus diesem Grund unterscheidet Sun die Client HotSpot VM von der Server HotSpot VM. Die Client-Version führt nur recht simple Optimierungen aus und übersetzt daher relativ schnell. Die Server-Version hingegen enthält einen volloptimierenden Compiler (Tabelle 3.1). Gemessen an JIT-Standards ist dieser Compiler langsam. Dies kann sich jedoch bei langer Laufzeit durch den aus Optimierungen resultierenden Zeitgewinn bezahlt machen.

Optimierung	**Beschreibung**	**JDK 1.3.1 Client**	**JDK 1.3.1 Server**
Range Check Elimination	Unter bestimmten Bedingungen entfällt der obligatorische Check, ob ein Array-Index gültig ist. Siehe *Kapitel 6.2.2 Teure Array-Zugriffe*	nein	ja
Null Check Elimination	Unter bestimmten Bedingungen entfällt der obligatorische Check, ob eine Array-Variable `null` ist. Siehe *Kapitel 6.2.2 Teure Array-Zugriffe*	nein	ja
Loop Unrolling	Mehrfaches Ausführen eines Schleifenkörpers ohne Überprüfung der Abbruchbedingung. Siehe *Kapitel 6.2.3 Loop Unrolling*	nein	ja
Instruction Scheduling	Neuordnung von Befehlen zur optimalen Ausführung auf einem bestimmten Prozessor	nein	ja (für UltraSPARC III)
Optimierung für Reflection API	Schnellere Ausführung von Methoden aus dem Reflection API (`java.lang.reflect`)	nein	ja
Dynamische Deoptimierung	Umkehroperation für eine Optimierung	nein	ja
Einfaches Inlining	Duplizieren und Einfügen von Methoden an Stellen, an denen diese ausgeführt werden. Der Overhead eines Methodenaufrufs entfällt somit. Einfaches Inlining trifft nur auf Klassenmethoden bzw. `final`-Methoden zu.	ja	nein
Vollständiges Inlining	Siehe Einfaches Inlining. Jedoch werden wesentlich mehr Methoden dupliziert und eingefügt.	nein	ja

Tabelle 3.1: Vergleich von Optimierungen der HotSpot Server und Client Compiler für Sun JDK 1.3.1

Optimierung	Beschreibung	JDK 1.3.1 Client	JDK 1.3.1 Server
Dead Code Elimination	Code, der nicht ausgeführt werden kann, wird entfernt.	nein	ja
Loop Invariant Hoisting	Variablen, die sich während einer Schleife nicht ändern, werden außerhalb der Schleife berechnet. Siehe *Kapitel 6.2.1 Loop Invariant Code Motion*	nein	ja
Common Subexpression Elimination	Die Ergebnisse einmal berechneter Ausdrücke werden wieder verwendet.	nein	ja
Constant Propagation	Konstanten werden durch ihre Werte ersetzt.	nein	ja
On Stack Replacement (OSR)	Während der Ausführung einer Methode kann interpretierter Code durch kompilierten ersetzt werden.	ja	ja

Tabelle 3.1: Vergleich von Optimierungen der HotSpot Server und Client Compiler für Sun JDK 1.3.1 (Fortsetzung)

Besonders interessant im Zusammenhang mit Java sind die Fähigkeiten *Inlining* und *Dynamische Deoptimierung*. Inlining ist eine Optimierung, bei der ein Methodenaufruf durch den Code der aufzurufenden Methode ersetzt wird. Dadurch wird der Verwaltungsaufwand für den Methodenaufruf, sprich das Anlegen und Beseitigen eines Frames, gespart. Während statische Compiler für Sprachen wie C relativ einfach Inlining anwenden können, ist das in Java nicht so einfach. Denn im Gegensatz zu C-Funktionsaufrufen sind in Java die meisten Methodenaufrufe *virtuell*, das heißt potenziell polymorph. Somit kann der Aufruf der Methode `doIt()` eines Objektes, das durch eine Referenz vom Typ `A` referenziert wird, durchaus zur Ausführung einer überschriebenen Version der Methode `doIt()` eines Objekts von `A`s Subtyp `B` führen. Listing 3.1 zeigt ein einfaches Beispiel für diesen Fall.

```
class A {
   public doIt() {
      System.out.println("doIt() von A");
   }
}
class B extends A {
   public doIt() {
      System.out.println("doIt() von B");
   }
   public static void main(String[] args) {
      A a = new B();
      a.doIt(); // führt doIt() der Klasse B aus!
   }
}
```

Listing 3.1: Beispiel für eine überschriebene Methode

Um die Sache noch ein wenig komplizierter zu machen, ist es in Java möglich, Klassen dynamisch zur Laufzeit nachzuladen. Somit ist das gefahrlose Inlining einer Methode nur möglich, wenn die einzureihende Methode entweder `final` oder `static` ist, da dann die Methode nicht von einer Unterklasse überschrieben werden kann. Die Fähigkeit `static` und `final` Methoden einzureihen wird in Tabelle 3.1 unter einfachem Inlining aufgeführt.

Mit vollständigem Inlining ist gemeint, dass auch Methoden, die nicht `final` oder `static` sind, eingeschoben werden können. Dies ist dann möglich, wenn keine Klasse geladen ist, die die fragliche Methode überschreibt. Da in Java aber auch zur Laufzeit noch Klassen geladen werden können, kann sich dieser Zustand ändern. Ist dies der Fall, so muss eine bereits eingereihte Methode evtl. wieder durch einen regulären Methodenaufruf ersetzt werden. Dies ist ein Beispiel für dynamische Deoptimierung. JIT-Compiler, die den Code in der Regel nur einmal übersetzen, sind zu solchen (De-)Optimierungen meist nicht in der Lage.

Neben dem gesparten Verwaltungsaufwand für Methodenaufrufe bietet vollständiges Inlining noch einen weiteren Vorteil: Die entstehenden längeren Code-Blöcke erlauben weitere Optimierungen.

Suns HotSpot-Technologie wurde von Hewlett Packard und Apple lizenziert. Daneben gibt es außerdem noch andere DAC-VMs. Eine davon ist die freie VM *JRockit* von Appeal Virtual Machines (BEA). Es handelt sich dabei um eine Java Laufzeitumgebung mit Fokus auf serverseitige Applikationen. Sun bemüht sich zudem, HotSpot auch für die J2ME-Plattform anzubieten.

- Sun J2SE: *http://java.sun.com/*
- Appeal Virtual Machines: *http://www.jrockit.com/*

3.1.4 Ahead-of-Time-Übersetzung

Anstatt Code erst zur Laufzeit in Binärcode zu übersetzen, kann man auch bereits beim Erstellen der Software Binärcode für die Zielplattform erzeugen. Dieser Ansatz wird auch als Ahead-of-Time-Übersetzung (AOT) bezeichnet.

AOT ermöglicht sehr gute Optimierungen und führt in der Regel zu robustem, schnellem Code. Dies liegt unter anderem daran, dass bereits vorhandene, ausgereifte Compiler zur Code-Generierung und -Optimierung genutzt werden können. So ist beispielsweise der freie GNU Java Compiler *GJC* lediglich ein Frontend zu anderen GNU Compilern. Dementsprechend wird Java von den GJC-Autoren auch nur als Teilmenge von C++ betrachtet.

Ein weiter Grund für die gute Performance von AOT sind eine im Vergleich zu JIT und DAC fast beliebig lange Code-Analyse-Phase, beinahe beliebiger Ressourcenverbrauch während der Übersetzung und die daraus resultierenden Optimierungen. Der kom-

merzielle Java Native Compiler *TowerJ* beispielsweise kann während des Kompilierens so genannte ThreadLocal-Objekte erkennen. ThreadLocals sind Objekte, die garantiert nur von einem Thread benutzt werden. Somit ist jegliche Synchronisation überflüssig, die Objekte können im schnelleren, threadspezifischen Stack statt im Heap gespeichert werden und unterliegen somit auch nicht der normalen Speicherbereinigung. Der erzeugte Code ist entsprechend schneller.

Zudem kann Binärcode nur schlecht dekompiliert werden und erschwert Reverse Engineering[2] erheblich. Somit ist gegenüber Bytecode ein besserer Schutz von geistigem Eigentum gewährleistet.

Alle AOT-Compiler haben jedoch mit Javas Fähigkeit zu kämpfen, Klassen dynamisch nachzuladen. Dies ist beispielsweise notwendig für Remote Method Invocation (RMI), Dynamische Proxies (`java.lang.reflect.Proxy`), Java Server Pages (JSP) etc. TowerJ löst dieses Problem, indem während der ersten Ausführung aufgezeichnet wird, welche Klassen nachgeladen und von einem eingebauten Interpreter ausgeführt werden mussten. In einem folgenden Übersetzungslauf werden diese Klassen dann ebenfalls in Binärcode übersetzt und optimiert.

Einen etwas anderen Weg beschreitet *Excelsiors JET*. JET verfügt über die Fähigkeit, zur Laufzeit nachgeladene Klassen mit einem JIT-Compiler zu übersetzen und als DLLs einzubinden. Dabei ist der JIT-Compiler selbst eine DLL und kann dementsprechend nach getaner Arbeit wieder aus dem Speicher entfernt werden.

Zusammenfassend lässt sich sagen, das AOT eine legitime Strategie ist, um schnelle und verlässliche Java-Programme für eine spezifische Zielplattform zu erzeugen. Die erzeugten Programme sind natürlich nicht portabel. Jedoch spricht nichts dagegen, neben verschiedenen nativen Versionen auch eine portable Bytecode-Version Ihrer Software auszuliefern.

Hier eine Auswahl von AOT-Compilern:

- TowerJ: *http://www.towerj.com/*
- NaturalBridge BulletTrain: *http://www.naturalbridge.com/*
- Excelsiors JET: *http://www.excelsior-usa.com/jet.html*
- Instantiations JOVE: *http://www.instantiations.com/jove/*
- GCJ ist noch in den frühen Phasen der Entwicklung: *http://gcc.gnu.org/java/*

2 Vorgang, bei dem aus Bytecode ein (visuelles) Modell erzeugt wird.

3.1.5 Java in Silizium

Natürlich ist man auch auf die Idee gekommen, aus der Java Virtuellen Maschine eine reale Java Maschine zu machen; mit anderen Worten, die VM in Silizium zu gießen. Dies ist insbesondere für den Mobiltelefon- und PDA-Markt interessant, der J2ME verwendet, und weniger für Geschäftsanwendungen auf Basis von J2SE oder J2EE. Der Vollständigkeit halber möchte ich kurz auf das Thema eingehen – für das Buch wird es im Weiteren nicht von Belang sein.

Schon 1996 hat Sun einen Prozessor-Kern namens *picoJava* spezifiziert. Dieser wurde auch von mehreren großen Firmen lizenziert, ein picoJava-Boom blieb jedoch aus. Keiner der Lizenznehmer hat jemals picoJava-basierte Chips verkauft.

Bereits zum Erscheinen der Spezifikation wurde der Ansatz kritisch beäugt. Wissenschaftler hatten schon für andere Sprachen wie LISP und Smalltalk Spezial-Prozessoren entwickelt, nur um zu entdecken, dass Software-Implementierungen auf RISC-Chips bessere Performance boten. Man zweifelte daran, dass Suns picoJava besser performte. Und tatsächlich stellte sich später heraus, dass picoJava weder schnell noch billig noch sparsam genug war, um im Markt für Mobiltelefone und PDAs mithalten zu können.

Stattdessen wurde in letzter Zeit ein etwas anderer Ansatz für Kleingeräte populär: Java-Beschleuniger. Dabei handelt es sich um Bausteine, die ähnlich wie Koprozessoren zusätzlich zum Hauptprozessor verwendet werden können. So lässt sich beispielsweise *Nazomis* Java-Koprozessor in bestehende Designs einbinden und erleichtert so Kleingeräte-Herstellern die Verwendung von Java unter Beibehaltung einer bereits vorhandenen Architektur.

Einen anderen Weg ging die Firma *ARM*. ARM hat seinen Chips den Java-VM-Befehlssatz schlicht als dritten Befehlssatz hinzugefügt. Ein einfaches Umschalten macht so aus dem herkömmlichen ARM-Chip eine Java VM.

- Suns picoJava: *http://www.sun.com/microelectronics/picoJava/*
- ARM: *http://www.arm.com/*
- Nazomi: *http://www.nazomi.com/*

3.2 Garbage Collection

Neben der Bytecode-Ausführung ist der andere entscheidende Aspekt für die Performance einer VM die Garbage Collection. Weder in der Java Sprachspezifikation [Gosling00] noch in der Java VM Spezifikation [Lindholm99] sind genaue Vorgaben für die Garbage Collection zu finden. Es steht den VM-Herstellern somit größtenteils frei, wie sie die Speicherverwaltung implementieren.

3.2.1 Objekt-Lebenszyklus

Um besser zu verstehen, was die Aufgabe der Speicherverwaltung ist, wollen wir uns den Lebenszyklus eines Objektes anschauen. Abbildung 3.2 gibt eine Übersicht.

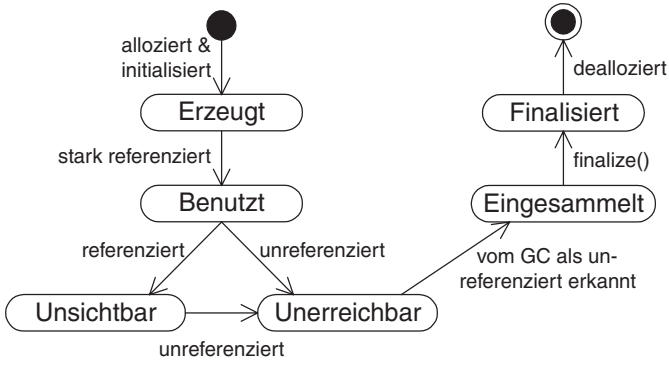

Abbildung 3.2: Lebenszyklus eines Objekts

Erzeugt

Es wurde Speicher für das Objekt alloziert und alle Konstruktoren sind ausgeführt worden. Das Objekt befindet sich also fertig initialisiert im Heap. Es wurde 'Erzeugt'.

Benutzt

Es existiert mindestens eine stark, vom Programm sichtbare Referenz auf das Objekt. Schwache, weiche und Phantom-Referenzen aus dem Paket `java.lang.ref` können zudem existieren, reichen aber nicht aus, um ein Objekt im Zustand 'Benutzt' zu halten.

Unsichtbar

Ein Objekt ist 'Unsichtbar', wenn keine starken Referenzen mehr existieren, die vom Programm benutzt werden könnten, trotzdem aber noch Referenzen vorhanden sind.

Nicht jedes Objekt durchläuft diesen Zustand. Er tritt aber zum Beispiel auf, wenn in einem Stackframe noch eine Referenz auf ein Objekt vorhanden ist, obwohl diese Referenz in einem abgeschlossenen Block deklariert und dieser Block bereits verlassen wurde. Listing 3.2 zeigt ein Beispiel.

```
public void do() {
   try {
      Integer i = new Integer(1);
      ...
   }
```

```
    catch (Exception e) {
       ...
    }
    while (true) {
       // äußerst lange Schleife
    }
}
```

Listing 3.2: In der while-Schleife ist i unsichtbar.

Das Objekt `i` wird in einem geschlossenen `try-catch`-Block alloziert. Eine effiziente VM-Implementierung wird die im Frame allozierte Referenz auf `i` jedoch nicht beim Verlassen des Blocks beseitigen, sondern erst, wenn der entsprechende Frame vom Stack genommen wird [Wilson00, S.196]. Daher ist `i` in der folgenden `while`-Schleife *sowohl* unsichtbar *als auch* referenziert und kann somit nicht vom Garbage Collector erfasst werden.

Unerreichbar

'Unerreichbar' sind jene Objekte, die nicht mehr von einer Objektbaumwurzel über Navigation zu erreichen sind. Zu den Objektbaumwurzeln gehören:

- Klassenvariablen (static)
- Temporäre Variablen auf dem Stack (lokale Methoden-Variablen)
- Besondere Referenzen von JNI-Code aus

Ist ein Objekt unerreichbar, so kann es vom Garbage Collector zu einem beliebigen späteren Zeitpunkt eingesammelt werden.

Eingesammelt

Ein Objekt ist dann 'Eingesammelt', wenn der Garbage Collector das Objekt als Garbage erkannt hat und es in die Warteschlange des Finalizer-Threads eingestellt hat. Ist die `finalize()`-Methode des Objekts nicht überschrieben, wird dieser Schritt übersprungen und das Objekt gelangt direkt in den Zustand 'Finalisiert'.

Finalisiert

'Finalisiert' ist ein Objekt, nachdem die `finalize()`-Methode aufgerufen wurde, sofern diese vorhanden ist bzw. überschrieben wurde. Beachten Sie, dass die `finalize()`-Methode meist in einem Extra-Thread, dem so genannten Finalizer-Thread ausgeführt wird. Falls die Threads Ihrer Applikation mit höherer Priorität laufen als der Finalizer-Thread und Sie die `finalize()`-Methode mit spezieller Aufräumlogik überschrieben haben, kann es sein, dass Sie die Garbage Collection blockieren, da der Finalizer-Thread nicht zum Zuge kommt und somit Objekte nicht dealloziert werden können.

Daher ist es grundsätzlich besser, sich nicht auf den Finalizer zu verlassen, sondern stattdessen eigene Lebenszyklus-Methoden zu implementieren und diese kontrolliert aufzurufen. Beispielsweise sollten Sie immer die Methode `dispose()` eines `java.awt.Graphics`-Objektes aufrufen, wenn Sie es nicht mehr benötigen, da sonst erst der Finalizer wichtige Ressourcen freigibt.

Dealloziert

Ein Objekt ist 'Dealloziert', wenn es nach der Finalisierung immer noch unerreichbar war und somit beseitigt werden konnte. Wann dies geschieht, liegt im Ermessen der VM.

3.2.2 Garbage Collection-Algorithmen

Sie haben gesehen, dass Objekte erst dealloziert werden können, wenn sie finalisiert und unerreichbar sind. Wie die Unerreichbarkeit bestimmt und der verfügbare Speicher verwaltet wird, hängt vom verwendeten Gargabe Collector ab. Dieser lässt sich meist über VM-Parameter beeinflussen und optimieren. Deshalb wollen wir uns kurz mit verschiedenen Garbage Collection-Algorithmen auseinander setzen.

Kopierender Kollektor

Ein einfacher *Kopierender Kollektor* unterteilt den Speicher in zwei Hälften. Er alloziert so lange Objekte in der ersten Hälfte, bis diese voll ist. Dann besucht er, von den Objektbaumwurzeln ausgehend, alle lebendigen Objekte und kopiert sie in die zweite Speicherhälfte.[3] Die nicht mehr referenzierten Objekte werden gelöscht. Anschließend tauschen die Speicherhälften ihre Rollen. Im verbliebenen Speicher der zweiten Hälfte werden nun neue Objekte alloziert, bis diese voll ist. Dann werden die lebendigen Objekte wiederum in die erste Hälfte kopiert usw.

Kopierende Kollektoren führen zu sehr schnellen Allokationszeiten, da der Speicher nicht fragmentiert wird. Tatsächlich reicht es aus, einfach einen Zeiger auf die Speicherzelle nach dem zuletzt allozierten Objekt zu pflegen. Viel schneller kann man Speicher nicht allozieren. Dieser Komfort hat jedoch seinen Preis, denn der Speicherverbrauch einfacher kopierender Kollektoren ist doppelt so groß wie der anderer Kollektoren.[4] Zudem fällt die Interaktion mit dem Speicherverwaltungssystem des Betriebssystems eher ungünstig aus. In jedem Kollektionszyklus muss jede Speicherseite einer Hälfte geladen und komplett beschrieben werden. Falls der gesamte Heap nicht in den realen

3 Wegen dieses Verhaltens werden kopierende Kollektoren auch *Scavengers* (Aasfresser) genannt – sie nehmen, was noch zu gebrauchen ist.

4 Hier ist nur der simpelste Kopier-Algorithmus beschrieben. Natürlich ist es möglich, den Speicher in mehr als zwei Teile zu unterteilen, jedoch pro Kollektionszyklus nur zwei dieser Teile zu betrachten [Jones96, S.127] und somit den Speicherverbrauch zu reduzieren.

Speicher des Rechners passt, führt dies unweigerlich zu teuren Seitenfehlern. Ein weiterer Nachteil ist, dass alle Objekte ständig im Speicher umgruppiert werden, was wiederum zu schlechter Lokalität führen kann.

Somit eignen sich kopierende Kollektoren insbesondere für kleine Speicherbereiche mit kurzlebigen Objekten.

Mark & Sweep

Der *Mark-Sweep*-Algorithmus funktioniert folgendermaßen: Ausgehend von den Objektbaumwurzeln werden alle erreichbaren Objekte besucht und mit einem Bit markiert. Wurden so alle lebendigen Objekte markiert, werden alle nicht-markierten Objekte aus dem Speicher gekehrt.

Mark-Sweep-Kollektoren haben gegenüber kopierenden Kollektoren einen geringeren Speicherverbrauch. Dafür neigen sie jedoch dazu, den Speicher zu fragmentieren. Das bedeutet auch, das die Verwaltung des freien Speichers umständlich ist, was wiederum dazu führt, dass die Allokation von Speicher länger dauert. Zudem kann bedingt durch die Fragmentierung der Speicher nicht vollständig genutzt werden und es kann passieren, dass Objekte, die kurz nacheinander alloziert wurden, nicht nahe beieinander im Heap liegen. Diese schlechte Lokalität führt wiederum zu Seitenfehlern.

Der Mark-Sweep-Algorithmus selbst ist simpel und schnell, führt aber zu Problemen insbesondere bei Systemen mit virtuellem Speicher. Er ist die Grundlage für den *Mark-Compact*-Algorithmus.

Mark & Compact

Genau wie beim Mark-Sweep-Algorithmus werden zunächst alle lebendigen Objekte markiert. Anschließend werden die lebendigen Objekte so im Speicher verschoben, dass es nur noch zwei Bereiche gibt: einen durchgängig mit Objekten belegten Bereich und einen freien Bereich. Der Heap wurde kompaktiert. Das heißt zur Verwaltung des freien Speichers genügt ein einziger Zeiger, der auf das Ende des belegten Bereichs zeigt. Dies wiederum bedeutet schnelle Allokation, da nicht umständlich nach einem freien Stück Speicher der gewünschten Größe gefahndet werden muss.

Je nach verwendetem Algorithmus kann während des Kompaktierens die Ordnung der Objekte beibehalten werden. Dies führt zu guter Lokalität und weniger Seitenfehlern. Auch das hat jedoch seinen Preis. Während Mark-Sweep-Kollektoren nach der Markierungsphase nur einmal durch den Heap wandern, müssen Mark-Compact-Algorithmen den Heap meist zwei- oder dreimal durchkämmen. Die Kollektion dauert also länger, führt aber anschließend zu einem besseren Laufzeitverhalten.

Inkrementelle und nebenläufige Speicherbereinigung

Alle bisher vorgestellten Verfahren gehen jeweils davon aus, exklusiv auf den Heap zugreifen zu können. Das bedeutet, dass das Programm während der Kollektion gestoppt wird. Dies führt zu unangenehmen Pausen, in denen die Applikation nicht auf Eingaben reagiert. Diese Pausen sind für Echtzeitanwendungen (z.B. Audio-/ Video-Anwendungen, Maschinen-Steuerungssoftware) nicht akzeptabel. Die Pausen sind zudem umso länger, je größer der Heap ist.

Inkrementelle und nebenläufige Speicherbereinigung versuchen, die Pausen auf ein Minimum zu reduzieren bzw. ganz zu beseitigen. Im Fall der inkrementellen Speicherbereinigung wird jeweils nur ein kleiner Teil des Heaps von unreferenzierten Objekten gesäubert und dann die Ausführung des Programms fortgesetzt. Bei nebenläufiger Speicherbereinigung wird parallel zur Programmausführung jeweils ein Teil des Heaps gereinigt. Dabei treten gewöhnlich Synchronisationsprobleme auf, die zu einem gewissen Verwaltungsaufwand führen.

Inkrementelle oder nebenläufige Kollektoren reduzieren Pausen, führen ansonsten aber meist zu einer schlechteren Performance als andere Verfahren.

Generationen-Kollektoren

Generationen-Kollektoren gehen davon aus, dass einige Objekte länger leben als andere. Weiterhin wird angenommen, dass die meisten Objekte jung sterben. Dementsprechend wird der Heap in mehrere Bereiche – Generationen – unterteilt. Neue Objekte werden grundsätzlich in der jungen Generation alloziert. Ist kein Speicher mehr in diesem Bereich vorhanden, wird er mittels eines der oben geschilderten Algorithmen aufgeräumt. Objekte, die eine bestimmte Anzahl von Aufräumphasen überleben, werden in die ältere Generation verschoben. Dabei müssen alle Zeiger auf das Objekt entsprechend angepasst werden.

Die ältere Generation wird genau wie die jüngere bei Bedarf aufgeräumt. Grundsätzlich sind mehr als zwei Generationen denkbar.

Generationen optimieren die Speicherbereinigung insofern, als dass nicht immer der ganze Speicher aufgeräumt wird, sondern nur der Teil, der gerade vollgelaufen ist. Somit sind die Pausen, die auftreten, wenn das Programm zur Speicherbereinigung gestoppt werden muss, relativ geringer als bei nur einem großen zusammenhängenden Speicherbereich, der immer komplett gesäubert werden muss. Dabei macht man sich zunutze, dass ein kleiner Speicherteil gewöhnlich sehr schnell mit Objekten belegt ist, die zu einem großen Teil direkt wieder aus dem Speicher entfernt werden können.

3.2.3 Performance-Maße

Zum Beurteilen der Performance von Mechanismen zur automatischen Speicherbereinigung gibt es vier wichtige Maße:

- Durchsatz
- Pausen
- Speicherverbrauch
- Promptheit

Durchsatz bezeichnet den prozentualen Anteil der Laufzeit eines Programms, der nicht mit Speicherbereinigung oder Allokation verbracht wird. Dies ist eine wichtige Größe bei lang laufenden Programmen wie Servern.

Pausen sind Zeiten, in denen die Applikation nicht reagiert, da gerade der Speicher aufgeräumt wird. Insbesondere bei interaktiven oder Echtzeit-Programmen ist hier der Maximalwert eine interessante Größe zur Beurteilung der Speicherverwaltung.

Speicherverbrauch ist für Systeme mit keinem oder begrenztem virtuellen Speicher eine wichtige Größe.

Promptheit bezeichnet die Zeit, die nach dem Tod eines Objekts vergeht, bis es tatsächlich beseitigt ist.

Gewöhnlich gilt, dass ein guter Wert in einer Kategorie zu Lasten des Wertes einer anderen Kategorie geht. Es kann also keinen per se richtigen, sondern nur einen am besten zu den Anforderungen passenden Garbage Collector geben.

3.2.4 HotSpots Garbage Collection

Suns VM ist ein gutes Beispiel für einen Generationen-Kollektor. Der Heap ist in fünf Generationen unterteilt (Abbildung 3.3).

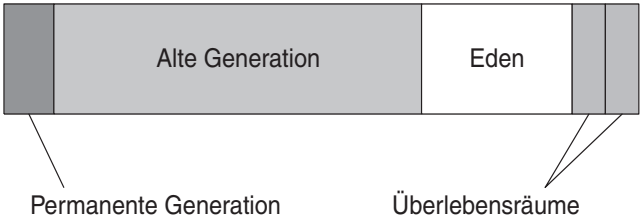

Abbildung 3.3: Aufteilung des Heaps der Sun HotSpot-VM in verschiedene Generationen

Objekte werden zunächst in *Eden* alloziert. Ist kein Platz mehr in Eden, greift ein kopierender Kollektor und verschiebt alle lebendigen Objekte aus Eden sowie einem der *Überlebensräume* in den anderen, leeren Überlebensraum. Wenn Objekte eine gewisse Zeit zwischen den beiden Überlebensräumen hin- und herkopiert wurden, werden sie befördert. Das heißt, sie werden in die *alte Generation* verschoben. In der Alten Generation wiederum greift ein Mark-Compact-Kollektor. Eine Besonderheit stellt die *permanente Generation* dar. Sie enthält Klassen- und Methoden-Objekte, die sehr selten – wenn überhaupt – dealloziert werden müssen.

Alternativ zum standardmäßig verwendeten Mark-Compact-Algorithmus für die alte Generation kann man bei Bedarf mittels des VM-Parameters `-Xincgc` auch einen inkrementellen Algorithmus verwenden. Dieser verursacht zwar weniger Pausen, hat jedoch einen größeren Verwaltungsaufwand.

Die Größe des Heaps sowie der einzelnen Generationen lässt sich über VM-Parameter beeinflussen (Tabelle 3.2). Um festzustellen, welche Einstellung für Ihr Programm optimal ist, experimentieren Sie mit verschiedenen Werten und vergleichen Sie die Garbage Collection-Daten, die Sie durch den VM-Parameter `-verbose:gc` erhalten.

Der Durchsatz ist gewöhnlich am besten, wenn möglichst selten eine vollständige Speicherbereinigung inklusive der Alten Generation erforderlich ist. Dies ist der Fall, wenn Eden und die beiden Überlebensräume (*junge Generationen*) im Verhältnis zum Rest des Heaps sehr groß sind. Dies geht jedoch auf Kosten von Speicherverbrauch und Promptheit. Pausen wiederum können minimiert werden, indem die Jungen Generationen möglichst klein gehalten und inkrementelle Garbage Collection für die alte Generation benutzt wird.

Parameter	**Beschreibung**
-Xms<wert>	Minimale Heapgröße
-Xmx<wert>	Maximale Heapgröße
-Xminf<wert>	Prozentualer Anteil des Heaps, der nach einer vollständigen Speicherbereinigung mindestens frei sein sollte. Standard: 40
-Xmaxf<wert>	Prozentualer Anteil des Heaps, der nach einer vollständigen Speicherbereinigung höchstens frei sein sollte. Standard: 70
-XX:NewRatio=<wert>	Verhältnis der Größe der Neuen Generationen zur Alten Generation
-XX:NewSize=<wert>	Startgröße der Neuen Generationen
-XX:MaxNewSize=<wert>	Bestimmt die maximale Größe der Neuen Generationen
-XX:SurvivorRatio=<wert>	Verhältnis der Größe von Eden zu einem der Überlebensräume
-XX:MaxPermSize=<wert>	Maximale Größe der Permanenten Generation

Tabelle 3.2: Parameter zum Optimieren der HotSpot-Speicherverwaltung

3.3 Industrie-Benchmarks

Zur Beurteilung von Java VMs lohnt es sich, Leistungstests oder Benchmarks zu benutzen. Ein Benchmark ist eine definierte Referenz, die zu Vergleichszwecken herangezogen werden kann. Hiermit ist meist ein definiertes Testverfahren gemeint, das reproduzierbar die Leistung von Soft- oder Hardware misst. Oft wird das Ergebnis in einer einzelnen Maßzahl zusammengefasst.

Diese Labor-Benchmarks beschreiben per Definition nur eine Abbildung der Wirklichkeit. Aus diesem Grund definiert Eric Raymond einen Benchmark als »an inaccurate measure of computer performance« und zitiert in seinem Buch die alte Hacker-Weisheit:

> *In the computer industry, there are three kinds of lies: lies, damn lies, and benchmarks. [Raymond96].*

Nun gibt es glücklicherweise Benchmarks, die von Organisationen wie *SPEC* (Standard Performance Evaluation Corporation, *http://www.spec.org/*), einzelnen unparteiischen Firmen und Verlagen bzw. Magazinen kreiert wurden und somit sicherlich vertrauenswürdiger sind als beispielsweise ein reiner Intel-Benchmark zum Vergleich von Intel- und Motorola-Prozessoren.

Dennoch muss man sich bei jedem Benchmark fragen, was dieser misst, welche Aussage sich aus dem Ergebnis ableiten lässt und wie nützlich diese Aussage in Bezug auf das eigene Programm ist. Es nützt Ihnen wenig, eine VM zu verwenden, die hervorragend bzgl. gleichzeitiger TCP/IP-Verbindungen skaliert, wenn Sie tatsächlich numerische Berechnungen anstellen wollen und nie auch nur eine einzige TCP/IP-Verbindung aufbauen.

In diesem Abschnitt werden einige sehr verschiedene Benchmarks kurz vorgestellt. Hier sind die entsprechenden URLs:

- VolanoMark: *http://www.volano.com/benchmarks.html*
- SPEC JVM98: *http://www.spec.org/osg/jvm98/*
- SPEC JBB2000: *http://www.spec.org/osg/jbb2000/*
- jBYTEMark 0.9 (BYTE Magazine) scheint nicht mehr durch einen regulären Link erreichbar zu sein. Sie können jedoch mit einer Suchmaschine nach *jbyte.zip* suchen.
- ECperf: *http://java.sun.com/j2ee/ecperf/index.html*

3.3.1 VolanoMark

VolanoMark ist ein Benchmark, der entstand, um die Performance einer VM und ihrer Skalierbarkeit in Bezug auf TCP/IP-Verbindungen zu messen. Die Motivation für den letzteren Aspekt liegt in Javas Ein-Thread-pro-Verbindung-Modell begründet (siehe

Kapitel 10.5 Skalierbare Server). Dieses besagt, dass für jede Verbindung ein dedizierter Thread existieren muss. Wenn man also sehr viele Verbindungen gleichzeitig unterhalten will (und genau das macht der Volano-Chat-Server), ist es wichtig, eine VM zu benutzen, die sowohl viele Sockets als auch viele Threads performant unterstützt.

Während des Tests wird gemessen, wie viele Nachrichten Clients über einen Server verschicken können. Beim Performance-Test laufen sowohl Server als auch Clients auf demselben Rechner mit 200 gleichzeitigen Loopback-Verbindungen. Beim Skalierbarkeitstest simulieren die Clients von einem anderen Rechner aus eine ständig steigende Zahl gleichzeitiger Verbindungen. Gemessen wird die maximal mögliche Anzahl simultaner Verbindungen.

Seit JDK 1.4 bietet Java mit dem `java.nio`-Paket ein alternatives, asynchrones Ein-/Ausgabe-Modell, das nicht mehr zwingend einen Thread pro Verbindung vorschreibt. VolanoMark in seiner jetzigen Form könnte also entsprechend an Bedeutung verlieren.

Nichtsdestotrotz ist VolanoMark ein gerne zitierter Benchmark für Performance und Netzwerk-Skalierbarkeit. Ergebnisse werden regelmäßig auf der Volano-Website publiziert.

3.3.2 SPEC JVM98

SPEC JVM98 ist ein Client-Benchmark der SPEC zum Messen von Java-VM-Performance. Er besteht aus acht verschiedenen Tests, von denen fünf reale Anwendungen sind. Sieben der Tests dienen zum Messen von Daten, der achte verifiziert die korrekte Ausführung des Bytecodes:

- compress: Werkzeug zum (De-)Komprimieren von Dateien
- jess: Java Expertensystem
- db: ein kleines Datenmanagement-Programm
- javac: Suns Java-Compiler
- mpegaudio: ein MPEG-3-Dekoder
- mtrt: ein multithreaded Raytracer
- jack: ein Parser-Generator mit lexikalischer Analyse
- check: Überprüft Java VM und Java Features

JMV98 testet nicht AWT-, Netzwerk-, Datenbank- und Grafik-Performance. Zudem erschien JVM98 vor Java 2. Es werden also auch keine der neueren Java-Features wie schwache Referenzen oder dynamische Proxies getestet.

3.3.3 SPEC JBB2000

SPEC JBB2000 ist ein serverseitiger SPEC-Benchmark, der eine 3-Tier-Applikation simuliert. Die Hauptlast liegt dabei auf dem Mittel-Tier, der die Geschäftslogik enthält. Tier eins und drei enthalten die Benutzerschnittstelle und die Datenverwaltung. Der Test läuft vollständig in einer VM ab und benötigt keine weiteren Komponenten wie eine Datenbank oder einen Webbrowser. Im Test werden weder Enterprise Java Beans (EJB), Servlets noch JSP verwendet.

JBB2000 misst ausschließlich VM-Performance und Skalierbarkeit. Nicht gemessen werden AWT-, Netzwerk-, Ein-/Ausgabe- und Grafik-Leistung.

3.3.4 jBYTEMark

jBYTEMark ist ein ursprünglich in C geschriebener Benchmark, der vom BYTE-Magazin nach Java portiert wurde. Er enthält ausschließlich rechenintensive Algorithmen, versucht also gar nicht erst eine reale Applikation nachzuempfinden. Dem Benchmark-Code lässt sich zudem leicht ansehen, dass er aus der C-Welt stammt. OO-Performance lässt sich mit diesem Benchmark nicht messen.

jBYTEMark scheint von BYTE schon lange aufgegeben worden zu sein, ist aber immer noch für einen schnellen Numbercrunching-Vergleich zu gebrauchen. Wichtig ist zu betonen, dass dieser Benchmark keinen Gebrauch von Threads macht und weder AWT-, Netzwerk-, Ein-/Ausgabe- noch Grafik-Leistung misst.

3.3.5 ECperf

Anders als die zuvor aufgeführten Benchmarks dient *ECperf* zum Messen der Skalierbarkeit und Performance von J2EE-Servern (Java 2 Enterprise Edition) und nicht einer speziellen Java VM. Dabei werden hauptsächlich Speicherverwaltung, Verbindungs-Pooling, Passivierung/Aktivierung von EJBs und Caching des EJB-Containers getestet. Die Leistung der gewöhnlich nötigen Datenbank fließt angeblich kaum in den Benchmark ein.

ECperf simuliert eine reale Anwendung, die die Bereiche Herstellung, Supply-Chain-Management und Verkauf abbildet. Dies führt zur Nutzung und somit indirekt zur Bewertung der folgenden technischen Aspekte von J2EE-Anwendungen:

- Transaktionale Komponenten
- Verteilte Transaktionen
- Messaging und asynchrones Aufgabenmanagement
- Mehrere Service-Provider mit mehreren Websites
- Schnittstellen von und zu Altsystemen

- Sichere Datenübertragung
- Rollenbasierte Authentisierung
- Persistente Objekte

ECPerf wurde von SPEC übernommen und firmiert dort unter dem Namen *SPECjAppServer200X*.

3.4 Die richtige VM auswählen

Die VM zu wechseln ist eine der einfachsten und billigsten Methoden, die Performance Ihrer Software zu verbessern. Wenn Sie nicht gerade auf die VM eines bestimmten Herstellers angewiesen sind, sollte dies Ihr erster Schritt sein. Sie können eine grobe Vorauswahl anhand von publizierten Benchmarks vornehmen. Achten Sie dabei darauf, dass die Benchmarks Leistungsaspekte messen, die für Ihre Anwendung wichtig sind. Anders ausgedrückt: Die Endgeschwindigkeit eines Autos ist in den USA irrelevant. Wichtig ist die Beschleunigung bis zur erlaubten Höchstgeschwindigkeit.[5]

Viele publizierte Benchmarks messen die Leistung eines Ein-Prozessor-Systems. Wenn Sie wollen, dass Ihre VM mit der Prozessorzahl Ihres Mehrprozessor-Systems skaliert, verifizieren Sie, dass die VM dazu geeignet ist. VMs, die so genannte Green-Threads benutzen, skalieren nicht mit der Prozessorzahl, sondern nutzen immer nur einen Prozessor. Ist jedoch nur ein Prozessor vorhanden, brillieren Green-Threads, da sie in der Regel leichtgewichtiger sind als Native-Threads. Ein Beispiel hierfür ist das Linux Blackdown JDK.

Falls Ihr Programm Echtzeit-Kriterien[6] standhalten muss, wählen Sie eine VM mit inkrementellem oder nebenläufigem Garbage Collector. Inkrementelle Garbage Collection hält die maximalen Pausenzeiten kurz, führt jedoch sonst meist zu schlechterer Performance.

Wenn Ihre Anwendung keine Massenware ist, sondern nur in einer definierten Umgebung lauffähig sein muss, ziehen Sie AOT-Compiler in Betracht. »Write once, run anywhere« spielt für Sie keine Rolle, insbesondere nicht, wenn Sie über den Quellcode verfügen und jederzeit regulären Bytecode einsetzen können.

Nachdem Sie sich für eine VM entschiedenen haben, informieren Sie sich über Optimierungsmöglichkeiten. Fast alle VMs haben Parameter, mit denen Sie entscheidenden Einfluss auf die Leistung der VM nehmen können. Dazu gehören die minimale

5 In den meisten Staaten sind das 70 mph (etwa 113 km/h).

6 Kaum eine Java VM hält harten Echtzeitkriterien stand. Jedoch gibt es bzgl. des Echtzeitverhaltens natürlich bessere und schlechtere VMs.

und maximale Heapgröße, die Stackgröße, GC-Algorithmen und GC-Generationen-Größen, Thread-Modelle, JIT/DA-Compiler und vieles mehr. Insbesondere, wenn Sie wenig Einfluss auf den Quellcode haben, ist dies Ihr bester Ansatzpunkt.

Die Wahl der richtigen VM kann leicht über Erfolg und Misserfolg Ihres Projekts entscheiden. Es ist daher blauäugig, einfach die erstbeste VM zu benutzen. Zudem konkurrieren alle VM-Hersteller darum, die schnellste VM herzustellen. Es lohnt sich also, immer mal wieder eine neue oder andere VM auszuprobieren. Dank Javas Portabilität sollte dies keine allzu große Schwierigkeit darstellen.

4 Messwerkzeuge

Es gibt vielerlei Möglichkeiten Performance zu messen. Auf Windows-Systemen können Sie beispielsweise einfach den Windows-Task-Manager (Abbildung 4.1) während der Programmausführung beobachten. Das gibt Ihnen immerhin schon einmal groben Aufschluss über Prozessor-Belastung und Speicherverbrauch. Genauere Daten erhalten Sie über den Windows-Systemmonitor (Management Konsole), den Sie im Verwaltungsordner der Systemsteuerung finden. Zudem gibt es diverse freie Werkzeuge bzw. Shareware-Programme (z.B. TaskInfo, *http://www.iarsn.com/*), die detaillierten Aufschluss über den System-Zustand geben.

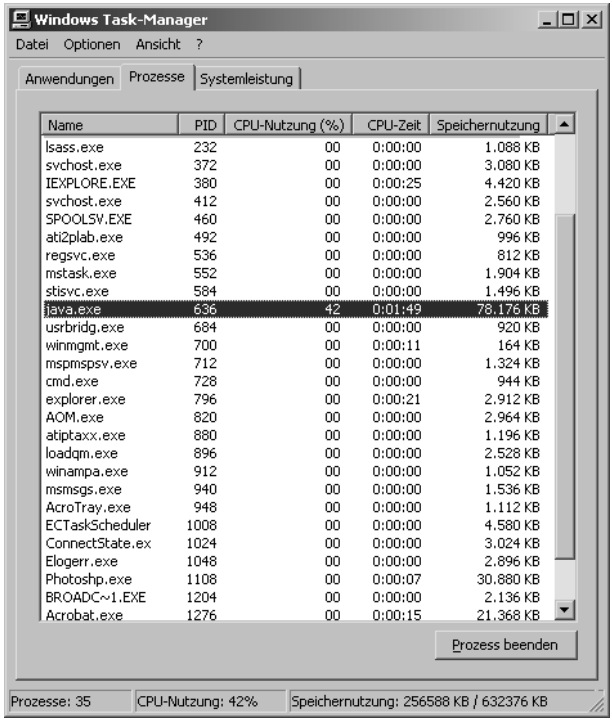

Abbildung 4.1: Prozessansicht des Windows-Taskmanagers. Der markierte Java-Prozess belegt die CPU zu 42 Prozent und rund 78 Mbyte Speicher.

Unter Unix bzw. Linux stellen Werkzeuge wie *top*, *ps*, *netstat*, *vmstat*, *iostat*, *sar* (System Accounting Reports), *truss* und *strace* nützliche Daten über Prozesse, Kernelzugriffe, Netzwerkbelastung und Speicherverbrauch zur Verfügung. Genaueres über ihre Benutzung lässt sich den entsprechenden Man-Pages entnehmen.

4.1 Profiler

Nun können die genannten Systemwerkzeuge nicht in Ihr Java-Programm reingucken. Sie können Ihnen nicht sagen, wie viele Objekte vom Typ *X* gerade existieren und wie viel Speicher sie verbrauchen, wie viel Zeit Ihr Programm in Methode *Y* verbringt und wie häufig diese Methode bereits aufgerufen wurde. Genau dafür gibt es so genannte Profiler. Diese Programme erleichtern die Laufzeit-Analyse Ihrer Programme und helfen Flaschenhälse und Speicherlöcher zu finden.

Hier eine kleine Auswahl kommerzieller Profiler:

- JProbe von Sitraka: *http://www.jprobe.com/*
- Optimizeit von VMGear: *http://www.optimizeit.com/*
- Quantify von Rational: *http://www.rational.com/*
- DevPartner von Compuware: *http://www.compuware.com/*

4.2 Hprof

Glücklicherweise müssen Sie jedoch nicht viel Geld für einen kommerziellen Profiler ausgeben, um einen Einblick in Ihre Programme zu erhalten. Sowohl die Sun als auch die IBM Java VM enthält bereits einen einfachen Profiler namens *Hprof*.

Hprof bedient sich des *Java Virtual Machine Profiler Interfaces* (*JVMPI*), einer bislang experimentellen, nicht-standardisierten Schnittstelle (Stand Anfang 2002). Die zugehörige, detaillierte Dokumentation der Schnittstelle befindet sich in der Dokumentation des Sun JDKs im Ordner *docs/guide/jvmpi/index.html* oder online unter *http://java.sun.com/j2se/1.4/docs/guide/jvmpi/index.html*. C-Kenntnisse und einen entsprechenden Compiler vorausgesetzt, können Sie mit Hilfe dieser Dokumentation Ihren eigenen Profiler schreiben. Nur mit Java ist dies leider nicht möglich.

Bevor Sie sich jedoch anschicken, das Rad neu zu erfinden, sollten Sie sich zunächst ein bisschen mit dem vorhandenen Hprof beschäftigen. Wenngleich dieser Minimal-Profiler nicht unbedingt dem Vergleich mit kommerziellen Produkten standhält, so ist er doch durchaus nützlich.

Hprof wird durch den `-Xrunhprof` Kommandozeilenparameter der Sun Java-VM gestartet:

```
java -Xrunhprof[:help][:<parameter>=<wert>, …] MainClass
```

Eine Übersicht über die möglichen Parameter finden Sie in Tabelle 4.1. Eine englische Kurzfassung dessen erhalten Sie auch, wenn Sie die `help`-Option angeben.

Option	Beschreibung	Standardwert
heap=dump|sites|all	Gibt den Inhalt des gesamten Heaps aus (dump), generiert Stacktraces, die anzeigen, wo Speicher allokiert wurde (sites) oder beides (all)	all
cpu=samples|times|old	Zum Messen der Rechenzeit in Methoden werden Stichproben (samples), Laufzeiten und Ausführungshäufigkeit einzelner Methoden (times) oder das in früheren VMs verwendete Format (old) benutzt	off
monitor=y|n	Gibt Informationen über Monitore für die Thread-Synchronisation aus	n
format=a|b	Textuelle (a) oder binäre (b) Ausgabe	a
file=<file>	Schreibt die Daten in die angegebene Datei	java.hprof(.txt bei textueller Ausgabe)
net=<host>:<port>	Schreibt die Daten über die angegebene TCP-Verbindung	off
depth=<wert>	Tiefe der auszugebenden Stacktraces	4
cutoff=<wert>	Prozentwert für Ranglisten, nach dem diese abgeschnitten wird	0.0001
lineno=y|n	Angabe von Zeilennummern in Stacktraces	y
thread=y|n	Angabe des Threads im Stacktrace	n
doe=y|n	Ausgabe bei Beendigung der VM	y
dooom=y|n	Ausgabe bei OutOfMemoryError (nur IBM VM)	y

Tabelle 4.1: Kommandozeilen-Parameter des in der Sun und IBM VM integrierten Profilers Hprof

4.2.1 Speicherabbild erstellen

Ruft man Hprof mit seinen Standardparametern auf, erhält man nach Beendigung der VM eine Übersicht über alle im Speicher befindlichen Objekte (Heap-Dump), eine nach Speicherverbrauch sortierte Liste dieser Objekte (Sites) sowie Stacktraces, die Auskunft darüber geben, an welchen Stellen im Code Objekte alloziert wurden.

Außer nach Beendigung der VM erhalten Sie diese Ausgabe auch, wenn Sie Strg \ in der Java-Konsole von Unix/Linux-Systemen bzw. Strg Pause bei Windows-Systemen drücken. Alternativ können Sie auf Unix/Linux-Systemen auch ein SIGQUIT an den Java-Prozess senden: `kill -QUIT <prozessid>`

```
01 package com.tagtraum.perf.memory;
02
03 public class HprofHeapDemo {
04
05    private byte[] byteArray;
06    private String string;
07    private HprofHeapDemo internalDemoInstance;
08
09    public void setByteArray() {
10       byteArray = new byte[1024];
11    }
12
13    public void setString() {
14       string = "1234567890"; // String der Länge 10
15    }
16
17    public void setHprofHeapDemo() {
18       internalDemoInstance = new HprofHeapDemo();
19    }
20
21    public static void main(String[] args) {
22       HprofHeapDemo demoInstance = new HprofHeapDemo();
23       demoInstance.setByteArray();
24       demoInstance.setString();
25       demoInstance.setHprofHeapDemo();
26    }
27 }
```

Listing 4.1: Beispiel-Programm, das einige Objekte alloziert

Wir wollen uns die Ausgabe von Hprof ein wenig genauer anschauen und führen das Beispielprogramm *HprofHeapDemo* folgendermaßen aus:

```
java -Xrunhprof com.tagtraum.perf.memory.HprofHeapDemo
```

Nach Beendigung des Programms befindet sich im aktuellen Verzeichnis die Datei *java.hprof.txt*. Diese Datei wird beim nächsten Lauf ohne Warnung überschrieben. Kopieren Sie deshalb Ergebnisse, die Sie für spätere Vergleiche behalten wollen, oder verwenden Sie den `file`-Parameter um eine andere Datei zu benutzen (Tabelle 4.1).

Wir erwarten, in der Datei die in *HprofHeapDemo* allozierten Objekte samt ihres Speicherverbrauchs wiederzufinden. Am Anfang der Datei finden wir zunächst einige Hinweise darauf, dass das ausgegebene Format experimentell ist, und einen Text, der grob erläutert, wie die Daten zu interpretieren sind. Danach beginnt der eigentliche Datenteil mit einer Liste aller Threads:

```
THREAD START (obj=838648, id = 1, name="Finalizer", group="system")
THREAD START (obj=838688, id = 2, name="Reference Handler", group="system")
THREAD START (obj=838708, id = 3, name="main", group="main")
```

```
THREAD START (obj=89d9588, id = 4, name="Signal Dispatcher", group="system")
THREAD START (obj=89da3c8, id = 5, name="CompileThread0", group="system")
THREAD END (id = 3)
THREAD START (obj=89d81c8, id = 6, name="Thread-0", group="main")
THREAD END (id = 4)
```

Der Threadliste folgt eine Liste von nummerierten Stacktraces:

```
...
TRACE 594:
...java.lang.ClassLoader.defineClass0(ClassLoader.java:Native method)
...java.lang.ClassLoader.defineClass(ClassLoader.java:486)
...java.security.SecureClassLoader.defineClass(SecureClassLoader.java:111)
...java.net.URLClassLoader.defineClass(URLClassLoader.java:248)
TRACE 596:
   HprofHeapDemo.main(HprofHeapDemo.java:22)
TRACE 597:
   HprofHeapDemo.setByteArray(HprofHeapDemo.java:10)
   HprofHeapDemo.main(HprofHeapDemo.java:23)
TRACE 598:
   HprofHeapDemo.setString(HprofHeapDemo.java:14)
   HprofHeapDemo.main(HprofHeapDemo.java:24)
TRACE 599:
   HprofHeapDemo.setHprofHeapDemo(HprofHeapDemo.java:18)
   HprofHeapDemo.main(HprofHeapDemo.java:25)
...
```

Die Stacktraces geben an, wo im Code etwas passierte, und werden anhand ihrer Nummern in den beiden folgenden Teilen referenziert. So findet man im Heap-Dump bei jedem Objekt einen Verweis auf einen Stacktrace. Dieser gibt den genauen Allokations-Ort an.

```
HEAP DUMP BEGIN (5485 objects, 931896 bytes)
...
OBJ 8aaf248 (sz=24, trace=596, class=HprofHeapDemo@8aaee08)
    internalDemoInstance    8aaf540
    byteArray    8aaf318
    string       8aaf458
ARR 8aaf318 (sz=1040, trace=597, nelems=1024, elem type=byte)
ARR 8aaf400 (sz=32, trace=598, nelems=10, elem type=char)
OBJ 8aaf458 (sz=24, trace=598, class=java.lang.String@838ea8)
    value        8aaf400
OBJ 8aaf540 (sz=24, trace=599, class=HprofHeapDemo@8aaee08)
CLS 8aaee08 (name=HprofHeapDemo, trace=594)
    super        838e48
    loader       8b7b68
    domain       8aabca8
...
HEAP DUMP END
```

So entnehmen wir dem Heap-Dump, dass Objekt `8aaf248` vom Typ `HprofHeapDemo` in Stacktrace 596 allokiert wird. Trace 596 wiederum verweist auf Zeile 22 der `main()`-Methode unserer Testklasse (Listing 4.1). Et voilà: In Zeile 22 wird ein `HprofHeapDemo`-Objekt instanziiert.

Weiter entnehmen wir dem Dump, dass Objekt `8aaf248` genau 24 Byte belegt und drei weitere Objekte mit den Namen `internalDemoInstance`, `byteArray` und `string` besitzt. Anhand ihrer Ids können wir auch diese Objekte näher unter die Lupe nehmen.

Durch ein vorangestelltes `ARR` ist das Objekt `8aaf318` im Dump als Array gekennzeichnet. Anhand der Id erkennen wir, dass es sich um den als `byteArray` bezeichneten `byte`-Array handelt. Zusätzlich zum Speicherverbrauch `sz` (1040 Byte für einen 1024 Byte großen Array) und Tracenummer erhalten wir Informationen darüber, wie viele Elemente dieser Array enthalten und welcher Elementtyp in ihm gespeichert werden kann.

Weiter stellen wir fest, dass in Trace 598, also der `setString()`-Methode, zwei Objekte instanziiert werden: ein `char`-Array mit zehn Elementen sowie ein String-Objekt, welches den `char`-Array als `value`-Objekt besitzt.

Gleich darunter finden wir das `internalDemoInstance`-Objekt, das jedoch im Gegensatz zu dem vorher gefundenen `HprofHeapDemo`-Objekt keinerlei andere Objekte besitzt. Das ist auch nicht weiter verwunderlich, da keiner seiner Instanzvariablen während der Ausführung ein Wert zugewiesen worden ist. Und schließlich ist da noch das durch ein vorangestelltes `CLS` markiertes Klassenobjekt für unsere Testklasse.

Neben `OBJ`, `ARR` und `CLS` gibt es noch den Typ `ROOT`, jedoch keinen Typ für die primitiven Datentypen wie `int` oder `boolean`. Diese sind implizit im Objekt enthalten und sofern es sich um Instanzvariablen handelt, machen sie sich im Größen-Attribut `sz` bemerkbar.

Das alles scheint schön übersichtlich zu sein – ist es aber leider nicht. Der abgedruckte Heap-Dump nämlich ist nur ein sehr kleiner Auszug aus dem tatsächlich erzeugten, über 10.000 Zeilen langen Dump. Ganz schön viel für solch ein kleines Programm.

Damit Sie nicht völlig im Datenwust versinken, folgt dem Dump noch ein wesentlich kürzerer Abschnitt namens Sites:

```
SITES BEGIN (ordered by live bytes)
          percent          live       alloc'ed  stack class
 rank   self  accum    bytes objs   bytes objs trace name
...
    1 64.18% 64.18%   598112  157  598112  157     1 [I
    2  9.23% 73.41%    86008  798   86008  798     1 [C
...
   10  0.88% 90.23%     8208    1    8208    1   234 [B
   11  0.27% 90.50%     2496  104    2496  104   236 java.lang.String
   12  0.26% 90.76%     2464   35    2464   35   129 [C
   13  0.24% 91.00%     2248    1    2248    1    85 [B
```

```
   14  0.18% 91.18%     1680   35    1680   35   131 java.util.HashMap
...
   19  0.11% 91.90%     1040    1    1040    1   597 [B
...
SITES END
```

Die Sites-Sektion listet die Objekttypen auf, die am meisten Speicher belegen. Die Spalte `self` gibt dabei an, wie viel Prozent des Heaps durch Objekte des in der Spalte `classname` angegebenen Typs belegt sind (Tabelle 4.2) und im angegebenen Trace alloziert wurden. Die Spalte `accum` enthält die Summe aller Einträge aus der `self`-Spalte bis zum aktuellen Rang.

Symbol	Bedeutung
[Z	`boolean`-Array
[B	`byte`-Array
[S	`short`-Array
[C	`char`-Array
[I	`int`-Array
[J	`long`-Array
[F	`float`-Array
[D	`double`-Array
[L<Unknown>	zweidimensionaler Array
<Klassenname>	Klasse oder Array einer Klasse (z.B. `java.lang.String`)

Tabelle 4.2: Typen, wie sie in der Sites-Sektion angegeben werden

Die Spalten `live bytes` und `live objs` geben an, wie viel Speicher zum Zeitpunkt des Dumps durch wie viele Objekte belegt wurde. Demgegenüber geben die Spalten `alloc'ed bytes` und `alloc'ed objs` an, wie viele Byte *jemals* durch wie viele Objekte des angegebenen Typs belegt wurden. Das beinhaltet also auch all jene Objekte, die bereits von der automatischen Speicherbereinigung eingesammelt wurden. In unserem Beispiel sind die Werte jeweils gleich, da die Speicherbereinigung keines der instanziierten Objekte beseitigen konnte.

Als einziges Objekt, das wir selbst erzeugt haben, finden wir unseren `byte`-Array auf Rang 19 – erkennbar an der Trace-Nummer 597. Alle anderen Objekte waren zu klein, um für die Auflistung im Sites-Abschnitt relevant zu sein, da die Liste automatisch bei einem `self`-Wert kleiner als 0.0001% abgeschnitten wird. Sie können diesen Wert über den Parameter `cutoff` (Tabelle 4.1) manipulieren.

Beachten Sie, dass in der Sites-Übersicht kein Unterschied zwischen Objekten und Objekt-Arrays gemacht wird. So wird für einen Array von Strings genauso wie für ein einzelnes String-Objekt der Typ `java.lang.String` aufgelistet. Der einzige Weg den

Unterschied sicher zu erkennen, ist es, der Trace-Referenz zu folgen und im Heap-Dump nachzuschauen, ob vor der Objekt-Id ein `OBJ` oder ein `ARR` steht. Primitive Datentypen werden übrigens genau wie im Heap-Dump gar nicht erst aufgelistet.

Zudem gibt es noch eine weitere Merkwürdigkeit. Die Summen der Spalten `live bytes` und `live objects` stimmen beim besten Willen nicht mit den Zahlen in der Heap-Dump-Begin-Zeile überein.

Zusammenfassend lässt sich sagen, dass den Heap zu inspizieren ein verlässliches Verfahren zum Aufspüren und zum Beseitigen von Speicherlöchern (oder besser unbeabsichtigt referenzierten Objekten) ist. Leider ist die von Hprof generierte Textdatei nicht besonders übersichtlich und wird häufig sehr groß. Um sich ein bisschen besser zurechtzufinden, können Sie beispielsweise die freien Werkzeuge *HPjmeter* von HP oder *HAT* (Heap-Analysis-Tool) benutzen. HAT ist ein von Sun nicht unterstütztes, experimentelles Werkzeug, funktioniert nur mit der Binär-Ausgabe von Hprof und ist etwas umständlich zu bedienen. HPjmeter hingegen ist ein kleines, recht komfortables Swing-Programm, das auch für andere Zwecke als nur die Heap-Analyse brauchbar ist. Ebenfalls gratis ist der *Win32 HeapInspector* von Paul Moeller. HeapInspector benutzt allerdings nicht den Hprof-Profiler, sondern klinkt sich per JVMPI direkt in die VM ein.

Zu finden sind die drei Werkzeuge unter folgenden URLs:

- HPjmeter: *http://www.hpjmeter.com/*
- HAT: *http://java.sun.com/people/billf/heap/index.html*
- Win32 HeapInspector: *http://www.geocities.com/moellep/debug/HeapInspector.html*

4.2.2 CPU-Profiling

Hprof kann nicht nur zum Inspizieren des Heaps, sondern auch zum Analysieren des Laufzeitverhaltens Ihres Programms benutzt werden. Dazu müssen Sie in der Kommandozeile den Parameter `cpu` angeben. Sie haben die Wahl zwischen drei verschiedenen Profiling-Modi: `samples`, `times` und `old`.

Wir wollen uns die Ausgaben aller drei Optionen für die Testklasse `HprofCPUDemo` ansehen. Nach dem Start der Klasse werden 100.000 mal drei verschiedene Methoden aufgerufen, die jeweils leere Schleifen unterschiedlicher Länge enthalten. Wir erwarten, dass der Laufzeitanteil der drei Methoden proportional zu ihrer Schleifenlänge ist. An dieser Stelle sei erwähnt, dass weder jikes 1.15, Sun javac 1.3.1, Sun javac 1.4 noch IBM javac 1.3.0 die leeren Schleifen wegoptimieren. Überhaupt optimieren Java-Compiler so gut wie gar nicht, weshalb ihnen in diesem Buch auch kein Kapitel gewidmet ist. Während der Ausführung erkennt zumindest IBMs JIT, dass es sich um leere Schleifen handelt, und entfernt sie. Die Sun VMs scheinen dies nur mit der HotSpot-Server-Version zu erkennen. Die folgenden Beispiel-Ergebnisse wurden daher mit Suns HotSpot-Client-Version produziert.

```
01 package com.tagtraum.perf.cpu;
02
03 public class HprofCPUDemo {
04
05    public void slowMethod() {
06       for (int i=0; i<20000; i++);
07    }
08
09    public void mediumMethod() {
10       for (int i=0; i<10000; i++);
11    }
12
13    public void fastMethod() {
14       for (int i=0; i<5000; i++);
15    }
16
17    public static void main(String[] args) {
18       HprofCPUDemo demoInstance = new HprofCPUDemo();
19       for (int i=0; i<100000; i++) {
20          demoInstance.slowMethod();
21          demoInstance.mediumMethod();
22          demoInstance.fastMethod();
23       }
24    }
25 }
```

Listing 4.2: Einfache Testklasse zur Illustration der verschiedenen CPU-Profiling-Modi

Stichproben

Das `samples`-Format unterscheidet sich von den beiden anderen Formaten insofern, als es stichprobenbasiert ist. Das heißt, der Profiler schaut alle *x* Millisekunden nach, welche Methode die VM gerade ausführt. Da sich das Stichprobenintervall nicht ändern lässt, bedeutet dies auch, dass dieser Modus ungeeignet ist, wenn Ihr Programm nur eine sehr kurze Laufzeit hat.

Um einen Einblick in das Format zu gewinnen, starten wir unserer Testklasse folgendermaßen:

```
java -Xrunhprof:cpu=samples com.tagtraum.perf.cpu.HprofCPUDemo
```

Die resultierende Ausgabe enthält wiederum verschiedene Abschnitte. Neu ist die Cpu-Samples-Sektion.

```
...
TRACE 8:
    HprofCPUDemo.fastMethod(HprofCPUDemo.java:14)
    HprofCPUDemo.main(HprofCPUDemo.java:22)
TRACE 7:
```

```
    HprofCPUDemo.slowMethod(HprofCPUDemo.java:6)
    HprofCPUDemo.main(HprofCPUDemo.java:20)
TRACE 6:
    HprofCPUDemo.mediumMethod(HprofCPUDemo.java:10)
    HprofCPUDemo.main(HprofCPUDemo.java:21)
...
CPU SAMPLES BEGIN (total = 174)
rank   self  accum   count trace method
   1 54.02% 54.02%      94     7 HprofCPUDemo.slowMethod
   2 30.46% 84.48%      53     6 HprofCPUDemo.mediumMethod
   3 13.22% 97.70%      23     8 HprofCPUDemo.fastMethod
...
   6  0.57% 99.43%       1     3 java.io.InputStreamReader.<init>
   7  0.57% 100.00%      1     1 sun.misc.URLClassPath$2.run
CPU SAMPLES END
```

Der Ausgabe unseres Beispiels entnehmen wir, dass insgesamt 174 Stichproben genommen wurden. Die Ergebnisse dieser Proben sind in einer Art Hitliste dargestellt. Auf Rang eins finden wir die Methode `slowMethod()`. Zudem können wir der Spalte `count` entnehmen, dass die VM während 94 der 174 Stichproben gerade diese Methode ausführte. Das entspricht 54,02% aller genommenen Stichproben, einem Wert, der in der Spalte `self` aufgelistet ist. In der `accum`-Spalte befindet sich die Summe aller gleich- oder höherrangigen `self`-Werte. `trace` wiederum verweist auf den entsprechenden Stacktrace und somit auf eine Zeile. Auf Rang zwei und drei der Liste finden wir gleichartige Einträge für die Methoden `mediumMethod()` und `fastMethod()`.

Qualitativ stimmt unsere Messung also mit unseren Erwartungen überein. Quantitativ ist das Ergebnis jedoch nicht so gut. Gemäß Quellcode müsste `slowMethod()` für jede Ausführung doppelt so lange benötigen wie `mediumMethod()` und `fastMethod()` halb so lange wie `mediumMethod()`. Wenn wir uns jedoch die `count`-Werte angucken, stellen wir fest, dass diese nicht allzu genau zu unseren Erwartungen passen. Ausgehend von `fastMethod()`s 23 Counts erhalten wir für `mediumMethod()` 53 statt erwarteter 46 Counts. Das ist eine Abweichung von über 15%.

Das heißt jedoch nicht, dass das Ergebnis unbrauchbar ist. Es illustriert lediglich, dass es sich um Stichproben handelt, die gemäß dem Gesetz der großen Zahlen umso verlässlicher werden, je mehr wir nehmen. Und 174 Stichproben sind nicht gerade übermäßig viele. Denken Sie außerdem daran, mit welchem Ziel Sie die Daten erheben. Es geht hier nicht um Präzisionsmessungen, sondern in der Regel um Hinweise darauf, welche Teile Ihres Programms Sie optimieren sollten. Relevant sind ohnehin nur die ersten paar Plätze der Rangliste. Und hier sind wiederum die zugehörigen Stacktraces äußerst interessant. Sie sagen Ihnen, welcher Code die besonders kritischen Methoden *aufruft*. Das ist essentiell; denn eine Methode, die gar nicht erst ausgeführt werden muss, ist wesentlich schneller als eine Methode, die optimiert wurde.

Abbildung 4.2 verdeutlicht diesen Zusammenhang durch einen aus den Stacktraces abgeleiteten Methodenaufruf-Graph. Er zeigt, dass die `main()`-Methode offensichtlich mindestens so viel Rechenzeit in Anspruch nehmen muss wie die drei Arbeitsmethoden zusammen. Die `main()`-Methode ist jedoch in der Rangliste gar nicht aufgelistet.

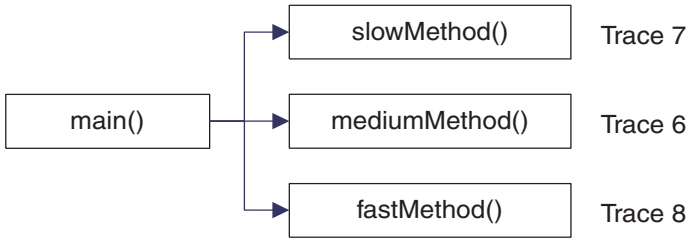

Abbildung 4.2: Aus Stacktraces abgeleiteter Methodenaufruf-Graph

Zur einfacheren Analyse der Rangliste samt ihrer Trace-Verweise können Sie das freie Werkzeug *PerfAnal* von Nathan Meyers benutzen. Das bereits im Heap-Abschnitt erwähnte HPjmeter erweist hier ebenfalls wertvolle Dienste.

- PerfAnal: *http://developer.java.sun.com/developer/technicalArticles/Programming/perfanal/*
- HPjmeter: *http://www.hpjmeter.com/*

Absolute Zeiten

Die `times`-Ausgabe sagt Ihnen genau, wie viel Zeit die VM in welcher Methode verbringt. Sie ist somit scheinbar viel präziser als das `samples`-Format. Scheinbar, da die Zeiten durch das andauernde Messen natürlich auch viel mehr verfälscht werden als durch gelegentliche Stichproben. Jedoch gibt es hier kein eindeutiges Besser oder Schlechter. Letztlich ist es eine Frage des Geschmacks, welches Format für Sie nützlicher ist.

Wir wollen uns die Ausgabe für unsere Testklasse anschauen. Im `times`-Modus wird die Klasse folgendermaßen gestartet:

```
java -Xrunhprof:cpu=times com.tagtraum.perf.cpu.HprofCPUDemo
```

Die Ergebnis-Ausgabe erfolgt wiederum durch Stacktraces und eine Rangliste.

```
TRACE 5:
    HprofCPUDemo.slowMethod(HprofCPUDemo.java:Unknown line)
    HprofCPUDemo.main(HprofCPUDemo.java:Unknown line)
TRACE 2:
    HprofCPUDemo.fastMethod(HprofCPUDemo.java:Unknown line)
```

```
    HprofCPUDemo.main(HprofCPUDemo.java:Unknown line)
TRACE 19:
    HprofCPUDemo.main(HprofCPUDemo.java:Unknown line)
TRACE 4:
    HprofCPUDemo.mediumMethod(HprofCPUDemo.java:Unknown line)
    HprofCPUDemo.main(HprofCPUDemo.java:Unknown line)
...
CPU TIME (ms) BEGIN (total = 17824)
rank   self  accum   count trace method
   1 51.86% 51.86%  100000     5 HprofCPUDemo.slowMethod
   2 27.42% 79.28%  100000     4 HprofCPUDemo.mediumMethod
   3 15.22% 94.50%  100000     2 HprofCPUDemo.fastMethod
   4  4.66% 99.16%       1    19 HprofCPUDemo.main
   5  0.11% 99.27%      43    15 java.lang.String.toLowerCase
...
  18  0.06% 100.00%      2     7 java.io.Win32FileSystem.normalize
CPU TIME (ms) END
```

Die Rangliste ist geordnet nach der Zeit, die die VM in einer Methode verbringt, *exklusive* aller Methodenaufrufe innerhalb dieser Methode. Dementsprechend ist die `main()`-Methode auf Rang vier platziert und die Arbeitsmethoden (`slowMethod()`, `mediumMethod()` und `fastMethod()`) nehmen die vorderen drei Plätze ein.

Die Spalte `self` repräsentiert den prozentualen Anteil der Laufzeit des Programms in der jeweiligen Methode, `accum` ist wiederum die Summe aller `self`-Werte bis zum aktuellen Rang und `count` gibt die absolute Anzahl an Aufrufen an.

Alte Zeiten

Das `old`-Format stammt noch aus alten JDK-1.1-Zeiten. Um diese Ausgabe zu erzeugen, müssen wir unsere Testklasse folgendermaßen starten:

```
java -Xrunhprof:cpu=old com.tagtraum.perf.cpu.HprofCPUDemo
```

Aus Gründen der Rückwärtskompatibilität können Sie alternativ auch `-prof` als Argument angeben. Die Ausgabe erfolgt dann jedoch in die Datei *java.prof* anstelle von *java.hprof.txt*:

```
java -prof com.tagtraum.perf.cpu.HprofCPUDemo
```

Das Format sieht völlig anders aus als die beiden anderen. Es gibt lediglich eine Sektion, in der alle Methodenaufrufe nach Häufigkeit sortiert aufgelistet sind.

```
count callee
   caller time
100000 HprofCPUDemo.slowMethod()V
   HprofCPUDemo.main([Ljava/lang/String;)V 8542
100000 HprofCPUDemo.mediumMethod()V
   HprofCPUDemo.main([Ljava/lang/String;)V 4105
```

```
100000 HprofCPUDemo.fastMethod()V
   HprofCPUDemo.main([Ljava/lang/String;)V 2473
750 java.util.jar.Attributes$Name.isValid(C)Z
   java.util.jar.Attributes$Name.isValid(Ljava/lang/String;)Z 20
750 java.lang.String.charAt(I)C
   java.util.jar.Attributes$Name.isValid(Ljava/lang/String;)Z 0
750 java.util.jar.Attributes$Name.isAlpha(C)Z
   java.util.jar.Attributes$Name.isValid(C)Z 0
...
1 com.tagtraum.perf.cpu.HprofCPUDemo.main([Ljava/lang/String;)V
   <unknown caller> 16053
...
```

Eine Zeile gibt jeweils an, wie häufig (`count`) eine Methode (`callee`) von einer anderen Methode (`caller`) aufgerufen und wie viel Zeit in ms (`time`) dafür benötigt wurde. Anders als im `samples`-Format versteht sich die Zeit hier *inklusive* aller Sub-Methodenaufrufe. Dementsprechend dauert der einzige Aufruf unserer `main()`-Methode (16.053ms) etwas länger als die Summe der Werte der in `main()` aufgerufenen Methoden (insgesamt 15.120ms).

Das `old`-Format ist in seiner Auflistung ein wenig geschwätziger als die anderen Formate. So wird nicht nur der Methodenname angegeben. Zusätzlich werden auch die Argument- und Rückgabetypen (Tabelle 4.3) aufgelistet.

Symbol	**Bedeutung**
[<Typ>	Eindimensionaler Array
L<Klassenname>	Klasse (Ljava/lang/String steht beispielsweise für `java.lang.String`)
$<Klassenname>	Innere Klasse
<init>	Konstruktor
<clinit>	Klassen-Initialisierer
Z	`boolean`
B	`byte`
S	`short`
C	`char`
I	`int`
J	`long`
F	`float`
D	`double`
V	`void`

Tabelle 4.3: Im old-Format verwendete Symbole und ihre Bedeutung

Natürlich gibt es auch für das `old`-Format freie Anzeigeprogramme. Eines davon ist *ProfileViewer* von Greg White und Ulf Dittmer.

- ProfileViewer: *http://www.capital.net/~dittmer/profileviewer/index.html*

4.2.3 Monitor-Information

Die Java-Plattform bietet dem Entwickler volle Unterstützung zur Verwendung von Threads. Fluch und Segen liegen hier nahe beieinander. Wenngleich Threads viele Probleme lösen, so schaffen sie auch einige. Das größte ist wohl die mit Threads einhergehende Komplexität, die wiederum zu Wartungs- und Debug-Problemen führt.

Für uns sind insbesondere als `synchronized` markierte Blöcke von Interesse. Diese Blöcke werden von so genannten Monitoren vor der gleichzeitigen Ausführung durch mehreren Threads geschützt. Gleich einem Staffelstab wird der Monitor eines synchronisierten Blocks von Thread zu Thread gereicht, und nur der Thread, der den Monitor besitzt, darf den Block ausführen.

Ein Monitor ist immer mit einem Objekt assoziiert. Wenn Sie zum Beispiel eine Klassenmethode synchronisieren, so wird der Monitor des Klassenobjekts benutzt. Synchronisieren Sie eine normale Methode, so wird der Monitor der Instanz (also der von `this`) benutzt. Außerdem können Sie das Objekt, dessen Monitor einen synchronisierten Block schützen soll, auch in Klammern hinter dem `synchronized`-Schlüsselwort angeben:

```
synchronized (monitorObject) {
   // mache etwas
}
```

Ein Objekt, das zum Synchronisieren von Threads benutzt wird, heißt auch Lock (Schloss). Es ist quasi der Beschützer eines synchronisierten Blocks.

Jeder synchronisierte Block ist in einer multithreaded-Umgebung per Definition ein Nadelöhr. Unter bestimmten Bedingungen kann die Ausführung des Programms sogar ganz gestoppt werden (z. B. durch ein Deadlock). Es ist daher wichtig, herausfinden zu können, welcher Thread in einer bestimmten Situation gerade auf welchen Monitor wartet.

```
01 package com.tagtraum.perf.monitor;
02
03 public class HprofMonitorDemo extends Thread {
04
05    private static class Lock extends Object {
06    };
07
08    private static Lock lock = new Lock();
09
10    public HprofMonitorDemo(int i) {
```

```
11        super("HprofMonitorThread-" + i);
12     }
13
14     public void run() {
15        for (int i = 0; i < 1000; i++) {
16           obtainLockAndSleep();
17        }
18     }
19
20     private void obtainLockAndSleep() {
21        synchronized (lock) {
22           try {
23              Thread.sleep(100);
24           } catch (InterruptedException ie) {
25              ie.printStackTrace();
26           }
27        }
28     }
29
30     public static void main(String[] args) {
31        HprofMonitorDemo demoInstance0 = new HprofMonitorDemo(0);
32        HprofMonitorDemo demoInstance1 = new HprofMonitorDemo(1);
33        HprofMonitorDemo demoInstance2 = new HprofMonitorDemo(2);
34        demoInstance0.start();
35        demoInstance1.start();
36        demoInstance2.start();
37     }
38  }
```

Listing 4.3: Demo-Programm für den Monitor-Dump von Hprof

Genau dies können Sie erreichen, indem Sie die Hprof-Option `monitor=y` setzen. Zur Illustration starten wir die Klasse *HprofMonitorDemo* aus Listing 4.3. Alle Instanzen von *HprofMonitorDemo* sind Threads, die um ein gemeinsames Lock-Objekt konkurrieren (Zeile 21) und sobald sie es besitzen, 100 ms warten (Zeile 23). Gestartet wird das Programm folgendermaßen:

```
java -Xrunhprof:monitor=y,doe=n
   com.tagtraum.perf.monitor.HprofMonitorDemo
```

Um einen Schnappschuss der Konkurrenz-Situation einzufangen, erzeugen wir noch zur Laufzeit einen Dump durch Strg Pause bzw. Strg \. Als Ergebnis erhalten wir in der Datei *java.hprof.txt* folgende Ausgabe:

```
...
THREAD START (obj=881960, id = 6, name="Thread-0", group="main")
THREAD START (obj=8a039f0, id = 7, name="HprofMonitorThread-0",
   group="main")
THREAD START (obj=8a03ae0, id = 8, name="HprofMonitorThread-1",
```

```
   group="main")
THREAD START (obj=8a03c08, id = 9, name="HprofMonitorThread-2",
   group="main")
...
MONITOR DUMP BEGIN
    MONITOR HprofMonitorDemo$Lock(8a03be8)
    owner: thread 8, entry count: 2
    waiting to enter: thread 9, thread 7
...
MONITOR DUMP END
...
```

Der erzeugte Monitor-Dump zeigt die Situation, dass Thread acht gerade den Monitor `HprofMonitorDemo$Lock(8a03be8)` besitzt und zwei andere Threads ihn gerne besäßen (`entry count: 2`). Die beiden wartenden Threads sind die Threads neun und sieben. Aus dem oberen Teil der Ausgabe können wir schließen, dass der Thread mit der Id neun dem Thread namens `HprofMonitorThread-2` entspricht, Thread sieben `HprofMonitorThread-0` und Thread acht `HprofMonitorThread-1`.

Vereinfacht heißt das: Je höher der Wert `entry count`, umso begehrter ist die synchronisierte Ressource.

Lässt man die Testklasse ohne Unterbrechung durchlaufen, so erhält man zudem noch die Sektion Monitor-Time:

```
MONITOR TIME BEGIN (total = 10 ms)
rank    self   accum   count trace
         monitor
   1 100.00% 100.00%    1499     3
         HprofMonitorDemo$Lock(8a03be8) (Java)
MONITOR TIME END
```

Hierbei handelt es sich wiederum um eine Rangliste. Diesmal um eine mit den am häufigsten besetzten Monitoren. Wie nicht anders zu erwarten ist dies in unserem Beispiel eine Instanz der Klasse `HprofMonitorDemo$Lock`. Die Spalte `self` gibt an, für wie viel Prozent aller besetzter Monitore gerade der aufgelistete Monitor verantwortlich ist. `accum` ist wiederum die Summe aller gleich- oder höherrangigen `self`-Werte. `count` gibt die Anzahl der Stichproben an, während der dieser Monitor bereits besetzt war. `trace` verweist auf die Methode, in der der Monitor aus der `monitor`-Spalte nicht erlangt werden konnte.

Während die `monitor`-Option sehr nützlich erscheint, so ist sie es tatsächlich nur begrenzt. Insbesondere im Zusammenhang mit Swing sind Komplettabstürze häufig, zusammen mit anderen Hprof-Optionen ist die `monitor`-Option fast gar nicht einsetzbar.

Verlässlicher hingegen ist der Thread-Dump, der auch ohne Hprof nach einem [Strg] [Pause] bzw. [Strg] [\] in die Standardausgabe geschrieben wird. Dieser enthält zwar keine statistischen, dafür aber andere wertvolle Informationen.

```
Full thread dump:

"Thread-0" prio=5 tid=0x2345a0 nid=0x5f0
   waiting on monitor [0..0x6fb30]

"HprofMonitorThread-2" prio=5 tid=0x825a60 nid=0x498
   waiting on monitor [0x8e0f000..0x8e0fdbc]
        at java.lang.Thread.sleep(Native Method)
        at HprofMonitorDemo.obtainLockAndSleep
           (HprofMonitorDemo.java:23)
        at HprofMonitorDemo.run(HprofMonitorDemo.java:16)

"HprofMonitorThread-1" prio=5 tid=0x8258b0 nid=0x5f4
   waiting for monitor entry [0x8dcf000..0x8dcfdbc]
        at HprofMonitorDemo.obtainLockAndSleep
           (HprofMonitorDemo.java:23)
        at HprofMonitorDemo.run(HprofMonitorDemo.java:16)

"HprofMonitorThread-0" prio=5 tid=0x824e98 nid=0x66c
   waiting for monitor entry [0x8d8f000..0x8d8fdbc]
        at HprofMonitorDemo.obtainLockAndSleep
           (HprofMonitorDemo.java:23)
        at HprofMonitorDemo.run(HprofMonitorDemo.java:16)

"Signal Dispatcher" daemon prio=10 tid=0x800140 nid=0x588
   waiting on monitor [0..0]

"Finalizer" daemon prio=9 tid=0x8990e40 nid=0x668
   waiting on monitor [0x8c4f000..0x8c4fdbc]
        at java.lang.Object.wait(Native Method)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.ReferenceQueue.remove(Unknown Source)
        at java.lang.ref.Finalizer$FinalizerThread.run
           (Unknown Source)

"Reference Handler" daemon prio=10 tid=0x89901e0 nid=0x34c
   waiting on monitor [0x8c0f000..0x8c0fdbc]
        at java.lang.Object.wait(Native Method)
        at java.lang.Object.wait(Unknown Source)
        at java.lang.ref.Reference$ReferenceHandler.run
           (Unknown Source)

"VM Thread" prio=5 tid=0x89fe128 nid=0x634 runnable

"VM Periodic Task Thread" prio=10 tid=0x7feea0 nid=0x5a4
   waiting on monitor

"Suspend Checker Thread" prio=10 tid=0x7ff7b8 nid=0x638 runnable
```

Der abgedruckte Dump zeigt alle Threads der laufenden VM. Hervorgehoben ist jeweils der Status der drei HprofMonitorThreads. Während `HprofMonitorThread-0` und `HprofMonitorThread-1` sich im so genannten Wait-Set befinden, d.h. darauf warten, den Monitor `[0x8dcf000..0x8dcfdbc]` zu erlangen, besitzt `HprofMonitorThread-2` zwar den Monitor, wartet jedoch darauf, dass die 100 ms `sleep()`-Zeit verstreichen.

Hätten wir `lock.wait(100)` anstelle von `Thread.sleep(100)` aufgerufen, wäre übrigens die Wahrscheinlichkeit sehr groß gewesen, dass keiner der drei Threads zum Zeitpunkt des Thread-Dumps im Besitz des Monitors ist und alle auf den Monitor warten. Zur Erinnerung: Beim Aufruf von `object.wait()` gibt der ausführende Thread alle Monitore auf, beim Aufruf von `Thread.sleep()` hingegen behält und blockiert er sie.

Neben den Status `waiting on monitor` und `waiting for monitor entry` erscheint im Dump zudem die Bezeichnung `runnable` für Threads, die nicht aus dem ein oder anderen Grund blockiert sind. Außerdem finden wir allerlei nützliche Informationen wie zum Beispiel, ob der Thread ein Daemon-Thread ist, welche Priorität er hat, wie seine Objekt-Id lautet (`tid`) und von welchem Betriebssystemthread er ausgeführt wird (`nid`).

Threaddumps sind je nach VM und Betriebssystem leicht unterschiedlich. Der oben abgebildete Dump stammt vom Sun JDK 1.3.1 und wurde unter Windows 2000 erzeugt.

Mehr zum Thema Threads finden Sie in *Kapitel 9 Threads*.

4.3 HotSpot-Profiling

Um Programme mit eingeschaltetem HotSpot zu messen, können Sie Suns Java-VM mit der Option `-Xprof` starten. Der Aufruf unseres Beispielprogramms lautet folgendermaßen:

```
java -Xprof com.tagtraum.perf.cpu.HprofCPUDemo
```

Beachten Sie, dass hierdurch nicht der Hprof-Profiler gestartet wird, sondern ein spezieller HotSpot-Profiler. Die Ausgabe erfolgt auch nicht in eine Datei, sondern in den Standard-Ausgabestrom und gliedert sich in mehrere Abschnitte. Jeder Thread hat dabei seine eigene Sektion, eingeleitet jeweils durch die Zeile `Flat profile of XX.XX secs (YYYY total ticks): <Name des Threads>`. Jede Thread-Sektion ist wiederum unterteilt in mehrere Subsektionen. Genau wie im `samples`-Modus von Hprof erfolgt das Messen stichprobenbasiert. Eine Stichprobe entspricht hier jeweils einem `tick`.

Für unser Beispiel-Programm lautet die Ausgabe wie folgt:

```
Flat profile of 14.72 secs (1467 total ticks): main

  Interpreted + native   Method
  0.2%     2  +     1    HprofCPUDemo.main
```

```
  0.1%     2  +     0    HprofCPUDemo.mediumMethod
  0.1%     0  +     1
      java.security.AccessController.doPrivileged
  0.1%     0  +     1    java.util.jar.JarVerifier.<init>
  0.1%     0  +     1    java.io.BufferedReader.readLine
  0.1%     0  +     1    sun.security.provider.Sun$1.run
  0.1%     0  +     1    java.io.Win32FileSystem.getLength
  0.7%     4  +     6    Total interpreted

     Compiled + native   Method
 54.1%   793  +     0    HprofCPUDemo.slowMethod
 30.9%   454  +     0    HprofCPUDemo.mediumMethod
 14.1%   207  +     0    HprofCPUDemo.fastMethod
 99.1%  1454  +     0    Total compiled

  Thread-local ticks:
  0.2%     3             Class loader

Global summary of 14.82 seconds:
100.0%  1477             Received ticks
  0.1%     2             Compilation
  0.2%     3             Class loader
```

Da wir implizit nur einen, nämlich den `main`-Thread starten, haben wir auch nur eine threadspezifische Sektion, gefolgt von einer globalen Zusammenfassung aller genommenen Stichproben.

Der threadspezifische Abschnitt besteht aus mehreren Teilen: einem für Methoden, die interpretiert wurden, einem für Ergebnisse von kompilierten Methoden sowie einem threadspezifischen. Alle Teile bestehen aus vier Spalten. Die erste Spalte ist jeweils der prozentuale Anteil einer Methode an der Gesamtlaufzeit des Threads, die zweite Spalte enthält die Anzahl an Ticks, die im Java-Teil der Methode verbracht wurden, und die dritte Spalte ist die Anzahl an Ticks, die in einer nativen Subroutine verbracht wurden. Spalte vier schließlich beinhaltet den Namen der Methode.

Interessant für uns sind jeweils die Zeilen, die mit hohen Prozentzahlen beginnen. In unserem Beispiel sind das insbesondere die ersten drei Zeilen des `Compiled`-Abschnitts. Dort finden wir die drei Arbeitsmethoden unserer Beispielklasse wieder. Die Prozentzahlen entsprechen im Groben denen der `self`-Spalten im `samples`- und `times`-Format – keine Überraschungen also.

Eine Eigenheit der `-Xprof`-Ausgabe ist es, dass für jeden Thread nur jene Abschnitte ausgegeben werden, in denen auch tatsächlich etwas gemessen wurde. Während unsere Beispiel-Klasse Ausgaben für kompilierte und interpretierte Methoden erzeugt, fehlen die beiden optionalen Abschnitte `Stub` und `Runtime`.

Im `Stub`-Abschnitt werden die Aufrufe von JNI-Methoden zusammengefasst. `Runtime` enthält die Stichproben, während der die VM mit sich selbst beschäftigt war.

```
Flat profile of 21.43 secs (2092 total ticks): main
...
          Stub + native   Method
  9.7%     0  +   197    java.util.zip.ZipFile.getEntry
  6.3%     1  +   127    java.util.zip.ZipFile.read
  2.7%     0  +    55    java.util.zip.ZipFile.getNextEntry
  0.5%     0  +    11    java.lang.Throwable.fillInStackTrace
...
 19.8%     1  +   401    Total stub

  Runtime stub + native  Method
  0.0%     1  +     0    interpreter_entries
  0.0%     1  +     0    Total runtime stubs
...
```

Aufgrund der fehlenden Stacktraces hat die Ausgabe des HotSpot-Profilers nur begrenzten Wert. Es ist nicht ersichtlich, welche Methoden diejenigen Methoden aufrufen, in denen Ihr Programm die meiste Zeit verbringt. Es fehlen somit die Daten für einen Methodenaufruf-Graph. Die Ausgabe kann daher lediglich als Startpunkt für eine Analyse dienen, die auch die Daten aus Testläufen mit Hprof oder einem anderen Profiler berücksichtigt. Verzichten Sie auf diese Analyse, optimieren Sie unter Umständen eine Methode, die offensichtlich sehr häufig aufgerufen wird, ohne zu überprüfen, ob diese Methode wirklich so häufig aufgerufen werden muss.

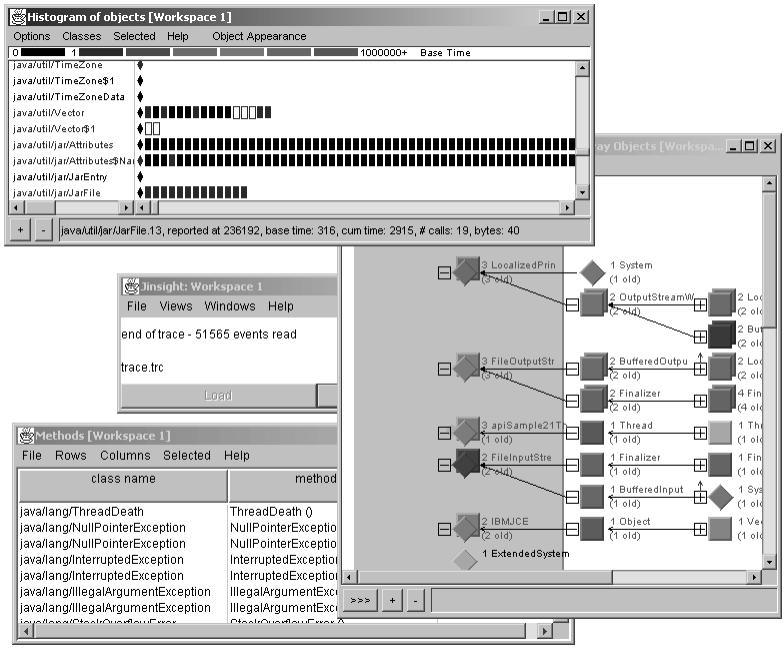

Abbildung 4.3: Jinsight von IBM alphaworks

4.4 Jinsight

Neben Hprof gibt es für die IBM VM noch einen freien Profiler namens *Jinsight*. Jinsight funktioniert genau wie der oben beschriebene Hprof, ist jedoch dank einer grafischen Benutzeroberfläche einfacher zu bedienen. Wenn Sie ein IBM JDK verwenden, lohnt es sich auf jeden Fall, einen Blick auf Jinsight zu werfen.

- Jinsight von alphaworks IBM: *http://www.alphaworks.ibm.com/tech/jinsight*

4.5 Mikro-Benchmarks

Zum Messen der Ausführungszeit einer bestimmten Methode können Sie anstelle eines Profilers auch die Methode `java.lang.System.currentTimeMillis()` benutzen. Sie eignet sich insbesondere für Mikro-Benchmarks. Der wohl beliebteste Performance-Test mittels `System.currentTimeMillis()` sieht etwa so aus:

```
...
long start = System.currentTimeMillis();
int iterations = 10000;
for (int i=0; i<iterations; i++) {
   objectToTest.methodToTest();
}
long time = System.currentTimeMillis()-start;
System.out.println("Benötigte Zeit für "
   + iterations + " Iterationen: "
   + time + "ms");
```

Listing 4.4: Standard Mikro-Benchmark mit System.currentTimeMillis()

Wenn Sie solche Mikro-Benchmarks durchführen, behalten Sie folgende Punkte im Hinterkopf:

- Der Aufruf von `System.currentTimeMillis()` benötigt selbst einige Millisekunden und das Ergebnis ist nicht unbedingt auf die Millisekunde genau. Unter Windows NT/2000 beträgt die Genauigkeit 10 ms, unter Windows 95 sogar nur 50 ms. Wählen Sie daher für die Variable `iterations` einen Wert, der zu einer Gesamtlaufzeit führt, die um Größenordnungen länger ist als die Genauigkeit von `System.currentTimeMillis()` Ihrer Testplattform. Liegt die Laufzeit der Methode selbst im Milli- oder Hundertstelsekunden-Bereich, führt eine Laufzeit länger als fünf Sekunden in der Regel zu sinnvollen Ergebnissen. Bei langsameren Methoden sollten Sie die Laufzeit erhöhen, so dass Sie die Methode wenigstens einige hundert Mal aufrufen.
- HotSpot ist unter Umständen nicht in der Lage, Code, der nicht in einer Extra-Methode steht, beim ersten Lauf zu kompilieren. Rufen Sie daher Ihre Benchmark-Methode zweimal auf.

4.6 Makro-Benchmarks

Sie können `System.currentTimeMillis()` natürlich genauso gut für Makro-Benchmarks verwenden. Bei größeren Applikationen und insbesondere bei Webapplikation lohnt es sich jedoch, auf vorhandene Werkzeuge zurückzugreifen. So können Sie beispielsweise die Performance einer Webanwendung mit *Apache JMeter* messen. JMeter ist ein freies Werkzeug und eignet sich ferner für Stress-Tests von FTP-Servern sowie beliebigen JDBC-Anfragen.

- Apache Jmeter: *http://jakarta.apache.org/jmeter/index.html*

Außerdem gibt es noch einige kommerzielle Werkzeuge zum Messen der Performance eines Systems. Hier eine wertfreie Auswahl:

- Mercury Interactive LoadRunner: *http://www.mercuryinteractive.com/products/loadrunner/*
- Segue SilkPerformer: *http://www.segue.com/html/s_solutions/s_performer/s_performer.htm*
- Empirix e-Load: *http://www.empirix.com/*

4.7 Performance Metriken

Oft ist es auch sinnvoll, Performancedaten über installierte Applikationen zu erheben und zu analysieren. Dabei ist häufig nicht interessant, wie schnell eine einzelne Methode ausgeführt werden konnte, sondern wie lange es dauerte, beispielsweise eine Bestellung aufzunehmen.

Indem man Messpunkte in eine Applikation kodiert, kann man diese Daten relativ einfach erheben. Natürlich ist dies auch über Logdateien möglich – die sind jedoch meist sehr umständlich auszuwerten.

Unterstützung für Messpunkte wird vielleicht bald Bestandteil der Java-Plattform sein. Es existiert bereits eine entsprechende Spezifikationsanforderung (*http://www.jcp.org/jsr/detail/138.jsp*).

4.8 Speicher-Schnittstellen

Die Java-Klassenbibliothek bietet einige wenige Schnittstellen, um Informationen über den Speicherverbrauch zu erhalten sowie im begrenzten Maße mit dem Garbage Collector zu interagieren. Exakte Information über die Garbage Collection sind jedoch nur über einen Kommandozeilen-Parameter der VM zu erhalten.

4.8.1 Speicherverbrauch

Von Ihrem Programm aus können Sie feststellen, wie viele Byte der Heap groß ist und wie viele davon noch frei sind. Die entsprechenden Methodenaufrufe lauten `Runtime.getRuntime().totalMemory()` und `Runtime.getRuntime().freeMemory()`. Beide Größen sind nicht-statisch und ändern sich mit der Zeit. Wenn Sie also den Speicherverbrauch Ihrer Applikation protokollieren möchten, so können Sie einfach einen Thread starten und periodisch die beiden Speicherwerte in eine Datei oder die Standardausgabe schreiben. Der entsprechende Code dafür ist denkbar einfach (Listing 4.5). Seit JDK 1.4 existiert zudem eine Methode `Runtime.getRuntime().maxMemory()`, die angibt, wie viel Speicher die VM maximal beanspruchen wird.

```
package com.tagtraum.perf.memory;

public class MemoryWriter extends Thread {
   public MemoryWriter() {
      super("MemoryWriter");
      // damit die VM sauber heruntergefahren werden kann:
      setDaemon(true);
      System.out.println("Total\tFree");
      start();
   }

   public synchronized void run() {
      try {
         while (true) {
            System.out.println(Runtime.getRuntime().totalMemory()
               + "\t" + Runtime.getRuntime().freeMemory());
            wait(500);
         }
      } catch (InterruptedException ie) {
         ie.printStackTrace();
      }
   }
}
```

Listing 4.5: Einfacher MemoryWriter

Der `MemoryWriter` muss nur noch an geeigneter Stelle instanziiert werden – dazu bietet sich beispielsweise die `main()`-Methode an. Alternativ können Sie auch einen Applikationsstarter schreiben, der zunächst einen Thread zum Schreiben des Speicherzustandes startet und dann die `main()`-Methode einer beliebigen Java-Applikation aufruft. `MemViewer` (Listing 4.6, Abbildung 4.4) ist ein solcher Applikationsstarter. Zum besseren Verständnis des Quellcodes zeigt das Klassendiagramm in Abbildung 4.5 grob den statischen Aufbau von `MemViewer`.

`MemViewer` lässt sich mit einer simplen grafischen Oberfläche, einer textuellen Ausgabe oder einem beliebigen `Visualizer` starten. Welcher `Visualizer` gestartet wird, hängt vom ersten Argument ab, das Sie `MemViewer` beim Start übergeben. Da die Textausgabe sehr viel einfacher ist als die GUI-Ausgabe, verfälscht sie das Messergebnis weniger. Dank des simplen Formats (TSV) kann sie von einer handelsüblichen Tabellenkalkulation importiert und in einem Diagramm visualisiert werden. Die GUI-Ausgabe hat jedoch den Vorteil, dass Sie Ihnen direkt zur Laufzeit ein visuelles Feedback gibt.

Hier sind die möglichen Startparameter:

```
java com.tagtraum.perf.memviewer.MemViewer
   text[:file=<filename>]|gui|<visualizer classname>
   <main classname>
```

<`main classname`> steht dabei für die Haupt-Klasse der Applikation, die Sie starten wollen.

Abbildung 4.4: Typisches Sägezahnmuster des belegten Heap-Speichers

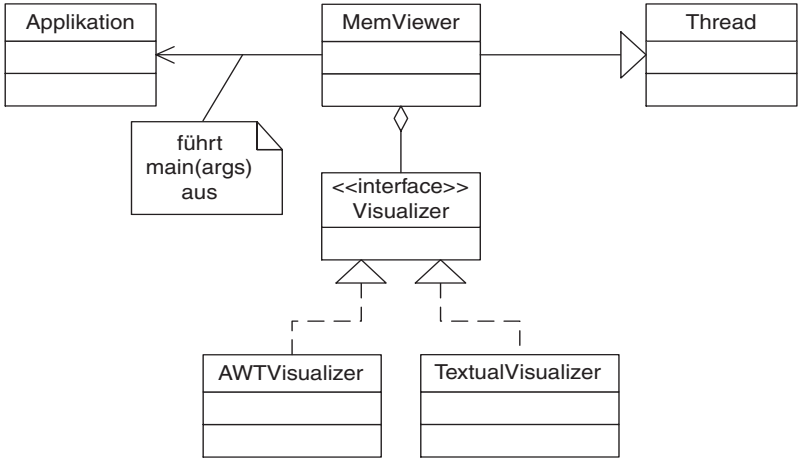

Abbildung 4.5: Klassendiagramm des MemViewers

```
package com.tagtraum.perf.memviewer;

import java.lang.reflect.Method;
import java.util.Properties;
import java.util.StringTokenizer;

//Hauptklasse des MemViewers.
public class MemViewer extends Thread {

   private Visualizer visualizer;
   private long interval = 1000;

   public MemViewer(String arg) throws Exception {
      StringTokenizer st = new StringTokenizer(arg, ":");
      String[] args = new String[st.countTokens() - 1];
      String type = st.nextToken();
      for (int i = 0; st.hasMoreTokens(); i++) {
         args[i] = st.nextToken();
      }
      Class visualizerClass = getVisualizerClass(type);
      visualizer = (Visualizer) visualizerClass.newInstance();
      visualizer.init(getProperties(args));
   }

   public synchronized void run() {
      while (true) {
         visualizer.showMemory(new MemState());
         try {
            wait(interval);
         } catch (InterruptedException ie) {
            ie.printStackTrace();
         }
      }
   }

   public static void main(String[] args) throws Exception {
      if (args.length < 2) usage();
      MemViewer memViewer = new MemViewer(args[0]);
      memViewer.setDaemon(true);
      memViewer.start();
      Method mainMethod = Class.forName(args[1]).getMethod("main",
         new Class[]{String[].class});
      String[] newArgs = new String[args.length - 1];
      System.arraycopy(args, 1, newArgs, 0, newArgs.length);
      mainMethod.invoke(null, new Object[]{newArgs});
   }

   public static void usage() {
      System.out.println("MemViewer");
      System.out.println("java "
         + "com.tagtraum.perf.memviewer.MemViewer "
```

```
            + "text[:file=<filename>]|gui|<visualizer classname> "
            + "<main classname>");
         System.exit(0);
      }

      private static Properties getProperties(String[] args) {
         Properties properties = new Properties();
         for (int i = 0; i < args.length; i++) {
            int index = args[i].indexOf('=');
            if (index != -1 && index < args[i].length() - 1) {
               properties.setProperty(args[i].substring(0,
                   index).toLowerCase(), args[i].substring(index + 1));
            }
         }
         return properties;
      }

      private static Class getVisualizerClass(String type)
            throws ClassNotFoundException {
         Class visualizerClass;
         if (type.equals("text")) {
            visualizerClass = TextualVisualizer.class;
         } else if (type.equals("gui")) {
            visualizerClass = AWTVisualizer.class;
         } else {
            visualizerClass = Class.forName(type);
         }
         return visualizerClass;
      }

   }
```

Listing 4.6: Klasse MemViewer

```
   package com.tagtraum.perf.memviewer;

   import java.util.Date;

   //Schnappschuss des Speicherzustandes.
   public class MemState {
      private long totalMemory;
      private long freeMemory;
      private long time;

      public MemState() {
         this.totalMemory = Runtime.getRuntime().totalMemory();
         this.freeMemory = Runtime.getRuntime().freeMemory();
         this.time = System.currentTimeMillis();
      }
```

```
    public long getTotalMemory() {
        return totalMemory;
    }

    public long getFreeMemory() {
        return freeMemory;
    }

    public long getUsedMemory() {
        return totalMemory - freeMemory;
    }

    public long getUsedMemoryPercent() {
        return getUsedMemory() * 100 / getTotalMemory();
    }

    public long getFreeMemoryPercent() {
        return getFreeMemory() * 100 / getTotalMemory();
    }

    public long getTime() {
        return time;
    }

    public String toString() {
        return new Date(getTime()).toString()
            + ": Total: " + getTotalMemory()
            + ", Free: " + getFreeMemory()
            + " (" + getFreeMemoryPercent() + "%)"
            + ", Used: " + getUsedMemory()
            + " (" + getUsedMemoryPercent() + "%)";
    }
}
```

Listing 4.7: Die Klasse MemState bildet den Speicherzustand zu einem gegebenen Zeitpunkt ab.

```
package com.tagtraum.perf.memviewer;

import java.util.Properties;

// Visualizer Interface.
public interface Visualizer {
    public void init(Properties properties) throws Exception;
    public void showMemory(MemState memState);
}
```

Listing 4.8: Interface Visualizer

```
package com.tagtraum.perf.memviewer;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.util.Date;
import java.util.Properties;

// TextualVizualizer schreibt den Speicherzustand in
// eine Datei oder die Standardausgabe.
public class TextualVisualizer implements Visualizer {

   private PrintWriter out;

   public void init(Properties properties) throws Exception {
      if (properties.containsKey("file")) {
         out = new PrintWriter(
            new FileWriter(properties.getProperty("file")));
      } else {
         out = new PrintWriter(System.out);
      }
      out.println("MemViewer - " + new Date());
      // drucke Tabellenkopf
      out.println("Time\tTotal\tFree\tFree%\tUsed\tUsed%");
   }

   public void showMemory(MemState memState) {
      out.print(memState.getTime());
      out.print('\t');
      out.print(memState.getTotalMemory());
      out.print('\t');
      out.print(memState.getFreeMemory());
      out.print('\t');
      out.print(memState.getFreeMemoryPercent());
      out.print('\t');
      out.print(memState.getUsedMemory());
      out.print('\t');
      out.println(memState.getUsedMemoryPercent());
   }
}
```

Listing 4.9: Klasse TextualVizualizer

```
package com.tagtraum.perf.memviewer;

import java.awt.*;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;
import java.text.NumberFormat;
import java.util.Properties;

// AWTVisualizer gibt den Speicherzustand in einem
```

```
// AWT-Fenster aus. Es werden dabei jeweils die letzten 2048
// Einträge im Speicher gehalten.
public class AWTVisualizer extends Frame implements Visualizer {

   private MemStatesRingBuffer memStates;
   private long maxTotal;
   private double scaleFactor;
   private Polygon totalPolygon;
   private Polygon usedPolygon;
   private int currentFrameHeight;
   private NumberFormat numberFormat;

   public AWTVisualizer() {
      super("MemViewer");
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            hide();
            dispose();
         }
      });
      numberFormat = NumberFormat.getInstance();
      memStates = new MemStatesRingBuffer();
      totalPolygon = new Polygon();
      usedPolygon = new Polygon();
      setSize(450, 200);
      show();
   }

   public void init(Properties properties) {
      // wird nicht benutzt.
   }

   public void showMemory(MemState memState) {
      memStates.add(memState);
      update(getGraphics());
   }

   public void update(Graphics g) {
      if (memStates.getLast().getTotalMemory() > maxTotal
            || getHeight() != currentFrameHeight) {
         maxTotal = memStates.getLast().getTotalMemory();
         currentFrameHeight = getHeight();
         scaleFactor = (double)maxTotal/(double)currentFrameHeight;
      }
      computePolygons();
      super.update(g);
   }

   public void paint(Graphics g) {
      super.paint(g);
      g.setColor(Color.blue);
```

```
        g.fillPolygon(totalPolygon);
        g.setColor(Color.red);
        g.fillPolygon(usedPolygon);
        g.setColor(Color.black);
        String line = (numberFormat.format(memStates.getLast().
           getUsedMemory()/1024)) + "KB / " + (numberFormat.format(
           memStates.getLast().getTotalMemory()/1024)) + "KB";
        g.drawString(line, 10, currentFrameHeight - 10);
        setTitle("MemViewer - " + line);
    }

    private void computePolygons() {
        totalPolygon = new Polygon();
        usedPolygon = new Polygon();
        int x = 0;
        totalPolygon.addPoint(x, currentFrameHeight);
        usedPolygon.addPoint(x, currentFrameHeight);
        for (int i = Math.max(memStates.size() - getWidth(), 0);
              i < memStates.size(); i++, x++) {
           MemState memState = (MemState) memStates.get(i);
           totalPolygon.addPoint(
              x, scale(memState.getTotalMemory())
           );
           usedPolygon.addPoint(x, scale(memState.getUsedMemory()));
        }
        totalPolygon.addPoint(x, currentFrameHeight);
        usedPolygon.addPoint(x, currentFrameHeight);
    }

    private int scale(long memory) {
        return currentFrameHeight-(int)((double)memory/scaleFactor);
    }

     // Private Datenstruktur, die jeweils die letzten 2048
     // Elemente hält.

    private static class MemStatesRingBuffer {
        private MemState[] buffer = new MemState[2048];
        private int last;
        private int first;

        public void add(MemState memState) {
           buffer[last] = memState;
           last = ++last % buffer.length;
           if (last == first) first = ++first % buffer.length;
        }

        public MemState get(int index) {
           if (index >= size() || index < 0)
              throw new IndexOutOfBoundsException("Index: "
```

```
                + index + ", Size: " + size());
            return buffer[(first + index) % buffer.length];
        }

        public MemState getLast() {
            return get(size() - 1);
        }

        public int size() {
            if (first <= last) {
                return last - first;
            }
            return last + (buffer.length - first);
        }

    }
}
```

Listing 4.10: Klassen AWTVisualizer und MemStatesRingBuffer

4.8.2 Geschwätzige Garbage Collection

Insbesondere zum Optimieren der VM-Parameter von Server-Programmen ist es sinnvoll, sich genauer anzuschauen, wann der Heap vom Garbage Collector aufgeräumt wird. Die besten Daten dazu erhalten Sie, wenn Sie den Parameter `-verbose:gc` beim Start Ihres Programms angeben.

```
java -verbose:gc <main classname>
```

Die Ausgabe erfolgt in die Standardausgabe und sieht für Sun JDK 1.3.1 wie folgt aus:

```
[GC 511K->223K(1984K), 0.0100300 secs]
[GC 979K->565K(1984K), 0.0090931 secs]
[GC 1073K->805K(1984K), 0.0209879 secs]
[GC 1150K->960K(1984K), 0.0134366 secs]
[Full GC 960K->898K(2076K), 0.1322406 secs]
[Full GC 898K->898K(2076K), 0.1231679 secs]
[GC 2273K->1864K(2940K), 0.0062519 secs]
...
```

Jede Zeile enthält Angaben über die Art der Speicherbereinigung (z.B. in der ersten Zeile: `GC`), den belegten Speicher (`511K`) vor und nach der Speicherbereinigung (`223K`), die Größe des Heaps (`1984K`) sowie die Dauer der Speicherbereinigung (`0.0100300 secs`).

In Suns JDK 1.4 gibt es zudem die Option `-Xloggc:<file>`, die dafür sorgt, dass die Ausgabe in eine Datei geschrieben wird:

```
java -Xloggc:gc.txt <main classname>
```

Das Format gleicht dem oben beschriebenen Format, bis auf den Punkt, dass in der ersten Spalte jeweils ein Zeitstempel steht.

```
2.23492e-006: [GC 1087K->462K(16320K), 0.0154134 secs]
3.42373: [GC 1550K->654K(16320K), 0.0122571 secs]
5.32063: [GC 1736K->957K(16320K), 0.0148594 secs]
7.01177: [GC 2044K->980K(16320K), 0.0083648 secs]
8.59185: [GC 2068K->1012K(16320K), 0.0066291 secs]
9.96373: [Full GC 1696K->1028K(16320K), 0.1462759 secs]
12.2401: [GC 2052K->1148K(16320K), 0.0122527 secs]
12.6752: [GC 2167K->1300K(16320K), 0.0085760 secs]
...
```

Die Sun-JDK-1.3.x und JDK-1.4 verfügen jeweils über einen Generationen-Kollektor. Daher gibt es kleine (`GC`) und vollständige (`Full GC`) Speicherbereinigungen. Während der kleinen Speicherbereinigung wird nur der Teil des Heaps aufgeräumt, der für junge Objekte reserviert ist. Dies erfolgt in der Regel sehr viel häufiger und schneller als eine vollständige Bereinigung. Wenn Sie den eingebauten inkrementellen Garbage Collector benutzen, werden Sie zudem viele `Inc GC`-Einträge finden.

Auch zum Anzeigen der Garbage-Collection-Daten existieren Werkzeuge. Eines ist der von mir geschriebene *GCViewer*.

- GCViewer: *http://www.tagtraum.com/*

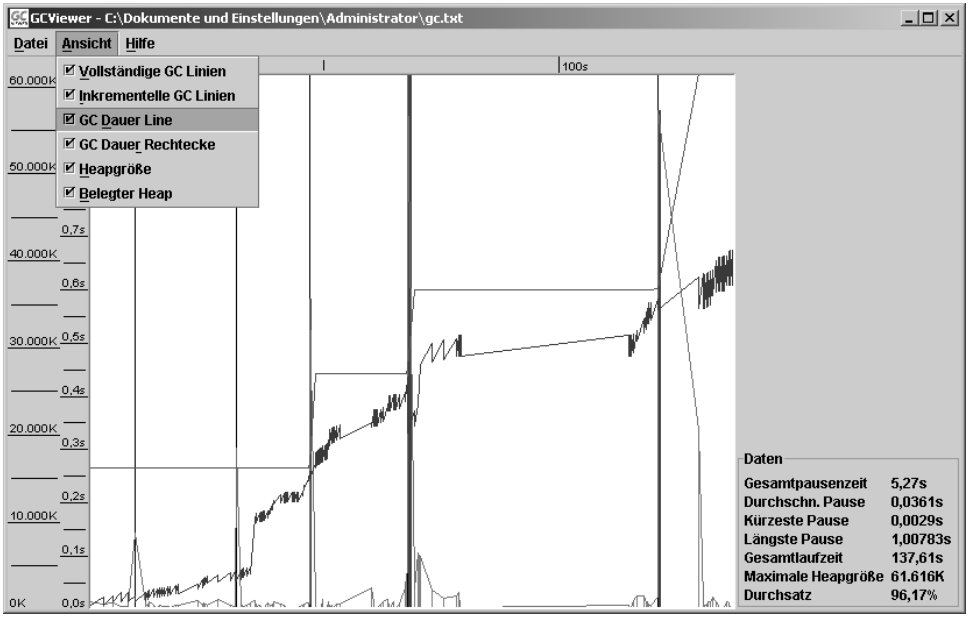

Abbildung 4.6: GCViewer visualisiert die Garbage Collector-Aktivität.

4.8.3 Manuelle Speicherbereinigung

Um aus Ihrem Programm heraus die Garbage Collection anzustoßen, können Sie die Methode `System.gc()` aufrufen. Dies garantiert jedoch nicht, dass tatsächlich der Speicher aufgeräumt wird. Der Methodenaufruf wird vom Garbage Collector lediglich als Hinweis darauf verstanden, dass gerade ein günstiger Moment zum Aufräumen ist.

Grundsätzlich ist vom Aufruf von `System.gc()` jedoch abzuraten, da er zu einer vollständigen Speicherbereinigung führen kann, die unter Umständen unnötig ist und zu einer verhältnismäßig langen Pause der Ausführung des Programms führen kann.

Auf Sun JDK 1.3.1/1.4.0-Systemen lässt sich die explizite Garbage Collection mit dem nicht offiziell unterstützen VM-Parameter `-XX:+DisableExplicitGC` ausschalten. Dies kann jedoch bei Systemen, die RMI und dessen verteilten Garbage Collector (DGC) benutzen, zu Problemen führen, da dieser von expliziter Garbage Collection Gebrauch macht. Siehe *Kapitel 11.3 Verteilte Speicherbereinigung*.

5 Zeichenketten

In Java werden Zeichenketten üblicherweise als `java.lang.String`-Objekte repräsentiert und String-Manipulationen scheinen denkbar einfach. Bietet doch die String-Klasse so komfortable Methoden wie `substring()` zum Erstellen eines Teilstrings oder `trim()` zum Beseitigen von unerwünschten Leerzeichen (genauer: Whitespace). All diese Methoden – es sind noch einige mehr – geben jeweils ein String-Objekt zurück. Nun ist jedoch jedes Objekt der Klasse String *unveränderbar* (*immutable*). Bei jedem zurückgegebenen String-Objekt handelt es sich also um ein *neues* Objekt.

Das Erzeugen von Objekten ist nicht ganz billig. Speicher muss alloziert und sämtliche Konstruktoren müssen ausgeführt werden. Mit anderen Worten: Wenn wir nicht unbedingt müssen, würden wir es gerne vermeiden. Aus diesem Grund verfügt Java über eine zweite Klasse, die explizit zur Manipulation von Zeichenketten gedacht ist: `java.lang.StringBuffer`.

5.1 Strings einfügen

Die wichtigsten `StringBuffer`-Methoden heißen `append()`, `insert()` und `substring()`. Sie ermöglichen das Anfügen und Einfügen von Zeichen oder Zeichenketten sowie das Erstellen von Teilstrings. Im Folgenden wollen wir den Code zum Einfügen eines Strings in einen anderen untersuchen. Zunächst die reine String-Variante:

```
String halloWelt = "Hallo Welt ";
String weite = "weite ";
String hallo = halloWelt.substring(0, 7);
String welt = halloWelt.substring(7);
String halloWeiteWelt = hallo.concat(weite).concat(welt);
```

Das Einfügen von Zeichenketten ist nicht nur vergleichsweise kompliziert, es kostet auch Speicher und Rechenzeit. Jeder Aufruf von `substring()` führt zu einem neuen Objekt, Gleiches gilt für `concat()`. Macht vier neue Objekte plus das `halloWelt`-Objekt und das `weite`-Objekt, insgesamt also sechs Objekte.

Mit der `StringBuffer`-Klasse sieht das etwas anders aus:

```
String halloWelt = "Hallo Welt ";
String weite = "weite ";
StringBuffer stringBuffer = new StringBuffer(halloWelt);
stringBuffer.insert(7, weite);
String halloWeiteWelt = stringBuffer.toString();
```

Wir benötigen nur vier Objekte.

In der String-Version hätten wir auch den +-Operator anstelle von `concat()` benutzen können. Der entsprechende Code sähe folgendermaßen aus:

```
String halloWelt = "Hallo Welt ";
String weite = "weite ";
String hallo = halloWelt.substring(0, 7);
String welt = halloWelt.substring(7);
String halloWeiteWelt = hallo + weite +welt;
```

Um die drei Varianten zu vergleichen, habe ich sie viele Male in einer Schleife ausgeführt, die Zeit gemessen und normalisiert. Wie Tabelle 5.1 zeigt, ist die `StringBuffer`-Variante tatsächlich die schnellste, gefolgt von der String-Variante mit `concat()`. Das Schlusslicht ist die Variante mit dem +-Operator.

String-Variante	StringBuffer-Variante	String-Variante mit +
100%	75,6%	120,3%

Tabelle 5.1: Relative Geschwindigkeit verschiedener Varianten zum Einfügen von Strings in Strings

Warum, fragen Sie sich vielleicht, ist die +-Variante so langsam? Die Antwort steht im Bytecode. Wenn Sie den Quellcode übersetzen und anschließend wieder dekompilieren, können Sie herausfinden, wie der Compiler mit dem +-Operator verfährt. Der dekompilierte Code sieht so aus:

```
String halloWelt = "Hallo Welt ";
String weite = "weite ";
String hallo = halloWelt.substring(0, 7);
String welt = halloWelt.substring(7);
String halloWeiteWelt
   = new StringBuffer(hallo).append(weite).append(welt).toString();
```

Der Compiler sorgt also dafür, dass die `StringBuffer`-Klasse anstelle von `concat()` benutzt wird. So ist es übrigens auch in der Sprachspezifikation vorgesehen [Gosling00, §15.18.1.2]. Dies scheint jedoch in unserem Beispiel zu noch größerem Aufwand und somit zu einer noch schlechteren Zeit zu führen.

Bleibt festzuhalten:

Das Einfügen in Strings erfolgt am besten mit `StringBuffer`.

5.2 Strings anfügen

Nun ist Anfügen gegenüber Einfügen etwas simpler. Wir wollen die oben vorgestellten drei Varianten auf dieselbe Weise untersuchen. Hier die drei entsprechenden Code-Stücke:

```
// concat()
String s = "";
for (int i=0; i<1000; i++) {
   s = s.concat("aString");
}

// StringBuffer
StringBuffer sb = new StringBuffer();
for (int i=0; i<1000; i++) {
   sb.append("aString");
}

// +-Operator
String s = "";
for (int i=0; i<1000; i++) {
   s += "aString";
}
```

Zusätzlich wollen wir noch testen, wie sich der `StringBuffer` verhält, wenn wir ihn mit der erwarteten endgültigen Größe vorinitialisieren. Hier die entsprechende Methode:

```
// initialisierter StringBuffer
StringBuffer sb = new StringBuffer(7000);
for (int i=0; i<1000; i++) {
   sb.append("aString");
}
```

concat()	StringBuffer	Passend initialisierter StringBuffer	+-Operator
100%	1,0%	0,7%	289,1%

Tabelle 5.2: Tausendfaches Anfügen einer Zeichenkette

Das Ergebnis ist eindeutig:

Wenn Sie wiederholt Strings an einen anderen String anfügen, macht sich der `StringBuffer` *bezahlt – insbesondere dann, wenn Sie ihn mit der zu erwartenden Größe vorinitialisieren.*

Am schlechtesten schneidet der +-Operator ab. Zwischen dem passend initialisierten `StringBuffer` und dem +-Operator liegt ein Faktor größer 400.

Der Grund liegt wiederum in der Übersetzung durch den Compiler. Er generiert folgenden Code:

```
// Dekompilat
String s = "";
for (int i=0; i<1000; i++) {
   s = new StringBuffer(s)
      .append("aString").toString();
}
```

Dadurch wird in jedem Schleifendurchlauf je ein zusätzliches `StringBuffer`-Objekt erzeugt. Zudem ist der Aufruf von `toString()` offensichtlich nicht ganz billig.

Zugegeben, der hier dargestellte Mikro-Benchmark ist maßgeschneidert für den `StringBuffer`. Er brilliert besonders, wenn häufig `append()` und selten `toString()` aufgerufen wird, wie es in obigem Test der Fall ist.

Etwas anders sieht es aus, wenn wir den Test ein wenig modifizieren. Anstatt immer denselben String anzufügen, fügen wir die aktuelle Zeichenkette an sich selbst an.

```
// concat()
String s = "aString";
for (int i=0; i<10; i++) {
   s = s.concat(s);
}

// initialisierter StringBuffer
StringBuffer sb = new StringBuffer(7168);
sb.append("aString");
for (int i=0; i<10; i++) {
   sb.append(sb.toString());
}
sb.toString();

// StringBuffer
StringBuffer sb = new StringBuffer("aString");
for (int i=0; i<10; i++) {
   sb.append(sb.toString());
}
sb.toString();

// +-Operator
String s = "aString";
for (int i=0; i<10; i++) {
   s += s;
}
```

In diesem Test ist die `StringBuffer`-Variante etwa genauso schnell wie die +-Operator-Variante. Beide sind langsamer als die `concat()`-Version, die wiederum nur halb so schnell ist wie der passend initialisierte `StringBuffer`.

concat()	StringBuffer	Passend initialisierter StringBuffer	+-Operator
100%	160,9%	50,6%	154,3%

Tabelle 5.3: Wiederholtes Duplizieren und Anhängen einer Zeichenkette

Der Grund für den Geschwindigkeitsvorteil des vorinitialisierten `StringBuffers` liegt darin, dass dieser nie seine Kapazität vergrößern muss. Der nicht initialisierte `StringBuffer` hingegen muss intern ständig neue, größere `char`-Arrays anlegen und deren Inhalt hin- und herkopieren. Das liegt daran, dass `StringBuffer` genau wie `String` intern einen `char`-Array benutzt, um die einzelnen Zeichen zu speichern. Die Größe von Arrays lässt sich jedoch nicht im Nachhinein ändern. Wenn die Kapazitätsgrenze erreicht ist, muss ein neuer, größerer Array angelegt und der Inhalt des alten Arrays in den neuen kopiert werden. Der neue Array hat dabei übrigens mindestens die zweifache Länge des alten Arrays plus zwei.

Benutzen Sie also `StringBuffer` zum Anfügen von Strings, wenn Sie mehrere Strings anfügen wollen und die entsprechende Methode häufig aufgerufen wird. Initialisieren Sie außerdem den `StringBuffer` mit einer angemessenen Kapazität – voreingestellt ist 16. Wenn Sie den `StringBuffer(String s)`-Konstruktor verwenden, wird zur Länge des Strings 16 addiert und dies als initiale Länge des internen `char`-Arrays gesetzt.

Wenn Sie jedoch in einer einzelnen, nur einmal ausgeführten Zeile ein paar Strings aneinander fügen wollen, lohnt es sich nicht, explizit einen `StringBuffer` zu benutzen. Dies erledigt in der Regel der Compiler für Sie. Zudem ist das Verwenden des +-Operators wesentlich einfacher und führt zu besser lesbarem Code.

Insbesondere, wenn Sie String-Literale aneinander reihen wollen, sind `StringBuffer` eher kontraproduktiv. Kommen Sie also nicht auf die Idee, Folgendes:

```
String s = new StringBuffer(19).append("zero ").append("one ")
   .append("two ").append("three ").toString()
```

sei schneller als dies:

```
String s = "zero " + "one " + "two " + "three ";
```

Der Compiler ist in der Lage, zu erkennen, dass Sie String-Literale miteinander verketten, und führt die Verkettung bereits vor der Übersetzung durch (*Constant Folding*). In Bytecode übersetzt wird also tatsächlich:

```
String s = "zero one two three ";
```

Und das ist kaum zu optimieren. Essentiell ist jedoch, dass die Verkettung in *einer* Zeile bzw. genauer *einem* Ausdruck steht.

```
String s = "zero ";
s += "one ";
s += "two ";
s += "three ";
```

Obiger Code resultiert in folgender, offensichtlich ungünstiger Übersetzung:

```
String s = "zero ";
s = new StringBuffer(s).append("one ").toString();
s = new StringBuffer(s).append("two ").toString();
s = new StringBuffer(s).append("three ").toString();
```

Vermeiden Sie also auf mehrere Zeilen verteilte +=-Operationen.

5.3 Bedingtes Erstellen von Strings

Es liegt in der Natur von Strings, dass sie meist für Ausgaben benutzt werden. Eine prominente Anwendung ist das Schreiben von Meldungen in eine Protokolldatei. Häufig lässt sich die Anzahl oder Detailliertheit der Meldungen durch einen Parameter verändern. Dies ist das so genannte Loglevel. Hier ein Beispiel:

```
01 public class ConditionalStringManipulationDemo {
02
03    private int logLevel = 2;
04
05    public static void main(String[] args) {
06       new ConditionalStringManipulationDemo().doIt();
07    }
08
09    public void doIt() {
10       // logge das jetzige Datum mit Loglevel 1
11       log("Datum: " + new java.util.Date().toString(), 1);
12    }
13
14    public void log(String message, int logLevel) {
15       // sofern das Loglevel hoch genug ist,
16       // wird die Nachricht ausgegeben
17       if (logLevel > this.logLevel) {
18          System.out.println(message);
19       }
20    }
21 }
```

Wenn Sie das Programm starten, werden Sie keine Ausgabe sehen. Das Loglevel steht auf zwei (Zeile 2) und die Methode `log()` wird mit Loglevel eins aufgerufen (Zeile 11). Die übergebene Nachricht wird also nicht geloggt. Der entsprechende String `"Datum: " + new java.util.Date().toString()` wird aber trotzdem erstellt, da sein Wert vor der Übergabe zur `log()`-Methode berechnet wird. Obwohl das Programm also die Meldung nicht ausgibt, wird sie erstellt – und, nebenbei bemerkt, einen Datumsstring auf diese Weise zu erstellen, ist nicht gerade billig.

Besser wäre es also, wenn wir vor dem Erstellen überprüften, ob wir die Nachricht auch tatsächlich ausgeben wollen. Dies könnte mittels einer Methode `isLog()` (Zeilen 14-16) geschehen. Der Code sähe folgendermaßen aus:

```
01 public class ConditionalStringManipulationDemo2 {
02
03    private int logLevel = 2;
04
05    public static void main(String[] args) {
06       new ConditionalStringManipulationDemo2().doIt();
07    }
08
09    public void doIt() {
10       // überprüfe mit isLog(), ob überhaupt geloggt werden soll
11       if (isLog(1)) log("Datum: " + new java.util.Date());
12    }
13
14    public boolean isLog(int logLevel) {
15       return logLevel > this.logLevel;
16    }
17
18    public void log(String message) {
19       System.out.println(message);
20    }
21 }
```

Tabelle 5.4 zeigt den enormen Geschwindigkeitsunterschied zwischen beiden Varianten. Tatsächlich war es schwierig, überhaupt sinnvolle Messergebnisse zu erhalten, da der Unterschied so groß ist.

ohne isLog()	mit isLog()
100%	0,2%

Tabelle 5.4: Die Variante mit `isLog()` *ist um den Faktor 500 schneller.*

Wenn Sie können, vermeiden Sie also das Erzeugen von Strings. Ausgerechnet Protokollmeldungen zur Fehlersuche, die im Normalbetrieb noch nicht einmal ausgegeben werden, sollten nicht der Grund sein, warum Ihre Applikation zu langsam läuft.

Gängige Logging-Frameworks wie das bewährte, freie *Log4J* und das `java.util.logging`-Paket (seit JDK 1.4) verfügen über entsprechende Methoden, die testen, ob eine Nachricht überhaupt geloggt würde. Im Falle des `java.util.logging`-APIs ist dies die Methode `java.util.logging.Logger.isLoggable()`. Im Falle von Log4j heißt die Methode `org.apache.log4j.Logger.isEnabledFor()`. Zusätzlich gibt es noch spezielle Methoden für diverse Loglevel.

- Jakarta Log4J: *http://jakarta.apache.org/log4j/*

5.4 Stringvergleiche

Sicherlich haben auch Sie am Anfang Ihrer Java-Karriere schmerzlich herausfinden müssen, dass Stringvergleiche mit dem ==-Operator nicht immer zum gewünschten Ergebnis führen. Strings selben Inhalts sind leider nicht immer auch dieselben Objekte, sondern allenfalls gleich. Hier zwei Beispiele:

```
if (new StringBuffer("string").toString() == "string") {
   System.out.println("Diese Zeile wird nie ausgegeben werden.");
}
```

Die `toString()`-Methode des `StringBuffers` erstellt ein neues String-Objekt, das zwar semantisch gleich, jedoch nicht identisch mit dem Literal `"string"` ist. Ebenso verhält es sich im folgenden Beispiel:

```
if (new String("string") == "string") {
   System.out.println("Diese Zeile wird nie ausgegeben werden.");
}
```

Der hier verwendete String-Konstruktor erzeugt ein neues String-Objekt, das wiederum inhaltlich dem String-Literal gleicht, jedoch nicht mit ihm identisch ist. Ein solcher Konstruktor wird auch *kopierender Konstruktor* (*Copy Constructor*) genannt, da er eine Kopie des Objekts anlegt. Im Fall von String-Objekten ist dies gewöhnlich reine Ressourcenverschwendung. Statt eines Objekts haben Sie auf einmal zwei, die sich zudem nur höchst ineffizient miteinander vergleichen lassen; denn gewöhnlich läuft der Vergleich von Strings über die `equals()`-Methode. Sie überprüft zunächst, ob es sich um dieselben Instanzen handelt (Objekt-Identität), dann, ob die Länge die gleiche ist, und schließlich, ob alle Zeichen gleich sind (semantische Gleichheit). Offensichtlich ist der Vergleich am schnellsten, wenn Sie identische Objekte miteinander vergleichen (konstante Laufzeit – zur Klassifizierung von Algorithmen siehe auch *Kapitel 8.1 Groß-O-Notation*), und am langsamsten, wenn Sie inhaltlich gleiche, aber nicht identische Strings vergleichen (lineare Laufzeit). Genau Letzteres trifft für Objekte zu, die mit dem kopierenden Konstruktor erstellt wurden:

```
// Tun Sie dies nicht! Lineare Laufzeit!
new String("string").equals("string");
```

Zeichenweise Stringvergleiche und somit lineare Laufzeit sind jedoch häufig, wenn Sie Strings erst zur Laufzeit konstruieren. Dieser aufwändige Vergleich lässt sich vermeiden, da jede Java VM einen Stringkonstantenpool unterhält, in dem jeweils eine eindeutige Instanz eines Strings gespeichert ist.

Es befinden sich garantiert alle String-Literale im Konstantenpool. Daher gilt immer:

```
String a = "string";
String b = "string";
if (a==b) {
   // Wird garantiert ausgegeben:
   System.out.println("Die Objekte sind identisch.");
}
```

Praktisch ist diese Tatsache zum Beispiel in der Swing-Programmierung:

```
public SwingExample implements ActionListener {

...

   public void setUpMenuBar() {
      ...
      JMenuItem item = new JMenuItem("Open");
      // setzt das eindeutige Literal "open" als Kommando
      item.setActionCommand("open");
      item.addActionListener(this);
      ...
   }

   // Wird aufgerufen, wenn der Menüeintrag "Open" angeklickt wird
   public void actionPerformed(ActionEvent e) {
      // der Vergleich ist korrekt, da wir mit dem Literal
      // und somit mit derselben Instanz vergleichen
      if (e.getActionCommand() == "open") {
         ...
      }
   }
}
```

Wenn Sie nicht nur mit Literalen arbeiten, können Sie die `intern()`-Methode des String-Objekts verwenden, um eine Referenz auf die eindeutige String-Instanz aus dem Pool zu erlangen. Beispiel:

```
String a = "string";
String b = new String("string");
if (a==b) {
   System.out.println("Diese Zeile wird nie ausgegeben.");
}
if (a==b.intern()) {
   System.out.println("a und b.intern() sind identisch.");
}
```

Nun werden Sie sich vermutlich fragen, warum man nicht immer mittels `intern()`auf Objektidentität vergleicht, anstatt die `equals()`-Methode zu verwenden. Oder, warum die `equals()`-Methode nicht `intern()` verwendet. Nun, `intern()` hat zwar konstante Laufzeit[1], diese ist aber meist höher als die im schlechtesten Fall lineare Laufzeit von `equals()`. Wir wollen verschiedene Fälle vergleichen. Hier Fragmente des entsprechenden Testcodes:

```
private String a = "0123456789";
private String b = new String("0123456789");
private String c = "9012345678";
private String d = "d";
// Alle Schleifen werden vielfach ausgeführt, dabei wird
// innerIterations von 1 bis 10 gesteigert
private int innerIterations;
...
// Gleiche, nicht-identische Strings
for (int i = 0; i < innerIterations; i++) a.equals(b);

// Verschiedene Strings gleicher Länge
for (int i = 0; i < innerIterations; i++) a.equals(c);

// Unterschiedliche Länge
for (int i = 0; i < innerIterations; i++) a.equals(d);

// Gleiche Strings verglichen mit intern()
String e = b.intern();
for (int i = 0; i < innerIterations; i++) a.equals(e);
```

Abbildung 5.1 zeigt die relative Laufzeit der verschiedenen Stringvergleiche. Es wurden pro Reihe jeweils bis zu zehn Vergleiche durchgeführt. Bei dem Vergleich mit `intern()` wurde jedoch pro Reihe nur einmal `intern()` aufgerufen. Wie nicht anders zu erwarten steigen alle Werte nahezu linear mit der Anzahl der Vergleiche. Jedoch sind die Steigungen (Tabelle 5.5) sehr unterschiedlich.

	Unterschiedliche Länge	**Verschiedene Strings gleicher Länge**	**Gleiche Strings mit intern()**	**Gleiche, nichtidentische Strings**
Steigung	2,3	6,7	1,6	28,4
Rechnerischer Schnittpunkt mit der `intern()`-Variante	134,8	18,7	-	3,6

Tabelle 5.5: Steigung der Ausführungszeit in Abhängigkeit von der Anzahl der Vergleiche sowie die Mindestanzahl an Vergleichen, die zu einer längeren Laufzeit führen als die intern()-Variante.

1 Zumindest ist das für die Sun JDKs der Fall.

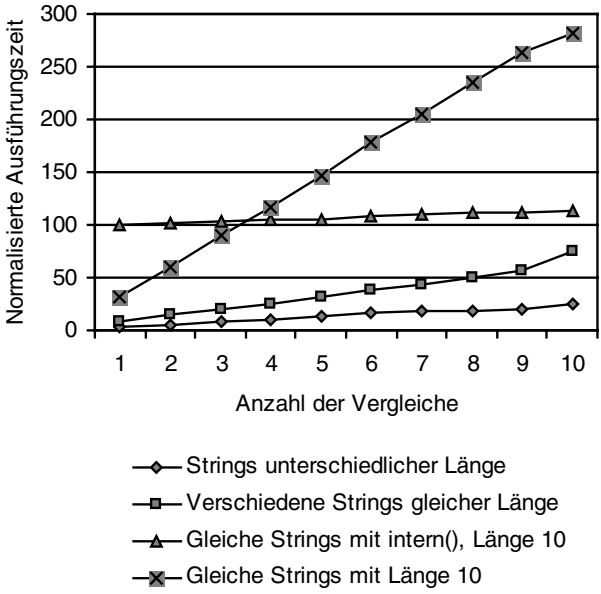

Abbildung 5.1: Kosten von Stringvergleichen verschiedener bzw. gleicher, aber nicht identischer Strings mit und ohne intern()

Aus den Daten folgt, dass sich der relativ teure Aufruf von `intern()` lohnt, wenn Sie Strings pro `intern()`-Aufruf häufig genug vergleichen und unter den verglichenen Strings möglichst viele gleiche, nicht-identische Strings sind. Wenn Sie jedoch nicht besonders häufig vergleichen und die verglichenen Strings meist auch noch unterschiedliche Längen haben, lohnt sich `intern()` eher nicht.

Bedenken Sie zudem, dass die Ausführungsgeschwindigkeit von `intern()` stark von der Implementierung der VM abhängt, da es sich um eine `native` Methode handelt. Die oben genannten Werte haben also lediglich Beispielcharakter.

5.5 Groß- und Kleinschreibung

Etwas aufwändiger ist der Stringvergleich, wenn Sie Groß- und Kleinschreibung ignorieren wollen. Grundsätzlich gibt es dazu drei Strategien:

- Sie benutzen die String-Methode `equalsIgnoreCase()`.
- Sie konvertieren beide Strings mittels `toLowerCase()` oder `toUpperCase()` in Groß- bzw. Kleinbuchstaben und vergleichen mit `equals()`.
- Sie benutzen einen `java.text.Collator` bzw. `java.text.CollationsKeys` zum Vergleich.

5.5.1 Vergleich mittels equalsIgnoreCase()

Wir wollen zunächst `equalsIgnoreCase()` betrachten. In Suns JDK 1.3.1 und IBMs JDK 1.3.0 vergleicht die Methode zunächst mit `null`, überprüft dann, ob die Länge gleich ist, und vergleicht anschließend, ob alle Buchstaben entweder auf Anhieb dieselben sind oder aber zumindest ihre Großbuchstaben sich gleichen.[2]

Seit JDK 1.4 wird vor den oben genannten Tests zunächst auf Objektidentität geprüft. Das hatte Sun offensichtlich vorher vergessen. Was also für IBM JDK 1.3.0 und Sun JDK 1.3.1 der schlechteste Fall war, ist für JDK 1.4 der beste (Tabelle 5.6).

Sun JDK 1.3.1 Client	IBM JDK 1.3.0	Sun JDK 1.4.0 Client
100%	83%	7,7%

Tabelle 5.6: Vergleich zweier identischer Strings der Länge zehn mit equalsIgnoreCase()

Nun darf man nicht vergessen, dass der Fall zweier identischer Strings nicht unbedingt der häufigste ist. Wenn Sie jedoch `equalsIgnoreCase()` häufig verwenden, die Chance, dass Sie identische Objekte vergleichen, hoch ist und Sie noch JDK 1.3.x benutzen, lohnt es sich, evtl. folgenden Code zu benutzen:

```
// s1 und s2 sind jeweils Stringobjekte
// s1 ist zudem nicht null
if (s1 == s2 || s1.equalsIgnoreCase(s2)) {
   System.out.println("s1 und s2 sind gleich.");
}
```

Verwenden Sie diesen Code nicht, wenn die genannten Bedingungen nicht zutreffen. Sie würden lediglich Ihren eigenen Code verschmutzen.

5.5.2 toLowerCase() oder toUpperCase(), das ist hier die Frage

Das oben bereits erwähnte zweite Verfahren, Strings unabhängig von Groß- und Kleinschreibung zu vergleichen, ist nicht unbedingt zu empfehlen. Die beiden String-Methoden `toLowerCase()` und `toUpperCase()` erzeugen jeweils ein neues String-Objekt, es sei denn der String enthält nur Klein- bzw. Großbuchstaben.[3] Pro Vergleich müssen also meist zwei neue Objekte erzeugt werden, was zu einem hohen Aufwand führt. Vom folgenden Code ist somit unbedingt abzuraten – `equalsIgnoreCase()` leistet wesentlich bessere Dienste.

2 Für den Fall, dass die Großbuchstaben nicht dieselben sind, werden außerdem noch die Kleinbuchstaben überprüft, da im georgischen Alphabet kleingeschriebene Buchstaben gleich sein können, obwohl ihre jeweiligen Großbuchstaben dies nicht sind.

3 Um dies festzustellen, wird jedes Mal eine lineare Suche nach einen Groß- oder Kleinbuchstaben durchgeführt. Das Ergebnis dieser Suche wird nicht gecached.

```
String a = "a und B";
String b = "B und a";
// sehr teurer Vergleich!
if (a.toLowerCase().equals(b.toLowerCase())) {
   System.out.println("Die beiden Strings sind gleich.");
}
```

Die beiden Methoden können dennoch nützlich für Vergleiche sein. Beispielsweise muss in *HTTP* (*Hypertext Transfer Protokoll*) die Groß- und Kleinschreibung von Headernamen ignoriert werden. Zu diesem Zweck ist eine Hashtabelle nützlich, die die Groß- und Kleinschreibung von Schlüsseln ignoriert. Mit `toLowerCase()` bzw. `toUpperCase()` können Sie vor jedem Einfügen oder Entnehmen den Schlüssel in seine klein- (oder groß-)geschriebene Form bringen.

```
Map map = new HashMap();

public Object get(String key) {
   return map.get(key.toLowercase());
}

public Object put(String key, Object object) {
   return map.put(key.toLowerCase());
}
...
```

Nun unterliegt die Groß- und Kleinschreibung gewissen Regeln. Es ist zum Beispiel sehr viel wahrscheinlicher, dass Sie Java im Deutschen mit großem *J* und kleinem *ava* schreiben und nicht *jAVa*. Genauso wird der HTTP *Accept*-Header meistens mit großem *A* und ansonsten klein geschrieben. Diese Tatsache können Sie sich zunutze machen, indem Sie zwei statt einer Hashmap benutzen. In der ersten Hashmap legen Sie das Objekt unter dem Original-Schlüssel ab, in der zweiten unter dem kleingeschriebenen:

```
public class CaseInsensitiveMap {

   Map map = new HashMap();
   Map lowerCaseMap = new HashMap();

   public Object get(String key) {
      Object value = map.get(key);
      return value != null ?
         value : lowerCaseMap.get(key.toLowerCase());
   }

   public Object put(String key, Object value) {
      // dies ist nicht ausreichend!
      Object oldValue = map.put(key, value);
      return key != null ?
```

```
            lowerCaseMap.put(key.toLowerCase(), value) : oldValue;
    }
...
}
```

Leider ist der Code nicht ganz so einfach, wie oben beschrieben. Betrachten Sie folgenden Fall:

```
CaseInsensitiveMap map = new CaseInsensitiveMap();
map.put("java", "erster Eintrag");
map.put("Java", "zweiter Eintrag");
System.out.println(map.get("java"));
```

Dies führt zu dieser unerwünschten Ausgabe:

```
erster Eintrag
```

Der erste Eintrag ist lediglich aus der `lowerCaseMap` entfernt worden, nicht jedoch aus der normalen `map`. Wir müssen uns also ein wenig mehr Mühe geben. Der folgende Code löst das Problem, wenngleich um den Preis von ein bisschen mehr Komplexität bei den Operationen Entfernen und Hinzufügen sowie einer zusätzlichen Datenstruktur und somit größerem Speicherverbrauch.

```
...
// Abbildung von lowerCase-Schlüsseln auf andere Schreibweisen
Map equivalentKeys = new HashMap();

public Object remove(String key) {
   Object oldValue = null;
   if (key != null) {
      String lowerCaseKey = key.toLowerCase();
      oldValue = lowerCaseMap.remove(lowerCaseKey);
      Set s = (Set)equivalentKeys.get(lowerCaseKey);
      if (s != null) {
         // Falls der Wert auch unter anderen Schlüsseln
         // hinterlegt war, müssen wir die entsprechenden
         // Einträge aus der map entfernen.
         for (Iterator i = s.iterator(); i.hasNext(); ) {
            map.remove(i.next());
         }
      }
   }
   else {
      oldValue = lowerCaseMap.remove(null);
      map.remove(null);
   }
   return oldValue;
}

public Object put(String key, Object value) {
```

```
    Object oldValue = null;
    if (key != null) {
       String lowerCaseKey = key.toLowerCase();
       oldValue = lowerCaseMap.get(lowerCaseKey);
       if (oldValue != value) {
          remove(key);
          lowerCaseMap.put(lowerCaseKey, value);
       }
       map.put(key, value);
       // Damit wir in konstanter Zeit entfernen
       // können, merken wir uns alle verschiedenen
       // Schreibweisen des Schlüssels.
       Set set = (Set)equivalentKeys.get(lowerCaseKey);
       if (set == null) {
          set = new HashSet();
          equivalentKeys.put(lowerCaseKey, set);
       }
       set.add(key);
    }
    else {
       oldValue = lowerCaseMap.put(null, value);
       map.put(null, value);
    }
    return oldValue;
 }
 ...
```

Um alle verschiedenen Schreibweisen eines Schlüssels aus der `map` entfernen zu können, merken wir uns diese in einem Set, das wir unter der kleingeschriebenen Version des Schlüssels in einer zusätzlichen Map namens `equivalentKeys` hinterlegen. Auf diese Weise können wir alle zu einem Eintrag in `lowerCaseMap` äquivalenten Einträge zuverlässig auch aus der `map` entfernen. Dies ist für die Integrität der Datenstruktur unerlässlich.

In unserem Beispiel konvertieren wir die Schlüssel der `lowerCaseMap` in ihre kleingeschriebene Form. Wenn Sie dies tun, müssen Sie sich bewusst sein, dass der Buchstabe *ß* erhalten bleibt. Sie können also mit dem Schlüssel *gemäß* nicht denselben Wert finden wie mit dem Schlüssel *GEMÄSS*. Anders verhält es sich, wenn Sie statt der `lowerCaseMap` eine entsprechende `upperCaseMap` verwenden und alle Schlüssel mittels `toUpperCase()` konvertieren. Das Wort *gemäß* würde von `toUpperCase()` zu *GEMÄSS* umgewandelt. Somit könnten Sie mit den Schlüsseln *gemäß* und *GEMÄSS* dieselben Einträge finden.

Alternativ zu der beschriebenen Implementierung mittels `toLowerCase()` oder `toUpperCase()` können Sie als Datenstruktur auch eine `SortedMap` mit einem Groß-/Kleinschreibung ignorierenden `java.util.Comparator` verwenden. `String.CASE_INSENSITIVE_ORDER` ist ein solcher `Comparator`. Anstatt also eine eigene Klasse zu implementieren, können Sie einfach folgenden Code verwenden:

```
SortedMap map = new TreeMap(String.CASE_INSENSITIVE_ORDER);
```

Wie wir noch sehen werden, ist dieser Code jedoch sehr viel langsamer als die oben beschriebene Implementierung.

5.5.3 Wenn *Ä* gleich *a* sein soll

Im letzten Abschnitt ist bereits angeklungen, dass String-Vergleiche nicht immer ganz so einfach sind, wie sie scheinen. Im Deutschen sind insbesondere *ß* und Umlaute etwas kompliziert zu handhaben, im Französischen hat man mit den Accents so seine Schwierigkeiten.

Java versucht diese Schwierigkeiten mit der `java.text.Collator`-Klasse abzudecken. Ein Kollator ist – frei übersetzt – ein spezieller Text-Vergleicher.[4] Kollatoren gibt es für verschiedene Sprachen und in unterschiedlicher Stärke. Gewöhnlich erhalten Sie den gewünschten Kollator von der Fabrikmethode `Collator.getInstance()`:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY); // Unterscheidungsgrad
if (collator.equals("gross", "Groß")) {
   // Diese Zeile wird ausgegeben, da die beiden Strings nur
   // Unterschiede dritter Ordnung haben.
   System.out.println("gross und Groß sind gleich.");
}
```

Alternative Schreibweisen	Unterscheidungsgrad (Stärke)
Umlaute oder entsprechende Buchstaben ohne ¨. Beispiel: *ü* und *u*	primär
Groß-/Kleinschreibung. Beispiel: *G* und *g*	sekundär
Doppel-S oder S-Zett. Beispiel: *ss* und *ß*	sekundär
Verschiedene Buchstaben. Beispiel: *q* und *z*	grundsätzlich nicht gleich
Umlaute oder entsprechende Buchstaben mit angehängtem e. Beispiel: *ä* und *ae*	grundsätzlich nicht gleich

Tabelle 5.7: Alternative Schreibweisen von Buchstaben im Deutschen und ihr Unterscheidungsgrad gemäß dem deutschen Collator des Sun JDKs

Wenn Sie nur Groß- und Kleinschreibung sowie Doppel-S und S-Zett ignorieren wollen, müssen Sie als Unterscheidungsgrad `Collator.SECONDARY` setzen (Tabelle 5.7). Wenn Sie darüber hinaus auch noch die Umlaut-Punkte ignorieren wollen, müssen Sie als Unterscheidungsgrad `Collator.PRIMARY` setzen. Es gibt leider keinen Grad, der beispielsweise *ä* mit *ae* gleichsetzt.[5]

4 Kollation (lat.): Vergleich einer Abschrift mit der Urschrift zur Prüfung der Richtigkeit.

5 Sie können jedoch einen entsprechenden eigenen Regelsatz für den `java.text.RuleBasedCollator` schreiben.

Da die Collator-Klasse die Schnittstelle `Comparator` implementiert, kann man mittels einer Collator-Instanz und einer `SortedMap` eine ähnliche Datenstruktur aufbauen wie die oben beschriebene `CaseInsensitiveMap`. Als `Comparator` muss lediglich der entsprechende Collator gesetzt werden:

```
Collator collator = Collator.getInstance(Locale.GERMAN);
collator.setStrength(Collator.SECONDARY);
SortedMap map = new TreeMap(collator);
```

Performanter ist es jedoch, wenn Sie `java.util.CollationKey`s an Stelle von Strings als Schlüssel einsetzen. `CollationKeys` sind String-Wrapper, die sich effizient vergleichen lassen – und zwar nach den Regeln des Kollators, der sie erzeugt hat. In der Praxis sieht das wie folgt aus:

```
public class CollationKeyMap {
   private Map map = new HashMap();
   private Collator collator;

   public CollationKeyMap(Locale locale, int collatorStrength) {
      collator = Collator.getInstance(
         locale == null ? Locale.getDefault() : locale
      );
      collator.setStrength(collatorStrength);
   }

   public Object get(String key) {
      if (key == null) return map.get(null);
      return map.get(collator.getCollationKey(key));
   }

   public Object put(String key, Object value) {
      if (key == null) return map.put(null, value);
      return map.put(collator.getCollationKey(key), value);
   }
   ...
}
```

Nachdem ich nun so viele alternative Implementierungen für eine Map vorgestellt habe, die die Groß-/Kleinschreibung der Schlüssel ignoriert, möchte ich diese noch kurz vergleichen. Als Test füge ich zunächst mittels `put()` 1804 aus einem deutschen Text extrahierte Wörter[6] in eine der Maps und lese sie danach mittels `get()` wieder aus. Dabei wird eine Kopie des Schlüssels benutzt und so der unwahrscheinliche Fall der Objektidentität vermieden. Gemessen wird nur das Auslesen.

6 173 der verwendeten Wörter unterschieden sich nur bezüglich Groß-/Kleinschreibung.

Anschließend wiederhole ich den Test, wobei ich diesmal großgeschriebene Schlüssel zum Einfügen benutze und kleingeschriebene Schlüssel zum Auslesen. Im dritten Durchgang teste ich genau andersherum – kleingeschriebene Schlüssel zum Einfügen und großgeschriebene Schlüssel zum Auslesen.

Um einen Vergleich zum Auslesen aus einer Hashtabelle zu haben, die Groß-/Kleinschreibung nicht ignoriert, führe ich den Test außerdem mit einer normalen `HashMap` durch und lese die Daten jeweils mit einer Kopie des Einfügeschlüssels aus.

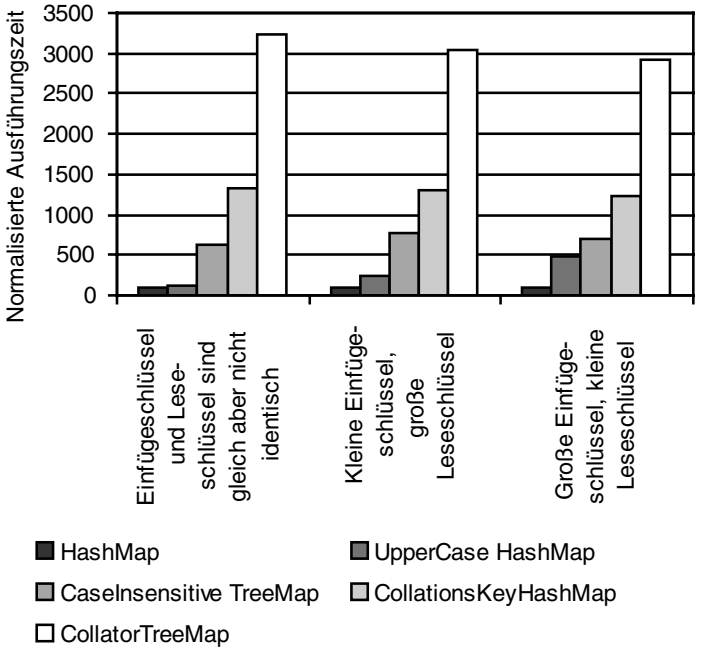

Abbildung 5.2: Geschwindigkeit des lesenden Zugriffs auf verschiedenen Datenstrukturen mit natürlichen, groß- oder kleingeschriebenen Schlüsseln

Das Ergebnis (Abbildung 5.2) lässt keine Fragen offen. Die UpperCase-HashMap ist allen anderen Tabellen überlegen und erreicht im günstigsten Fall (Einfügeschlüssel gleich Leseschlüssel) beinahe die Geschwindigkeit der normalen `HashMap`. Bei großgeschriebenen Ausleseschlüsseln verdoppelt sich die Ausführungszeit der UpperCase-HashMap, bei kleingeschriebenen Schlüsseln verdoppelt sie sich nochmals. Das ist darauf zurückzuführen, dass die Methode `toUpperCase()` eines bereits großgeschriebenen Strings wesentlich schneller ist als `toUpperCase()` eines kleingeschriebenen Strings, da einfach `this` zurückgegeben werden kann und kein neues Objekt erzeugt werden muss.

Die nächstschnellste Datenstruktur ist die `TreeMap` mit `String.CASE_INSENSITIVE_ORDER` als Vergleichsobjekt. Auf den Plätzen drei und vier landen die `HashMap` mit `CollationsKeys` und die `TreeMap` mit einem `Collator` als Vergleichsobjekt. In beiden Fällen wurde `Collator.SECONDARY` als Unterscheidungsgrad gesetzt.

5.6 Strings sortieren

Ein verwandtes Thema zum Groß-Klein-Vergleich ist das Sortieren. Auch hier müssen Strings möglichst effizient miteinander verglichen werden. Zusätzlich zu der Information gleich oder ungleich ist hier noch gefragt, ob ein String größer oder kleiner als ein anderer ist.

Üblicherweise werden zum Sortieren die Klassenmethoden `java.util.Arrays.sort()` oder `java.util.Collections.sort()` benutzt. Intern wird dabei die Methode `compareTo()` der Strings oder eines entsprechenden `Comparators` verwendet. Wenn Sie `compareTo()` benutzen, wird die *natürliche* Reihenfolge verwendet, wie sie im Unicode-Zeichensatz definiert ist. Das bedeutet, dass die Worte *Arbeit*, *Änderung*, *Zug* und *Andenken* in die Reihenfolge *Andenken*, *Arbeit*, *Zug* und *Änderung* sortiert werden, da der Buchstabe *Ä* im Unicode-Zeichensatz erst nach allen anderen lateinischen Buchstaben steht. Ebenso tauchen kleingeschriebene Wörter erst nach sämtlichen großgeschriebenen Wörtern auf. Das letztere Problem lässt sich leicht lösen, indem Sie `String.CASE_INSENSITIVE_ORDER` als Vergleichsobjekt angeben. Für die korrekte Lösung des Umlaut-Problems müssen Sie einen Kollator als `Comparator` übergeben. Am schnellsten jedoch wird korrekt sortiert, wenn Sie zunächst in der natürlichen Reihenfolge und anschließend erst mit einem Kollator sortieren lassen, da sich eine teilweise sortierte Liste in der Regel schneller sortieren lässt als eine völlig ungeordnete Liste.

natürliche Ordnung	CASE_INSENSITIVE_ ORDER	Kollator	CASE_INSENSITIVE_ ORDER + Kollator	natürlich + Kollator
100%	222%	926%	499%	359%

Tabelle 5.8: Sortieren eines String-Arrays mit verschiedenen Methoden

Tabelle 5.8 belegt diesen Rat mit Zahlen. Das Vorsortieren eines String-Arrays gemäß seiner natürlichen Ordnung (natürlich + Kollator: 359%) führt im Beispiel zu einem Geschwindigkeitsvorteil um den Faktor 2,5 gegenüber dem ausschließlichen Sortieren gemäß der Ordnung eines Kollators (Kollator: 926%) [vgl. Shirazi00, S.156f]. Benutzt wurden die gleichen Wörter wie schon im Abschnitt zuvor.

5.7 Formatieren

Nicht selten müssen Objekte in eine String-Darstellung überführt werden. Seien es Zahlen, Daten oder Beträge – immer wird ein String erstellt, der oft auch noch den örtlichen Gepflogenheiten entsprechen soll. Da das Erstellen selbst schon eine aufwändige Angelegenheit ist, macht es die Internationalisierung nicht gerade besser. Wir wollen an einem einfachen Beispiel illustrieren, wie hoch die Kosten der Stringerzeugung sind, indem wir das Erstellen des Strings für eine einfache ganze Zahl messen.

In den Startblöcken für den Testlauf stehen drei Kandidaten: `Integer.toString()`, `Long.toString()` und `NumberFormat.format()`.[7] Gemessen wurde jeweils folgende Schleife:

```
for (int i=0; i<1000; i++) Integer.toString(i);
// bzw. Long.toString(i) oder formatter.format(i)
```

Integer.toString()	Long.toString()	numberFormatter.format()
100%	609%	978%

Tabelle 5.9: Erzeugen eines Strings für einen `int` *auf verschiedene Weisen*

Wie Tabelle 5.9 zu entnehmen ist, schneidet `Integer.toString()` wesentlich besser ab als die beiden anderen Kandidaten. `Long.toString()` ist sechsmal, `NumberFormat` sogar fast zehnmal langsamer.

Einigermaßen verwunderlich ist der Unterschied zwischen der `Integer`- und der `Long`-Variante. Schließlich handelt es sich nicht um fundamental verschiedene Aufgaben. Ein Blick in den JDK-Quellcode offenbart jedoch, dass `Integer.toString(int)` besonders optimiert wurde, während `Long.toString(long)` lediglich die allgemeinere Variante mit beliebiger Basis `Long.toString(long, basis)` aufruft. Kein Wunder also, dass die `Integer`-Variante so viel schneller ist.

Der große Unterschied zwischen `Integer.toString()` und `NumberFormat` ist mindestens ebenso eindrucksvoll, aber nicht weiter verwunderlich. `Integer.toString()` ist ausschließlich für die gestellte Aufgabe geschrieben, während `NumberFormat` sehr viel flexibler ist. So kann man beispielsweise die Anzahl der Nach- und Vorkomma-Stellen spezifizieren, was mit `Integer.toString()` nicht möglich ist.

Grundsätzlich gilt: Flexibilität kostet. Keine der Formatierer-Klassen im `java.text`-Paket ist in der Lage, eine spezialisierte Implementierung in Punkto Geschwindigkeit zu schlagen. Stattdessen bieten sie Flexibilität, Unterstützung für Internationalisierung, Wartbarkeit und die Fähigkeit ausgegebenen Text mittels der `parseObject()`-Methode auch wieder einzulesen. Dessen sollten Sie sich unbedingt bewusst sein,

7 Es wurde eine Instanz der Klasse `NumberFormat` erstellt, die wieder verwendet wurde.

bevor Sie beginnen, aus Geschwindigkeitsgründen einen speziellen Formatierer zu implementieren. Und wenn Sie es doch tun, versuchen Sie von `java.text.Format` oder einer der Unterklassen zu erben. Auf diese Weise bewahren Sie wenigstens Schnittstellenkompatibilität.

5.7.1 Nachrichten erstellen

Um formatierte Nachrichten auszugeben, stellt das JDK die Klasse `java.text.MessageFormat` zur Verfügung. `MessageFormat` lässt sich über eine statische `format()`-Methode oder als Instanz benutzen. Wir wollen beide Varianten mit einer simplen selbst gestrickten Alternative vergleichen. Hier die getesteten Code-Stücke:

```
// Variante mit statischer format()-Methode
public static void formattedMessage() {
   String message = MessageFormat.format("On {1,time, long} "
      + "{1,date, long}, there was {2} on planet "
      + "{0,number,integer}.",
      new Object[] {new Integer(1), new Date(),
      "a disturbance in the Force"});
}

// Variante mit MessageFormat-Instanz
private static MessageFormat formatter = new MessageFormat(
   "On {1,time, long} {1,date, long}, there was {2} on planet "
   + "{0,number,integer}.");
public static void preFormattedMessage() {
   String message = formatter.format(new Object[]
      {new Integer(1), new Date(),
      "a disturbance in the Force"});
}

// selbst gestrickte Variante
public static void plainMessage() {
   String what = "a disturbance in the Force";
   Date date = new Date();
   int number = 1;
   String message = "On " + date + ", there was " + what
      + " on planet " + number;
}
```

Und hier die beiden (leicht verschiedenen) Ausgaben:

```
(pre)formattedMessage():
On 18:21:33 EST 22. Februar 2002, there was a disturbance in the Force on planet
1.
plainMessage():
On Fri Feb 22 18:21:33 EST 2002, there was a disturbance in the Force on planet 1
```

plainMessage()	formattedMessage()	preFormattedMessage()
100%	521%	129%

Tabelle 5.10: Vergleich verschiedener Nachrichtenformate

Wie nicht anders zu erwarten, ist `plainMessage()` um einiges schneller als die beiden Kontrahenten. Jedoch beträgt der Unterschied zur `preFormattedMessage()`-Variante lediglich 29%. Im Gegensatz dazu ist der Unterschied zur `formattedMessage()`-Version größer als der Faktor fünf. Dies rührt daher, dass intern jeweils ein neues `MessageFormat`-Objekt angelegt und der Muster-String jedes Mal neu analysiert wird. Diesen Aufwand haben wir uns durch Benutzen der Instanz erspart. Wenn Sie also `MessageFormat` benutzen wollen, versuchen Sie das Formatierer-Objekt wiederzubenutzen. Ziehen Sie dabei schwache Referenzen (`java.lang.ref.SoftReference`) zum Halten der Instanz in Betracht, da zu viele langlebige Objekte die automatische Speicherbereinigung ausbremsen (siehe *Kapitel 3.2 Garbage Collection*).

5.7.2 Datum und Zeit

Datum- und Zeit-Strings gehören sicherlich zu den am häufigsten ausgegebenen. Besonders prominent sind hier wiederum Protokolldateien, die üblicherweise einen Zeitstempel am Anfang der Zeile enthalten. Vor nicht allzu langer Zeit arbeitete ich in einem Projekt, in dem dieser Zeitstempel folgendermaßen erstellt wurde:

```
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.Calendar;

...

// Kopieren Sie diesen Code nicht!
Calendar calendar = Calendar.getInstance();
Date date = calendar.getTime();
SimpleDateFormat dateFormat = new SimpleDateFormat(
   "dd MMM yyyy HH:mm:ss,SSS"
);
String dateString = dateFormat.format(date);
```

Diese vier Zeilen wurden jedes Mal ausgeführt, wenn etwas zu protokollieren war. Der Code ist korrekt, er könnte jedoch etwas performanter sein. Lassen Sie uns ein paar Vergleiche anstellen. Hier drei alternative Implementierungen:

```
// Einfache Date-Variante
String dateString = new Date().toString();

// Vorinitialisierter SimpleDateFormatter
private static DateFormat simpleDateFormat =
```

```
    new SimpleDateFormat("dd MMM yyyy HH:mm:ss,SSS");
String dateString = simpleDateFormat.format(new Date());

// Vorinitialisierter Log4j DateTimeDateFormatter
private static DateFormat log4JdateTimeDateFormat =
    new DateTimeDateFormat();
String dateString = log4JdateTimeDateFormat.format(new Date());
```

Die Klasse `org.apache.log4j.helpers.DateTimeDateFormat` ist ein speziell für das angegebene Format optimierter Formatierer aus dem oben bereits erwähnten freien Log-Framework Log4J. Sie erbt von `java.text.DateFormat` und lässt sich entsprechend benutzen.[8]

Die Ausgabe der vier Varianten ist wiederum leicht unterschiedlich. Insbesondere werden in der einfachen `date.toString()`-Variante der Wochentag und die Zeitzone ausgegeben, während die Millisekunden fehlen.

```
// Einfache Date-Variante
Sat Feb 23 10:29:30 EST 2002

// Alle anderen Varianten
23 Feb 2002 10:29:30,751
```

Zugegeben, das ist ein wenig wie Äpfel mit Birnen vergleichen. Wir lassen die `Date.toString()`-Variante daher außer Konkurrenz antreten und benutzen sie lediglich als Referenzwert, da sie so schön einfach zu programmieren ist. In Tabelle 5.11 sehen Sie das Ergebnis.

date.toString()	Log4J	SimpleDateFormat	Vierzeiler
100%	30%	97%	277%

Tabelle 5.11: Normalisierte Geschwindigkeit verschiedener Code-Varianten, die einen formatierten Datumsstring erzeugen

Die Log4J-Variante ist um den Faktor drei schneller als `SimpleDateFormat`. Dieses wiederum ist beinahe um den Faktor drei schneller als der oben beschriebene Vierzeiler.

Eine Analyse der Hprof-Ausgabe ergibt, dass rund 60% der Ausführungszeit des Vierzeilers im Konstruktor von `SimpleDateFormat` verbracht wird. So lange dauert es, den Format-String zu verarbeiten. Das eigentliche Formatieren schlägt lediglich mit 30% zu Buche und der `Calendar.getInstance()`-Aufruf ist zwar nicht gerade schlau, aber im Endeffekt mit etwa 10% Anteil an der Ausführungszeit nicht so bedeutend.

8 Im JDK 1.4 `java.util.logging`-Paket scheint es (noch) keine entsprechend performanten Datumsformatierer zu geben.

Dies bestätigt die bereits weiter oben gemachte Erfahrung, dass das Initialisieren von `java.text.Format`-Objekten recht teuer ist und es sich daher lohnt, diese Objekte wieder zu verwenden.

Und was macht die Log4J-Klasse so schnell? Zum einen ist sie auf die Ausgabe von Datumsstrings in dem verwendeten Format spezialisiert. Sie muss sich also nicht wie `SimpleDateFormat` an Muster halten, sondern ist hart und somit effizient kodiert. Zum anderen offenbart ein Blick in den Quellcode, dass die Log4J-Klasse Teilergebnisse zwischenspeichert. Dabei macht sie sich die Tatsache zu nutze, dass sich nur die letzten drei Zeichen (nämlich die Millisekunden) des Strings ändern, wenn die `format()`-Methode innerhalb einer Sekunde mehrmals aufgerufen wird. In der `format()`-Methode wird also getestet, ob eine Sekundengrenze überschritten wurde. Ist dies nicht der Fall, wird lediglich der neue Millisekunden-Wert an den zwischengespeicherten Anfang gehängt. So wird ein Großteil des Aufwands gespart, der zum Erstellen eines komplett neuen Strings erforderlich wäre.

In unserem Test ist dieser Cache-Effekt natürlich voll zum Tragen gekommen, da wir immer wieder neue, aufeinander folgende `Date`-Objekte erzeugt haben. In einer echten Applikation wäre der Geschwindigkeitsvorteil sicherlich nicht so groß.

Wir haben gesehen, dass der Log4J-Formatierer vergleichsweise schnell ist. Wir können jedoch auch in einer echten Anwendung noch schneller einen Datumsstring erzeugen als mit dem Log4J-Formatierer. Dies gelingt, indem wir beim ersten Datumsstring einfach die Millisekunden seit 1970 und bei jedem folgenden Mal die Differenz zu diesem ersten Wert ausgeben. Die Differenz fällt meistens in den Wertebereich eines `ints` und kann daher mit der schnelleren Methode `Integer.toString()` statt `Long.toString()` ausgegeben werden. Zudem ergibt die Differenz in den meisten Fällen einen kürzeren String als der volle Wert. Und falls tatsächlich einmal lesbare Datumswerte benötigt werden, kann ein Werkzeug diese leicht erzeugen. Der Trick besteht darin, dies nur bei Bedarf und nicht zur Laufzeit zu tun. Listing 5.1 zeigt die Beispiel-Implementierung einer entsprechenden Klasse.

```
package com.tagtraum.perf.strings;

import java.text.DateFormat;
import java.text.FieldPosition;
import java.text.ParsePosition;
import java.util.Date;

public class DifferenceDateFormat extends DateFormat {

   private long baseTime;

   public DifferenceDateFormat() {}

   public synchronized StringBuffer format(Date date,
```

```
            StringBuffer toAppendTo, FieldPosition fieldPosition) {
        if (baseTime == 0) {
            baseTime = date.getTime();
            toAppendTo.append(Long.toString(baseTime));
        } else {
            long diff = date.getTime() - baseTime;
            // wir benutzen Integer.toString(), falls möglich
            if (diff <= Integer.MAX_VALUE
                  && diff >= Integer.MIN_VALUE) {
               toAppendTo.append((int)diff);
            }
            else {
               toAppendTo.append(diff);
            }
        }
        return toAppendTo;
    }

    public synchronized Date parse(String text, ParsePosition pos) {
        ...
    }

}
```

Listing 5.1: Datumsformatierer, der jeweils die Differenz zu einem Startwert ausgibt

Wenn wir mit der Klasse `DifferenceDateFormat` den oben beschriebenen Test durchführen, stellen wir fest, dass `DifferenceDateFormat` auf einen normalisierten Wert von 5,5% kommt. Das heißt sie ist 50-mal schneller als der Vierzeiler, 18-mal schneller als `date.toString()` und immerhin fünfmal schneller als die Log4J-Klasse.

5.8 String-Analyse

Die String-Analyse wird gewöhnlich in zwei Teilbereiche unterteilt:

- Lexikalische Analyse mittels eines Scanners (linear)
- Syntaktische Analyse mittels eines Parsers (hierarchisch)

In der lexikalischen Analyse wird eine Zeichenkette in Symbole (Tokens) unterteilt, in der syntaktischen Analyse werden die Symbole in einer hierarchischen Struktur abgelegt, die leicht interpretiert werden kann (beispielsweise als Objektbaum oder mathematischer Ausdruck). Da die String-Analyse das Gegenstück zum Formatieren darstellt und in vielen Applikationen ein zeitkritischer Faktor ist, werden wir auf einige zentrale Aspekte eingehen.

Genau wie beim Formatieren werden wir uns zunächst anschauen, wie schnell eine einfache ganze Zahl geparsed wird. Auch diesmal wollen wir die Standardmethoden aus den Klassen `Integer` und `Long` sowie `NumberFormat` gegeneinander antreten lassen. Zusätzlich schicken wir einen selbst geschriebenen `NumberParser` (Listing 5.2) ins Rennen, der ausschließlich Zahlen zur Basis zehn korrekt lesen kann. Im Gegensatz zur `Integer.toString()`-Methode ist `Integer.parseInt(string)` nämlich nicht optimiert. Sie delegiert lediglich an die allgemeinere `Integer.parseInt(string, basis)`-Methode. `NumberParser.parseInt(string)` ist im Wesentlichen eine vereinfachte Version der `Integer.parseInt(string, basis)`-Methode.

Beachten Sie, dass `NumberParser` zudem eine Methode `parseInt(string, offset, length)` anbietet, die es erlaubt, einen Teilbereich eines Strings zu parsen, ohne extra einen neuen Teilstring mittels `substring()` erzeugen zu müssen.

Gemessen wurde jeweils folgende Schleife:

```
for (int i=0; i<1000; i++) Integer.parseInt("123456");
// bzw. Long.ParseLong("123456"), formatter.parse("123456")
// oder NumberParser.parseInt("123456")
```

Integer.parseInt()	Long. parseLong()	numberFormat-ter.parse()	NumberParser. parseInt()
100%	172%	526%	32%

Tabelle 5.12: Vergleich verschiedener Analyse-Methoden für ints

Wie die Ergebnisse aus Tabelle 5.12 zeigen, entspricht das Bild in etwa den Erfahrungen vom Formatieren. Je spezifischer eine Analyse-Methode für die Aufgabe geschrieben wurde, desto besser das Ergebnis.

```
package com.tagtraum.perf.strings;

public class NumberParser {

   public static final int MAX_NEGATIVE_INTEGER_CHARS
      = Integer.toString(Integer.MIN_VALUE).length();
   public static final int MAX_POSITIVE_INTEGER_CHARS
      = Integer.toString(Integer.MAX_VALUE).length();

   public static int parseInt(String s)
         throws NumberFormatException {
      return parseInt(s, 0, s.length());
   }

   public static int parseInt(String s, int offset, int length)
         throws NumberFormatException {
```

```
        if (s == null) throw new NumberFormatException("null");
        int result = 0;
        boolean negative = false;
        int i = 0;
        int limit;
        int digit;

        if (length > 0) {
           if (s.charAt(offset) == '-') {
              if (length > MAX_NEGATIVE_INTEGER_CHARS)
                 throw new NumberFormatException(s);
              negative = true;
              limit = Integer.MIN_VALUE;
              i++;
           } else {
              if (length > MAX_POSITIVE_INTEGER_CHARS)
                 throw new NumberFormatException(s);
              limit = -Integer.MAX_VALUE;
           }
           while (i < length) {
              digit = s.charAt(offset + i++)-'0';
              if (digit < 0 || digit > 9)
                 throw new NumberFormatException(s);
              result *= 10;
              if (result < limit + digit)
                 throw new NumberFormatException(s);
              result -= digit;
           }
        } else {
           throw new NumberFormatException(s);
        }
        if (negative) {
           if (i > 1) {
              return result;
           } else {
              throw new NumberFormatException(s);
           }
        } else {
           return -result;
        }
     }
  }
```

Listing 5.2: Schneller Integer-Parser. In einer JDK 1.4 Version ließe sich der String-Parameter auch durch einen Parameter vom Typ CharSequence ersetzen. Somit könnten java.nio.CharBuffer, String und StringBuffer gleich behandelt werden.

5.8.1 Datum und Zeit

Natürlich ist das Parsen von Integern ein Mikroaspekt. Etwas schwieriger ist es schon, einen Datumsstring oder ganze Texte zu parsen. Wir wollen uns zunächst einmal mit dem Datum beschäftigen. Und zwar werden wir einen String folgenden Formats parsen: `dd MMM yyyy HH:mm:ss,SSS`. Dies entspricht beispielsweise: `01 Feb 2002 01:09:30,951`.

Natürlich lässt sich das Datum mit einem `SimpleDateFormat`-Objekt einlesen. Wir wollen zum Vergleich einen selbst geschriebenen Parser testen. Außer Konkurrenz werden wir zudem die `parse()`-Methode der `java.util.Date`-Klasse sowie den Parser der oben bereits erwähnten `DifferenceDateFormat`-Klasse ins Rennen schicken. Dieser benutzt einen Algorithmus, der dem der Klasse `NumberParser` (Listing 5.2) gleicht. Daher werden wir hier nicht näher darauf eingehen.

Der selbst geschriebene Parser ist in zwei Klassen implementiert: `AbsoluteTimeDateFormat` (Listing 5.3/Abbildung 5.3) und `DateTimeDateFormat` (Listing 5.4). Dabei erbt `AbsoluteTimeDateFormat` von `java.text.DateFormat` und `DateTimeDateFormat` von `AbsoluteTimeDateFormat`. Die Klassen korrespondieren zu entsprechenden Formatierern aus Log4J. `AbsoluteTimeDateFormat` unterstützt das Format `HH:mm:ss,SSS`, die Klasse `DateTimeDateFormat` unterstützt `dd MMM yyyy HH:mm:ss,SSS`.

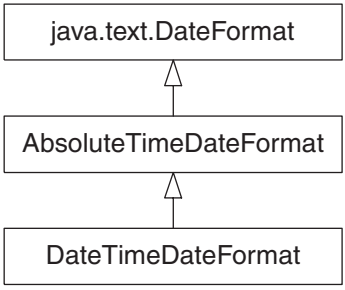

Abbildung 5.3: Klassendiagramm für AbsoluteTimeDateFormat und DateTimeDateFormat

```
package com.tagtraum.perf.strings;

import java.text.*;
import java.util.*;

public class AbsoluteTimeDateFormat extends DateFormat {

   private String lastString;
   private int lastOffset;
   private int lastHour;
   private int lastMinute;
   private int lastSecond;
```

```
public AbsoluteTimeDateFormat() {
   setCalendar(Calendar.getInstance());
   lastString = "";
}

public StringBuffer format(Date date, StringBuffer toAppendTo,
      FieldPosition fieldPosition) {
   ...
}

public Date parse(String s, ParsePosition p) {
   calendar.clear();
   try {
      subParse(s, p);
   } catch (RuntimeException e) {
      p.setErrorIndex(p.getIndex());
      return null;
   }
   return calendar.getTime();
}

protected void subParse(String s, ParsePosition p) {
   if (!getCachedTime(s, p)) {
      int offset = p.getIndex();
      parseTime(s, p);
      putCachedTime(s, offset);
   }
}

private void parseTime(String s, ParsePosition p) {
   calendar.set(Calendar.HOUR_OF_DAY,
      NumberParser.parseInt(s, p.getIndex(), 2));
   calendar.set(Calendar.MINUTE,
      NumberParser.parseInt(s, p.getIndex() + 3, 2));
   calendar.set(Calendar.SECOND,
      NumberParser.parseInt(s, p.getIndex() + 6, 2));
   calendar.set(Calendar.MILLISECOND,
      NumberParser.parseInt(s, p.getIndex() + 9, 3));
   p.setIndex(p.getIndex() + 12);
}

private boolean getCachedTime(String s, ParsePosition p) {
   if (s.regionMatches(p.getIndex(),
         lastString, lastOffset, 9)) {
      // setze gecachte Werte
      calendar.set(Calendar.HOUR_OF_DAY, lastHour);
      calendar.set(Calendar.MINUTE, lastMinute);
      calendar.set(Calendar.SECOND, lastSecond);
      // parse und setze Millisekunden
      calendar.set(Calendar.MILLISECOND,
         NumberParser.parseInt(s, p.getIndex() + 9, 3));
```

```
            p.setIndex(p.getIndex() + 12);
            // gib true zurück, da wir einen Cache-Hit hatten
            return true;
         }
         return false;
      }

      private void putCachedTime(String s, int offset) {
         lastString = s;
         lastOffset = offset;
         lastHour = calendar.get(Calendar.HOUR_OF_DAY);
         lastMinute = calendar.get(Calendar.MINUTE);
         lastSecond = calendar.get(Calendar.SECOND);
      }

   }
```

Listing 5.3: Die Klasse AbsoluteTimeDateFormat kann Zeitstrings des Formats HH:mm:ss,SSS parsen.

```
   package com.tagtraum.perf.strings;

   import java.text.*;
   import java.util.*;

   public class DateTimeDateFormat extends AbsoluteTimeDateFormat {

      private Map monthsMap;
      private String lastString;
      private int lastOffset;
      private int lastYear;
      private int lastDay;
      private int lastMonth;

      public DateTimeDateFormat() {
         super();
         lastString = "";
         // konstruiere Mapping von Monatsnamen auf ints
         String[] shortMonths
            = new DateFormatSymbols().getShortMonths();
         monthsMap = new HashMap();
         for (int i=0; i<shortMonths.length; i++)
            monthsMap.put(shortMonths[i], new Integer(i));
      }

      public StringBuffer format(Date date, StringBuffer toAppendTo,
               FieldPosition fieldPosition) {
         ...
       }

      protected void subParse(String s, ParsePosition p) {
         // parse zunächst den Datumsteil...
```

```
        if (!getCachedDate(s, p)) {
            int offset = p.getIndex();
            parseDate(s, p);
            putCachedDate(s, offset);
        }
        // ... und dann den Zeitteil.
        super.subParse(s, p);
    }

    private void parseDate(String s, ParsePosition p) {
        calendar.set(Calendar.DAY_OF_MONTH,
            NumberParser.parseInt(s, p.getIndex(), 2));
        calendar.set(Calendar.YEAR,
            NumberParser.parseInt(s, p.getIndex() + 7, 4));
        calendar.set(Calendar.MONTH,
            ((Integer)monthsMap.get(s.substring(p.getIndex()
            + 3, p.getIndex() + 6))).intValue());
        p.setIndex(p.getIndex() + 12);
    }

    private boolean getCachedDate(String s, ParsePosition p) {
        if (s.regionMatches(p.getIndex(),
              lastString, lastOffset, 11)) {
            calendar.set(Calendar.DAY_OF_MONTH, lastDay);
            calendar.set(Calendar.YEAR, lastYear);
            calendar.set(Calendar.MONTH, lastMonth);
            p.setIndex(p.getIndex() + 12);
            return true;
        }
        return false;
    }

    private void putCachedDate(String s, int offset) {
        lastDay = calendar.get(Calendar.DAY_OF_MONTH);
        lastYear = calendar.get(Calendar.YEAR);
        lastMonth = calendar.get(Calendar.MONTH);
        lastString = s;
        lastOffset = offset;
    }

}
```

Listing 5.4: Die Klasse DateTimeDateFormat kann Datumsstrings des Formats dd MMM yyyy HH:mm:ss,SSS parsen.

Um die Performance für ähnliche Daten zu verbessern, speichern beide Klassen das zuletzt geparste Datum. Im Fall von `AbsoluteTimeDateFormat` wird dabei der Millisekundenteil nicht mitgespeichert, in der Hoffnung, dadurch die Trefferwahrscheinlichkeit zu erhöhen. Die Abfrage, ob ein Cache-Treffer vorliegt, erfolgt jeweils in der Methode

`subParse()` mit den Methoden `getCachedDate()` bzw. `getCachedTime()`. Die beiden Methoden geben `true` zurück, wenn es Ihnen gelang, das `calendar`-Objekt mit zwischengespeicherten Werten zu manipulieren. Ist dies der Fall, werden die Methoden `parseTime()` bzw. `parseDate()` nicht mehr ausgeführt und die `parse()`-Methode gibt das Datum aus dem `calendar`-Objekt zurück an den Aufrufer.

Zu Beginn des Tests werden jeweils 1.000 verschiedene Datumsstrings in den drei zu testenden Formaten erstellt und in verschiedenen String-Arrays hinterlegt. Dabei repräsentieren die Strings aufeinander folgende Daten, von denen der Cache des selbst geschriebenen Parsers profitieren sollte. Im Test werden alle 1.000 Strings mehrmals nacheinander geparsed. Die Reihenfolge bleibt bei jedem Durchlauf gleich. Um den Effekt des Caches zu messen, wurde zudem eine Reihe mit gemischten, nicht-sequenziellen Daten durchgeführt.

Reihenfolge	Date.parse()	SimpleDate Format	DateTimeDate Format	DifferenceDate Format
sequenziell	100%	118%	24%	3%
zufällig	81%	112%	37%	3%

Tabelle 5.13: Parsen von sequenziellen und gemischten Daten mit verschiedenen Parsern

Das Ergebnis (Tabelle 5.1) zeigt, dass der selbst geschriebene Parser dem `SimpleDateFormat` klar überlegen ist und dass der Cache erheblich zur Performancesteigerung bei sequenziellen Daten beiträgt. Selbst bei zufälligen Daten ist der selbst geschriebene Parser `DateTimeDateFormat` mit 37% noch schneller als `SimpleDateFormat`. Unschlagbar, genau wie beim Formatieren, ist `DifferenceDateFormat`.

5.8.2 Strings teilen

Die wohl einfachste Form der lexikalischen Analyse ist das Aufteilen eines Strings in Teilstrings (Tokens), die jeweils durch Begrenzungszeichen (Delimiter) voneinander getrennt sind. Ein Satz besteht beispielsweise aus einem oder mehreren Wörtern, die durch Leer- und Satzzeichen voneinander getrennt sind. Seit JDK 1.4 existiert die sehr komfortable Methode `string.split()`, die es erlaubt, einen String gemäß eines regulären Ausdrucks in mehrere Teilstrings zu unterteilen. `string.split(regex)` entspricht dabei `java.util.regex.Pattern.compile(regex).split(string)`, woraus sich schließen lässt, dass es sich vermutlich lohnt, einmal kompilierte Patterns wiederzuverwenden.

Vor JDK 1.4 waren die einzigen standardmäßig vorhandenen Klassen zum Aufteilen von Strings bzw. Zeichenströmen `java.util.StringTokenizer` und `java.io.StreamTokenizer`.

Wir wollen die verschiedenen Klassen gegeneinander antreten lassen. Unser Testfall sieht vor, einen langen String mit knapp zweitausend Wörtern in ebendiese Wörter zu unterteilen und sie in einem String-Array zu speichern. Alle Wörter im Quellstring sind dabei durch genau ein Leerzeichen voneinander getrennt. Natürlich ist dies ein Spezialfall – jedoch kein unüblicher.

Alle oben erwähnten Klassen können unseren Testfall und auch andere mögliche Fälle abdecken. Keine ist *wie gemacht* für den Testfall. Wir wollen daher noch eine selbst geschriebene Klasse testen, die *genau* den Testfall abdeckt. Eine Klasse also, die exakt ein Zeichen als Begrenzer zwischen Tokens akzeptiert. Dies ist die Klasse `SingleDelimiterStringTokenizer` (Listing 5.5).

```
package com.tagtraum.perf.strings;

import java.util.*;

public class SingleDelimiterStringTokenizer
      implements Enumeration {

   private char delim;
   private int pos;
   private String string;
   private int length;

   public SingleDelimiterStringTokenizer(String string,
         char delim) {
      this.delim = delim;
      this.string = string;
      length = string.length();
      if (length > 0 && string.charAt(0) == delim)
         pos = 1;
      else
         pos = 0;
   }

   public boolean hasMoreElements() {
      return hasMoreTokens();
   }

   public boolean hasMoreTokens() {
      return !(pos >= length);
   }

   public Object nextElement() {
      return nextToken();
   }

   public String nextToken() {
      if (pos >= length) throw new NoSuchElementException();
```

```
        int start = pos;
        while (pos < length && string.charAt(pos) != delim) {
            pos++;
        }
        String token = string.substring(start, pos);
        pos++; // überspringe nächsten Delimiter
        return token;
    }

    public int countTokens() {
        if (pos >= length) return 0;
        int count = 0;
        int countPos = pos;
        while (true) {
            if (countPos >= length) return count;
            count++;
            while (string.charAt(countPos) != delim) {
                countPos++;
                if (countPos >= length) return count;
            }
            countPos++;
        }
    }
}
```

Listing 5.5: Schneller StringTokenizer, der exakt ein Zeichen zwischen Tokens akzeptiert

Unsere Testmethoden sehen folgendermaßen aus:

```
// normaler StringTokenizer
private String[] stringTokenizer(String s) {
    List list = new ArrayList();
    StringTokenizer st = new StringTokenizer(s, " ");
    while (st.hasMoreTokens()) {
        list.add(st.nextToken());
    }
    return (String[])list.toArray(new String[0]);
}

// spezialisierter StringTokenizer
private String[] singleDelimiterStringTokenizer(String s) {
    List list = new ArrayList();
    SingleDelimiterStringTokenizer st
        = new SingleDelimiterStringTokenizer(s, ' ');
    while (st.hasMoreTokens()) {
        list.add(st.nextToken());
    }
    return (String[])list.toArray(new String[0]);
}
```

```
// StreamTokenizer
private String[] streamTokenizer(String s)
      throws IOException {
   List list = new ArrayList();
   StreamTokenizer st = new StreamTokenizer(
      new ByteArrayInputStream(s.getBytes()));
   st.eolIsSignificant(false);
   st.slashSlashComments(false);
   st.slashStarComments(false);
   // setze Leerzeichen als einzigen Delimiter
   st.whitespaceChars(' ', ' ');
   while (st.nextToken() != StreamTokenizer.TT_EOF) {
      list.add(st.sval);
   }
   return (String[])list.toArray(new String[0]);
}

// string.split()
private String[] stringSplit(String s) throws IOException {
   return s.split(" ");
}

// vorkompiliertes Pattern
private Pattern pattern = Pattern.compile(" ");private static String[]
regexSplit(String s) throws IOException {
   return pattern.split(s);
}
```

Der Testaufbau ist am günstigsten für die `split()`-Methoden, da diese von vorneherein einen String-Array zurückgeben. Alle anderen Varianten müssen erst recht umständlich einen Array erstellen. Dennoch sind die `split()`-Versionen nicht die schnellsten. Wie Tabelle 5.14 zeigt, ist der `SingleDelimiterStringTokenizer` am schnellsten, gefolgt vom `StringTokenizer` und dann erst den beiden `split()`-Varianten. Der `StreamTokenizer` schneidet in unserem Test am schlechtesten ab.

StringTokenizer	StreamTokenizer	SingleDelimiter	string.split()	pattern.split()
100%	300%	67%	152%	147%

Tabelle 5.14: Aufteilen eines Strings in Wörter

Wenn Sie also sehr schnell einen String in seine Bestandteile zerlegen wollen und diese immer durch das gleiche Zeichen getrennt sind, lohnt es sich evtl., einen eigenen Tokenizer zu schreiben oder den hier vorgestellten `SingleDelimiterStringTokenizer` zu verwenden. Ansonsten sind `StringTokenizer` sowie – für komplexere Fälle – die `split()`-Methoden zu empfehlen. Vom `StreamTokenizer`, der ja an sich auch einem anderen Zweck dient, ist jedoch abzuraten.

5.8.3 Reguläre Ausdrücke und lexikalische Analyse mit Grammatiken

Seit JDK 1.4 enthält Java das Reguläre-Ausdrücke-Paket `java.util.regex`. Wir haben es gerade schon implizit mit der Methode `split()` benutzt. Alternativ zum in JDK 1.4 enthaltenen Paket können Sie beispielsweise auch das *Jakarta-Oromatcher-* oder *Regexp-*Paket sowie IBM Alphaworks *Regex for Java* benutzen. Es liegt außerhalb des Fokus dieses Buches, einen fairen Vergleich der vier Pakete durchzuführen. Da es (noch) keine Java Standardschnittstelle zu regulären Ausdrücken gibt, die eine Service-Provider-Architektur unterstützt, sei Ihnen daher empfohlen, Ihre Software so zu bauen, dass Sie das Reguläre-Ausdrücke-Paket leicht gegen ein anderes austauschen können. Hierzu sind die Muster *Fabrikmethode* und *Adapter* [Gamma96, S. 115f./S.151f.] sehr hilfreich. Dies macht jedoch nur Sinn, wenn reguläre Ausdrücke wirklich ein zeitkritischer Faktor in Ihrer Applikation sind.

Ebenso liegt die lexikalische Analyse mit Grammatiken nicht im Fokus dieses Buches. Geeignete Werkzeuge zum Erzeugen von entsprechenden Parsern sind beispielsweise *JLex*, *JFlex* und *JavaCC* (*Java Compiler Compiler*).

- Jakarta ORO: *http://jakarta.apache.org/oro/*
- Jakarta Regexp: *http://jakarta.apache.org/regexp/*
- IBM Alphaworks Regex for Java: *http://www.alphaworks.ibm.com/tech/regex4j*
- JFlex von Gerwin Klein: *http://www.jflex.de/*
- JLex von Elliot Berk: *http://www.cs.princeton.edu/~appel/modern/java/JLex/*
- JavaCC: *http://www.webgain.com/products/java_cc/*

6 Bedingte Ausführung, Schleifen und Switches

`if`, `for`, `while` und `switch` sind essentielle Befehle für den Fluss jedes Java-Programms. Sie bestimmen, ob und wie oft etwas ausgeführt wird. Um diese Strukturen möglichst effizient zu nutzen, bedarf es etwas Achtsamkeit.

6.1 Bedingte Ausführung

Die bedingte Ausführung mit `if` ist trivial. Doch selbst `if`-Konstrukte können durch unachtsam dahingeschriebenen Code bei häufiger Ausführung zum Flaschenhals werden. Daher lohnt es, sich einige Regeln anzueignen.

6.1.1 Logische Operatoren

Beginnen wir mit logischen Operatoren.

Bevorzugen Sie die bedingten Operatoren && und || an Stelle von & und |

Der einzige Unterschied zwischen `&&` und `&` bzw. `||` und `|` für logische (nicht arithmetische!) Operationen liegt darin, dass bei `&` und `|` immer beide Operanden evaluiert werden, bei `&&` und `||` jedoch der zweite Operand nur berechnet wird, wenn dies auch notwendig ist [Gosling00, §15.23/24]. Bei logischen Oder-Operationen ist dies der Fall, wenn der erste Operand `false`, bei logischen Und-Operationen, wenn der erste Operand `true` ist. Das bedingte Evaluieren der Operanden wird auch *Short-Circuiting* genannt.

Beispiel:

```
// Tun Sie dies nicht!
if (a==2 | a==aufwaendigeBerechnung()) {
   ...
}
```

Die Methode `aufwaendigeBerechnung()` wird jedes Mal ausgeführt, selbst wenn `a` gleich zwei und das Ergebnis von `aufwaendigeBerechnung()` somit irrelevant ist. Verwenden Sie stattdessen den bedingten `||`-Operator:

```
// aufwaendigeBerechnung() wird nur ausgeführt, wenn a!=2
if (a==2 || aufwaendigeBerechnung()) {
   ...
}
```

Gleiches gilt für Und-Operationen:

```
// Tun Sie dies nicht!
if (a==2 & aufwaendigeBerechnung()) {
   ...
}
```

Auch hier wird die Methode `aufwaendigeBerechnung()` immer ausgeführt – egal ob `a` gleich zwei ist oder nicht. Und das, obwohl das Ergebnis bereits feststeht, wenn `a` ungleich zwei ist. Stattdessen sollten Sie daher folgenden Code verwenden:

```
// aufwaendigeBerechnung() wird nur ausgeführt, wenn a==2
if (a==2 && aufwaendigeBerechnung()) {
   ...
}
```

In den obigen Beispielen haben wir jeweils einen einfachen Ausdruck (`a==2`) als ersten und einen aufwändigen Ausdruck (`aufwaendigeBerechnung()`) als zweiten Operand benutzt. Dies macht nur Sinn, wenn wir wissen, dass `a` tatsächlich einigermaßen oft gleich zwei ist. Wüssten wir, dass in den meisten Fällen `a` ungleich zwei ist, sollten wir besser `a==2` als zweiten Operand setzen. Allgemein gilt:

Ordnen Sie die Operanden in logischen Und/Oder-Operationen nach Berechnungsaufwand und Wahrscheinlichkeit.

So können Sie sicherstellen, dass während der Ausführung der am wenigsten aufwändige Entscheidungsweg gewählt wird.

6.1.2 String-Switches

Das nächste Code-Beispiel zeigt die bedingte Ausführung verschiedener Methoden in Abhängigkeit von einem String. Da in Java das `switch/case`-Konstrukt nur `ints` akzeptiert, müssen wir umständlich etwas Ähnliches mit `if/else` kodieren.

```
// Tun Sie dies nicht!
if (s.equals("Action1")) {
   action1();
}
```

```
if (s.equals("Action2")) {
   action2();
}
if (s.equals("Action3")) {
   action3();
}
if (s.equals("Action4")) {
   action4();
}
```

Ein solches Code-Stück findet sich häufig in Swing-Anwendungen. So wie hier gezeigt, werden immer alle `if`-Konstrukte und somit auch alle `equals()`-Methoden aufgerufen. Angenommen, `s` ist gleich `"Action1"`, und weiter angenommen, `s` wird in der `action1()`-Methode nicht verändert, dann ist das Ausführen der restlichen drei `if`-Konstrukte samt ihrer `equals()`-Methoden überflüssig. Gilt für alle Ausdrücke in den `if`-Konstrukten, dass nicht gleichzeitig zwei von ihnen wahr sein können, so sollte man besser folgenden Code schreiben:

```
if (s.equals("Action1")) {
   action1();
}
else if (s.equals("Action2")) {
   action2();
}
else if (s.equals("Action3")) {
   action3();
}
else if (s.equals("Action4")) {
   action4();
}
```

Wenn Sie jetzt zusätzlich noch wissen, dass für die Häufigkeit `h` der Aktionen gilt ***h(Action4) > h(Action2) > h(Action3) > h(Action1)***, dann sollten Sie die `if`-Konstrukte auch entsprechend sortieren:

```
if (s.equals("Action4")) {
   action4();
}
else if (s.equals("Action2")) {
   action2();
}
else if (s.equals("Action3")) {
   action3();
}
else if (s.equals("Action1")) {
   action1();
}
```

Diese Version ist schon sehr viel schneller als die erste. Im schlechtesten Fall wird die Ausführungszeit jedoch immer noch linear mit der Anzahl der verschiedenen Aktionen steigen. Mit anderen Worten, sie skaliert nicht. Wenn Sie in sehr viele verschiedene Aktionen verzweigen müssen, kann es sich daher lohnen, einen anderen Weg zu gehen.

6.1.3 Befehlsobjekte

Ersetzen Sie lange `if`/`else`-Konstrukte durch eine Tabelle mit Befehlsobjekten.

Listing 6.1 zeigt, wie Sie ein langes `if`/`else`-Konstrukt mit linearer Laufzeit elegant durch eine Hashmap-Leseoperation (Zeile 41) mit nahezu konstanter Laufzeit ersetzen können. Dazu wird zunächst eine gemeinsame Schnittstelle `Action` für alle Aktionen definiert (Zeilen 1-5). Im Konstruktor der `ActionSwitchDemo`-Klasse werden dann anonyme Klassen, die `Action` implementieren, instanziiert und unter einem Schlüssel in der Hashmap `actionMap` hinterlegt (Zeilen 11-25). Da anonyme Klassen auf die Methoden der sie umgebenden Instanz zugreifen dürfen, können wir die Methoden `actionX()` von `ActionSwitchDemo` leicht aus den anonymen Klassen aufrufen.

Das beschriebene Vorgehen korrespondiert zu dem Verhaltensmuster *Befehl*, auch bekannt als *Command*, *Action* oder *Transaction* [Gamma96, S. 245f.]. Das Muster wird sehr gerne in Zusammenhang mit Benutzeroberflächen benutzt und findet javaseitig in der Swing-Klasse `javax.swing.Action` seine Entsprechung.

```
01 package com.tagtraum.perf.ifsloopsandswitches;
02
03 public interface Action {
04    public void action();
05 }
```

```
01 package com.tagtraum.perf.ifsloopsandswitches;
02
03 import java.util.*;
04
05 public class ActionSwitchDemo {
06
07    private Map actionMap;
08
09    public ActionSwitchDemo() {
10       actionMap = new HashMap();
11       actionMap.put("Action1", new Action() {
12          public void action() {
13             action1();
14          }
15       });
16       actionMap.put("Action2", new Action() {
17          public void action() {
```

```
18              action2();
19           }
20        });
21        actionMap.put("Action3", new Action() {
22           public void action() {
23              action3();
24           }
25        });
26     }
27
28     public void action1() {
29        System.out.println("Action1");
30     }
31
32     public void action2() {
33        System.out.println("Action2");
34     }
35
36     public void action3() {
37        System.out.println("Action3");
38     }
39
40     public void performAction(String s) {
41        Action action = (Action) actionMap.get(s);
42        if (action != null) action.action();
43     }
44
45     public static void main(String[] args) {
46        ActionSwitchDemo asd = new ActionSwitchDemo();
47        asd.performAction("Action2");
48     }
49 }
```

Listing 6.1: Vermeiden von langen if/else-Konstrukten mit anonymen Action-Klassen

Die obige Lösung ist einigermaßen schnell, sie skaliert und ist typsicher. Der einzige Nachteil ist die hohe Anzahl anonymer Klassen. Denn jede dieser unscheinbaren `Action`-Klassen verbraucht knapp 9 Kbyte Speicher.[1] Das bedeutet, hundert `Action`-Klassen belegen knapp ein Mbyte.

Wenn Sie sicher sind, dass der Speicherverbrauch von `Action`-Klassen tatsächlich ein Problem darstellt, können Sie Reflection (siehe Paket `java.lang.reflect`) als Implementierungs-Alternative in Erwägung ziehen.

```
01 package com.tagtraum.perf.ifsloopsandswitches;
02
03 import java.lang.reflect.InvocationTargetException;
```

1 Gemessen auf einem Windows-2000-System mit Sun JDK 1.4.0.

```
04 import java.lang.reflect.Method;
05 import java.util.*;
06
07 public class ReflectiveActionSwitchDemo {
08
09    private static class ReflectiveAction implements Action {
10       private Method method;
11       private Object targetObject;
12
13       public ReflectiveAction(Object targetObject,
14             String methodName) throws NoSuchMethodException {
15          this.targetObject = targetObject;
16          method = targetObject.getClass()
17             .getMethod(methodName, null);
18       }
19
20       public void action() {
21          try {
22             method.invoke(targetObject, null);
23          } catch (InvocationTargetException ita) {
24             Throwable t = ita.getTargetException();
25             if (t instanceof RuntimeException)
26                throw (RuntimeException) t;
27             if (t != null)
28                t.printStackTrace();
29             else
30                ita.printStackTrace();
31          } catch (IllegalAccessException iae) {
32             iae.printStackTrace();
33          }
34       }
35    }
36
37    private Map actionMap;
38
39    public ReflectiveActionSwitchDemo()
40          throws NoSuchMethodException {
41       actionMap = new HashMap();
42       actionMap.put("Action1",
43          new ReflectiveAction(this, "action1"));
44       actionMap.put("Action2",
45          new ReflectiveAction(this, "action2"));
46       actionMap.put("Action3",
47          new ReflectiveAction(this, "action3"));
48    }
49
50    public void action1() {
51       System.out.println("Action1");
52    }
53
54    public void action2() {
```

```
55        System.out.println("Action2");
56     }
57
58     public void action3() {
59        System.out.println("Action3");
60     }
61
62     public void performAction(String s) {
63        Action action = (Action) actionMap.get(s);
64        if (action != null) action.action();
65     }
66
67     public static void main(String[] args)
68           throws NoSuchMethodException {
69        ReflectiveActionSwitchDemo asd
70           = new ReflectiveActionSwitchDemo();
71        asd.performAction("Action2");
72     }
73  }
```

Listing 6.2: Vermeiden von langen if/else-Konstrukten durch Reflection

Statt einer anonymen Klasse für jede Aktion, benutzt `ReflectiveActionSwitchDemo` nur eine Klasse – nämlich `ReflectiveAction` – für alle Aktionen. `ReflectiveAction` besorgt sich bereits im Konstruktor das Methoden-Objekt der Methode, die es später aufrufen soll (Zeilen 16,17). Wird die `action()`-Methode aufgerufen, muss nur noch die `invoke()`-Methode des Methodenobjektes ausgeführt werden (Zeile 22). Somit kann ein `ReflectiveAction`-Objekt initialisiert werden, eine beliebige Methoden aufzurufen.[2] Für diese Flexibilität gilt es jedoch einen Preis zu zahlen:

- Trotz erheblicher Verbesserungen seit Sun JDK 1.4.0[3] ist `invoke()` verglichen mit einem direkten Methodenaufruf immer noch langsam.
- Sie müssen sich zur Laufzeit mit allerlei unangenehmen Ausnahmen herumschlagen.

Grundsätzlich ist daher die Lösung mit anonymen Klassen vorzuziehen.

6.2 Schleifen

Genau wie `if`-Konstrukte sind Schleifen trivial. Doch auch hier gilt es für Hochgeschwindigkeits-Code einige Regeln zu beachten.

2 Der Einfachheit halber habe ich Methoden-Argumente und Rückgabewerte hier nicht berücksichtigt. Das Beispiel ließe sich aber entsprechend erweitern.

3 Methodenaufrufe über Reflection sind gegenüber Sun JDK 1.3.1 etwa dreimal schneller geworden.

6.2.1 Loop Invariant Code Motion

Die wichtigste Regel lautet:

Entfernen Sie alle unnötigen Operationen aus der Schleife.

Beispiel:

```
// aufwaendigeBerechnung(int) gebe für gleiche Parameter
// immer den gleichen Wert zurück.
int a = 0;
int b = 5;
for (int i=0; i<100; i++) {
   // Tun Sie dies nicht!
   b += aufwaendigeOperation(a) + 5 + i;
}
```

Die Operation `aufwaendigeOperation(a)+5` ist eine Invarianz und sollte vor der Schleife ausgeführt werden:

```
int a = 0;
int b = 5;
int c = aufwaendigeOperation(a) + 5;
for (int i=0; i<100; i++) {
   b += c + i;
}
```

Stellen Sie sicher, dass sich nichts in der Schleife befindet, was nicht auch wirklich mehrfach ausgeführt werden muss. Teilweise wird diese Optimierung auch von VMs durchgeführt. Das Verfahren heißt *Loop Invariant Code Motion*. VMs sind jedoch nicht unbedingt in der Lage zu erkennen, dass `aufwaendigeOperation(a)` für gleiche `a`-Werte auch immer den gleichen Wert zurückgibt. Es bleibt Ihnen überlassen, ob Sie sich auf die VM verlassen oder auf Nummer sicher gehen wollen.

Beachten Sie, dass die Abbruchbedingung ein Teil der Schleife ist. Sie ist quasi die erste Zeile. Folgender Code ist somit ineffizient, da die Methode `list.size()` bei jeder Iteration aufgerufen wird, während ein Aufruf vor der Schleife reichen würde:

```
// list sei vom Typ List
// Dies ist ineffizient!
for (int i=0; i<list.size(); i++) {
   Object o = list.get(i);
   ...
}
```

Verfahren Sie also besser folgendermaßen:

```
for (int i=0, n=list.size(); i<n; i++) {
   Object o = list.get(i);
   ...
}
```

Der Methodenaufruf `list.size()` wird nur einmal ausgeführt und der Rückgabewert der Variablen `n` zugewiesen. Dabei ist `n` eleganterweise nur in der `for`-Schleife sichtbar.

6.2.2 Teure Array-Zugriffe

Vermeiden Sie wiederholten Zugriff auf Arrays.

Diese Regel trifft auf folgenden Code zu:

```
private int arrayAccessInLoop() {
   int[] array = new int[1];
   for (int i=0; i<20000000; i++) {
      // Wiederholter Array-Zugriff ist sehr aufwändig!
      array[0] += i;
   }
   return array[0];
}
```

Gemäß Java Sprachspezifikation muss die VM bei jedem Array-Zugriff überprüfen, dass die Array-Referenz nicht auf `null` zeigt und dass der angegebene Index innerhalb der Array-Grenzen liegt. Das heißt, bei obigem Code überprüfen die meisten VMs ohne Not immer wieder, ob das Array-Element `0` tatsächlich vorhanden ist. Neuere VMs wie der HotSpot Server Compiler verfügen über eine Optimierung namens *Range Check Elimination*, die es erlaubt, auf die Überprüfung zu verzichten, wenn der Compiler aufgrund einer Codeanalyse beweisen kann, dass das Element vorhanden sein muss. Ebenso kann unter Umständen auf den wiederholten `null`-Test verzichtet werden (*Null Check Elimination*).

Tests mit verschiedenen JDKs ergeben, dass die folgende funktional gleiche Variante des Codes wesentlich schneller ist als die obige Version (Tabelle 6.1).

```
private int arrayAccessBeforeLoop() {
   int[] array = new int[1];
   // Array-Wert in temp kopieren
   int temp = array[0];
   for (int i=0; i<20000000; i++) {
      temp += i;
   }
   // temp zurück in den Array kopieren
   array[0] = temp;
   return array[0];
}
```

Sun JDK 1.3.1 Client	Sun JDK 1.3.1 Server	Sun JDK 1.4 Client	Sun JDK 1.4 Server	IBM JDK 1.3.0
2	1,38	1,67	1,35	2,56

Tabelle 6.1: Faktor, um den die Methode arrayAccessBeforeLoop() in einer bestimmten VM schneller ist als die Methode arrayAccessInLoop()

6.2.3 Loop Unrolling

Eine weitere klassische Optimierungstechnik ist das so genannte *Loop Unrolling*. Dabei wird der Zähler einer Schleife in jeder Iteration statt um eins um beispielsweise zehn erhöht. Zum Ausgleich wird der Körper der Schleife zehnmal kopiert. Auf diese Weise muss nur jede zehnte Iteration getestet werden, ob die Abbruchbedingung erfüllt ist.

```
private int normalLoop() {
   int a=0;
   for (int i=0; i<20000000; i++) {
      a += i;
   }
   return a;
}
```

Obiger Code ließe sich also folgendermaßen umschreiben:

```
private int unrolledLoop() {
   int a=0;
   for (int i=0; i<20000000; i+=10) {
      a += i;
      a += i+1;
      a += i+2;
      a += i+3;
      a += i+4;
      a += i+5;
      a += i+6;
      a += i+7;
      a += i+8;
      a += i+9;
   }
   return a;
}
```

Der Erfolg dieser Optimierung ist jedoch je nach VM sehr unterschiedlich (Tabelle 6.2). In den HotSpot Server VMs führt Loop Unrolling zu einer schlechteren und in den HotSpot Client VMs sowie dem IBM JDK zu einer besseren Laufzeit. Absolut gesehen ist die Performance in den HotSpot Server VMs am besten (28%) – und zwar ohne manuelles Loop Unrolling.

Java VM	normalLoop()	unrolledLoop()	Faktor
Sun JDK 1.3.1 Client	100%	60%	1,68
Sun JDK 1.3.1 Server	28%	32%	0,90
Sun JDK 1.4.0 Client	138%	61%	2,26
Sun JDK 1.4.0 Server	28%	32%	0,89
IBM JDK 1.3.0	50%	34%	1,48

Tabelle 6.2: Normalisierte Ausführungszeit der beiden Methoden in verschiedenen Java VMs sowie der Faktor, den die Methode unrolledLoop() in einer VM schneller (>1) bzw. langsamer (<1) ist als die Methode normalLoop()

Für die Praxis bedeutet dies:

Manuelles Loop Unrolling lohnt sich meist nicht.

Es macht den Code unlesbar und führt bei HotSpot Server VMs sogar zu einer schlechteren Laufzeit.

6.2.4 Schleifen vorzeitig verlassen

Gelegentlich muss man eine Schleife vorzeitig verlassen. Häufig wird dazu eine boolesche Hilfsvariable benutzt. Der Code sieht etwa folgendermaßen aus:

```
// Tun Sie dies nicht!
private int booleanTerminatedLoop() {
   int a=0;
   boolean done = false;
   for (int i=0; i<20000000 && !done; i++) {
      a += i;
      if (a == -1) done = true;
   }
   return a;
}
```

Wenn eine bestimmte Bedingung eintrifft (hier `a==-1`)[4], wird die Variable `done` auf `true` gesetzt und die Schleife terminiert. Die Alternative zur booleschen Variable ist der von einigen Puristen verachtete `break`-Befehl:

```
private int breakTerminatedLoop() {
   int a=0;
   for (int i=0; i<20000000; i++) {
      a += i;
      if (a == -1) break;
   }
   return a;
}
```

4 Tatsächlich wird a niemals –1, das ist jedoch für das Beispiel unerheblich.

Dieser Code ist jedoch unabhängig von der VM wesentlich schneller als die Version mit boolescher Variable (Tabelle 6.3). Daher gilt:

Benutzen Sie `break` anstelle boolescher Hilfsvariablen, um Schleifen vorzeitig zu verlassen.

Java VM	booleanTerminatedLoop()	breakTerminatedLoop()	Faktor
Sun JDK 1.3.1 Client	100%	72%	1,39
Sun JDK 1.3.1 Server	122%	44%	2,77
Sun JDK 1.4.0 Client	152%	100%	1,52
Sun JDK 1.4.0 Server	102%	45%	2,29
IBM JDK 1.3.0	67%	45%	1,50

Tabelle 6.3: Normalisierte Ausführungszeit der beiden Methoden in verschiedenen Java VMs sowie der Faktor, um den die Methode breakTerminatedLoop() in einer VM schneller ist als die Methode booleanTerminatedLoop()

6.2.5 Ausnahmeterminierte Schleifen

Beim indizierten Zugriff auf eine Liste oder einen Array wird in Java immer eine Ausnahme ausgelöst, wenn ein Index außerhalb der Listen- oder Array-Grenzen liegt. Man kann also, anstatt bei jeder Iteration die Grenzen selbst zu testen, einfach auf die Ausnahme warten, die ausgelöst wird, wenn die Grenze überschritten wird. *Dies ist jedoch sehr schlechter Stil!*

Der Code sähe folgendermaßen aus:

```
ArrayList list;

// exceptionTerminatedLoop
// list sei initialisiert und mit Strings gefüllt
// Sehr schlechter Stil!
try {
   for (int i=0; true; i++) {
      String s = (String)list.get(i);
      ...
   }
} catch (IndexOutOfBoundsException e) {
   // ignoriere Ausnahme
}
```

Normalerweise würde man diesen Code so schreiben:

```
// normalLoop
// list sei initialisiert und mit Strings gefüllt
for (int i=0, n = list.size(); i<n; i++) {
   String s = (String)list.get(i);
   ...
}
```

Nicht nur, dass die normale Variante viel leichter lesbar ist, ein kleiner Test mit 100.000 Elementen offenbart zudem, dass schlechter Stil sich nicht unbedingt auszahlt. Lediglich mit dem IBM JDK ist die ausnahmebeendete Schleife um den Faktor 1,73 schneller. Mit allen anderen getesteten Java VMs ist die sauber kodierte Version entweder leicht schneller oder gleich schnell.

Java VM	**normalLoop()**	**exceptionTerminatedLoop()**	**Faktor**
Sun JDK 1.3.1 Client	100%	100%	1,00
Sun JDK 1.3.1 Server	57%	61%	0,93
Sun JDK 1.4.0 Client	101%	107%	0,94
Sun JDK 1.4.0 Server	29%	38%	0,77
IBM JDK 1.3.0	306%	177%	1,73

Tabelle 6.4: Normalisierte Ausführungszeit der beiden Methoden in verschiedenen Java VMs sowie der Faktor, den die Methode normalLoop() in einer VM schneller (>1) bzw. langsamer (<1) ist als die Methode exceptionTerminatedLoop()

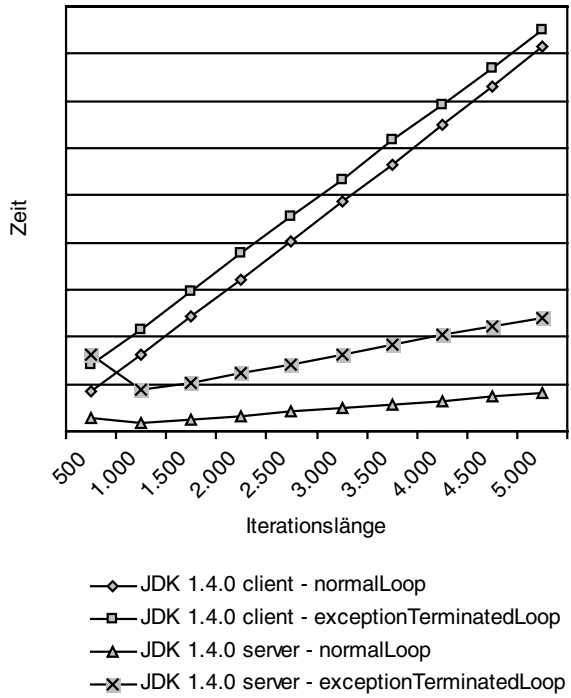

Abbildung 6.1: Dauer von normalen und ausnahmeterminierten Schleife in Abhängigkeit von der Iterationslänge (500-5.000) für Sun JDK 1.4.0

Natürlich hängt das Ergebnis dieses Tests von der Anzahl der Elemente in der Liste ab. Daher habe ich den Test noch einmal mit dem Sun JDK 1.4.0 und unterschiedlich vielen Elementen durchgeführt (Abbildung 6.1 und Abbildung 6.2).

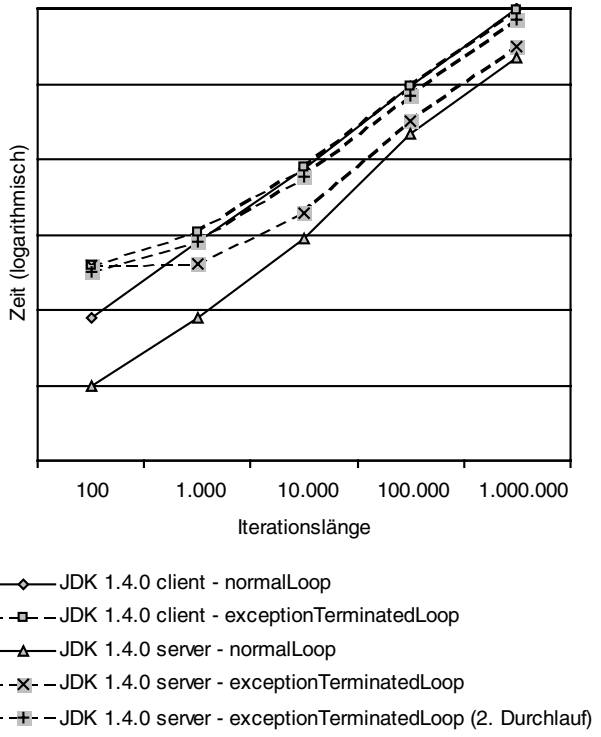

Abbildung 6.2: Dauer von normalen und ausnahmeterminierten Schleifen in Abhängigkeit von der Iterationslänge (100-1.000.000) für Sun JDK 1.4.0

Unabhängig von der Anzahl der Elemente in der Liste schnitt die normale Schleife mindestens genauso gut ab wie die ausnahmeterminierte. Bei der Server-Version ist zudem der zweite Durchlauf der ausnahmeterminierten Schleife signifikant langsamer, was darauf schließen lässt, dass der HotSpot-Server-Compiler Probleme hat, ausnahmegesteuerten Code zu optimieren. Zusammenfassend lässt sich sagen:

Ausnahmeterminierte Schleifen sind oft langsamer als normal terminierte Schleifen.

6.2.6 Iteratoren oder nicht?

Im letzten Beispiel haben wir über eine arraybasierte Liste iteriert. Dabei benutzten wir nicht den von der Methode `iterator()` zurückgegebenen `java.util.Iterator`, sondern haben einfach mit der `get()`-Methode über den Index zugegriffen.

Den Iterator zu benutzen wäre sicherlich besserer Stil gewesen. Doch sind Iteratoren performant? Wir wollen folgende vier verschiedene Methoden vergleichen:

```
ArrayList arrayList;
LinkedList linkedList;
...
// arrayList und linkedList seien initialisiert
// und mit 1.000 Strings gefüllt

// ArrayList mit Iterator
private String arrayListIteratorLoop() {
   String s = null;
   for (Iterator i = arrayList.iterator(); i.hasNext();) {
      s = (String)i.next();
   }
   return s;
}

// LinkedList mit Iterator
private String linkedListIteratorLoop() {
   String s = null;
   for (Iterator i = linkedList.iterator(); i.hasNext();) {
      s = (String)i.next();
   }
   return s;
}

// ArrayList-Iteration mit get(index)
private String arrayListLoop() {
   String s = null;
   for (int i=0, n = arrayList.size(); i<n; i++) {
      s = (String)arrayList.get(i);
   }
   return s;
}

// LinkedList-Iteration mit get(index)
private String linkedListLoop() {
   String s = null;
   for (int i=0, n = linkedList.size(); i<n; i++) {
      s = (String)linkedList.get(i);
   }
   return s;
}
```

Java VM	ArrayList mit get()	ArrayList mit Iterator	LinkedList mit get()	LinkedList mit Iterator
Sun JDK 1.3.1 Client	100%	259%	3.267%	164%
Sun JDK 1.3.1 Server	54%	219%	2.338%	111%
Sun JDK 1.4.0 Client	105%	216%	3.338%	142%
Sun JDK 1.4.0 Server	25%	90%	2.262%	73%
IBM JDK 1.3.0	406%	590%	3.140%	322%

Tabelle 6.5: Normalisierte Ausführungszeit des Iterierens über eine Liste mit 1.000 Elementen

Wie Sie Tabelle 6.5 entnehmen können, ist das Iterieren über eine `ArrayList` mit einem `Iterator` einiges langsamer als der direkte Zugriff mittels Index. Das Iterieren über eine `LinkedList` mittels `Iterator` hingegen ist ein Vielfaches schneller als der Zugriff über den Index.

Dabei ist der Strafzoll potenziell höher, wenn Sie auf eine `LinkedList` mittels `get()` zugreifen, als wenn Sie über eine `ArrayList` mit einem `Iterator` iterieren. Dies ist so, weil der Zugriff über `get()` oder den `Iterator` auf eine `ArrayList` jeweils in konstanter Zeit erfolgt. Bei einer `LinkedList` jedoch erfolgt der lesende Zugriff über einen Index in linearer Zeit und ist somit direkt von der Länge der Liste abhängig. Das Lesen aus einer `LinkedList` mit einem `Iterator` benötigt dagegen nur konstante Zeit, da der `Iterator` sich seine Position merkt und nicht erst mühsam den aktuellen Index suchen muss. Die Iteration über `LinkedList` skaliert also nur, wenn Sie den `Iterator` benutzen.

Für Objekte vom Typ `LinkedList` empfiehlt sich also der `Iterator`, für Listen vom Typ `ArrayList` der Zugriff mittels `get()`.

Nun ist es guter Stil, für Referenzen anstelle der Implementierungklasse den Schnittstellentyp zu verwenden. Also statt `ArrayList` oder `LinkedList` einfach `List` als Referenztyp zu benutzen:

```
List list = new ArrayList();
```

So können Sie auch nachträglich noch die implementierende Klasse (`ArrayList`) austauschen. Gleichzeitig wollen Sie aber auch performant über diese Liste iterieren können. Es bleiben Ihnen drei Möglichkeiten:

- Sie benutzen einfach immer einen `Iterator`, da der schlechteste Fall immer noch wesentlich günstiger ist (konstante Zugriffszeit) als der schlechteste Fall beim Zugriff über `get()` (lineare Zugriffszeit).
- Sie prüfen mittels des `instanceof`-Operators oder über die `getClass()`-Methode, ob es sich um eine `ArrayList` oder einen `Vector` handelt, und iterieren gegebenenfalls mittels `get()`. Ist die Liste weder vom Typ `ArrayList` noch `Vector`, benutzen Sie den `Iterator`.

- Sie prüfen, ob die Liste das Interface `java.util.RandomAccess` (ab JDK 1.4) implementiert. Ist dies der Fall, benutzen Sie `get()`, ansonsten den `Iterator`.

Welche der drei Möglichkeiten Sie wählen sollten, hängt davon ab, welches JDK Sie verwenden und wie viel Aufwand Sie tatsächlich treiben wollen, um optimal über eine Liste zu iterieren.

6.3 Optimale Switches

Auch Switches lassen sich optimieren. Und zwar sieht die Java VM Spezifikation zwei verschiedene Möglichkeiten der Übersetzung eines `switch`/`case`-Ausdrucks in Bytecode vor: *Lookupswitch* und *Tableswitch*.

Bei einem Lookupswitch werden alle `case`-Ausdrücke linear durchsucht und bei Übereinstimmung der entsprechende Code ausgeführt. Stimmt keiner der `case`-Werte mit dem `switch`-Wert überein, wird der Code der `default`-Marke ausgeführt. Bei einem Tableswitch hingegen wird ein Offset errechnet und direkt zum passenden `case`- oder `default`-Ausdruck gesprungen. Entsprechend kann ein Tableswitch in konstanter Zeit ausgeführt werden, während ein Lookupswitch lineare Ausführungszeit benötigt. Dies gilt aber nur, wenn der Bytecode direkt interpretiert wird. Natürlich ist jede VM frei, beliebige Optimierungen anzuwenden, die das Laufzeitverhalten komplett umkrempeln.

So viel zur Theorie. Wir wollen an einem Beispiel durchmessen, ob und welche Effekte wir beobachten können.

Die folgenden zwei Methoden sind funktional identisch, werden aber vom Compiler unterschiedlich übersetzt – `oneSwitch()`mit einem Lookupswitch und `twoSwitches()` mit zwei Tableswitches. Welcher Switch-Befehl benutzt wird, lässt sich leicht mit einem Decompiler oder dem im *bin*-Verzeichnis des JDK enthaltenen Disassembler `javap` mit der Option `-c` herausfinden.

Die Entscheidung, welcher der beiden Switch-Befehle benutzt wird, fällt der Compiler in Abhängigkeit davon, ob die Case-Werte direkt aufeinander folgen bzw. wie groß die Lücken zwischen aufeinander folgenden Werten sind. In der Methode `oneSwitch()` besteht eine Lücke zwischen 9 und 100. Diese Lücke ist groß genug, so dass der Compiler sich für den Lookupswitch entscheidet. In der Methode `twoSwitches()` folgen in beiden Switches alle Case-Werte einander. Daher entscheidet der Compiler sich für den Tableswitch.

```
// wird mit Lookupswitch übersetzt
private int oneSwitch() {
   int defaultValue = 99;
   int a = 0;
   for (int i=0; i<500; i++) {
```

```
        switch(i) {
           case 0: a=0; break;
           case 1: a=defaultValue; break;
           case 2: a=defaultValue; break;
           case 3: a=3; break;
           case 4: a=4; break;
           case 5: a=defaultValue; break;
           case 6: a=6; break;
           case 7: a=defaultValue; break;
           case 8: a=defaultValue; break;
           case 9: a=9; break;
           case 100: a=0; break;
           case 101: a=defaultValue; break;
           case 102: a=defaultValue; break;
           case 103: a=3; break;
           case 104: a=4; break;
           case 105: a=defaultValue; break;
           case 106: a=6; break;
           case 107: a=defaultValue; break;
           case 108: a=defaultValue; break;
           case 109: a=9; break;
           default: a=defaultValue;
        }
     }
     return a;
  }

  // wird mit Tableswitches übersetzt
  private int twoSwitches() {
     int defaultValue = 99;
     int a = 0;
     for (int i=0; i<500; i++) {
        switch(i) {
           case 0: a=0; break;
           case 1: a=defaultValue; break;
           case 2: a=defaultValue; break;
           case 3: a=3; break;
           case 4: a=4; break;
           case 5: a=defaultValue; break;
           case 6: a=6; break;
           case 7: a=defaultValue; break;
           case 8: a=defaultValue; break;
           case 9: a=9; break;
        }
        switch(i) {
           case 100: a=0; break;
           case 101: a=defaultValue; break;
           case 102: a=defaultValue; break;
           case 103: a=3; break;
           case 104: a=4; break;
           case 105: a=defaultValue; break;
```

```
            case 106: a=6; break;
            case 107: a=defaultValue; break;
            case 108: a=defaultValue; break;
            case 109: a=9; break;
            default: a=defaultValue;
        }
    }
    return a;
}
```

Java VM	oneSwitch()	twoSwitches()	Faktor
Sun JDK 1.3.1 Client	100%	104%	0,96
Sun JDK 1.3.1 Server	49%	45%	1,08
Sun JDK 1.4.0 Client	106%	104%	1,02
Sun JDK 1.4.0 Server	53%	7% (32%)	7,46 (1,63)
IBM JDK 1.3.0	72%	48%	1,50

Tabelle 6.6: Normalisierte Ausführungszeit der beiden Methoden sowie der Faktor, um den twoSwitches() schneller war als oneSwitch()

Die Ergebnisse in Tabelle 6.6 zeigen, dass die Werte für die beiden Methoden nur beim Sun JDK 1.4.0 Server und dem IBM JDK signifikant unterschiedlich waren. Wie erwartet war dabei die Methode `twoSwitches()` schneller.[5]

Für die Praxis bedeutet unser Ergebnis:

Switches aufzusplitten kann sich unter bestimmten Bedingungen lohnen, sollte jedoch nur an besonders performancekritischen Stellen angewandt werden.

Das Splitten wirkt sich dabei umso positiver aus, je mehr `case`-Ausdrücke im Switch enthalten sind und je öfter der `default`-Ausdruck ausgeführt wird.

5 Merkwürdigerweise war jedoch der erste Durchlauf von `twoSwitches()` beim Sun JDK 1.4.0 Server reproduzierbar wesentlich schneller als der zweite Durchlauf. In Tabelle 6.6 stehen die Werte für den zweiten Durchlauf in Klammern.

7 Ausnahmen

Das Wichtigste vorweg: Wenige, wenn nicht keine Java VMs sind auf das Auslösen und Verarbeiten von Ausnahmen optimiert. Daraus folgt:

Benutzen Sie Ausnahmen nur für Ausnahmesituationen.

Die in *Kapitel 6.2.5 Ausnahmeterminierte Schleifen* beschriebenen ausnahmeterminierten Schleifen sind deshalb nicht schneller als konventionell kodierte Schleifen, weil das Auslösen einer Ausnahme für die VM einen gewissen Kraftakt darstellt. Daraus folgt insbesondere:

Benutzen Sie Ausnahmen niemals zum Steuern des Kontrollflusses.

Ihr Code wird sonst nicht nur unlesbar und schwer wartbar, sondern auch langsam.

7.1 Ausnahmen durch sinnvolle Schnittstellen vermeiden

Es kommt vor, dass bestimmte Methoden abhängig von einem Zustand Ausnahmen auslösen müssen. Wenn Sie selbst eine solche Klasse kodieren und der ausnahmeauslösende Zustand *vor* dem Aufruf der Methode bekannt ist, können Sie die Schnittstelle so kodieren, dass die Ausnahme vom Aufrufer umgangen werden kann, indem er zuvor den Zustand abfragt. Ein einfaches Beispiel hierfür ist das Interface `java.util.Iterator`.

So lange es noch Elemente gibt, über die iteriert werden kann, liefert die Methode `next()` ein neues Objekt zurück. Gibt es keine Elemente mehr, wird eine `NoSuchElementException` ausgelöst. Dies lässt sich jedoch vermeiden, indem man vor jedem `next()`-Aufruf mit der Methode `hasNext()` überprüft, ob noch ein weiteres Element in der Datenstruktur enthalten ist. Dementsprechend wird gewöhnlich folgendes Idiom für eine Iteration benutzt:

```
for (Iterator i=collection.iterator(); i.hasNext();) {
   Object o = i.next();
}
```

Ein anderes Beispiel ist folgende (fiktive) Schnittstelle zum Überprüfen von Berechtigungen:

```
public interface Authorization {

   /**
    * Überprüft die Berechtigung einer Person, eine Aktion
    * auszuführen.
    *
    * @exception SecurityException falls die Person zur Ausführung
    *                              der Aktion nicht berechtigt ist.
    */
   public void checkPermission(Person person, Action action)
         throws SecurityException;

   /**
    * Gibt an, ob eine Person eine gegebene Aktion ausführen darf.
    *
    * @return true, falls die Person die Aktion ausführen darf.
    */
   public boolean hasPermission(Person person, Action action);
}
```

Die Methode `checkPermission()` kann sehr leicht in Code eingebettet werden – beispielsweise in einem EJB oder einer Bibliothek. Insbesondere ist sie geeignet für Routinechecks, ähnlich dem, ob ein Objekt `null` ist, bevor eine Methode ausgeführt wird oder nicht. Sie ist am besten zu benutzen, wenn generell erwartet wird, dass die erforderliche Berechtigung vorliegt.

Die Methode `hasPermission()` hingegen wird mit der Erwartung aufgerufen, dass ein Benutzer durchaus nicht die erforderliche Berechtigung hat. Sie kann beispielsweise dazu benutzt werden, in einer grafischen Benutzerschnittstelle nur jene Schaltflächen anzuzeigen, die der Benutzer auch betätigen darf. Es wäre ungerechtfertigt, vor dem Anzeigen einer Schaltfläche jeweils eine Ausnahme zu riskieren – zumal ja noch nicht einmal eine Ausnahme vorliegt.

Wenn Sie eine Schnittstelle entwerfen, denken Sie an Methoden zum Abfragen von Zuständen zum Vermeiden von unnötigen Ausnahmen.

Das heißt nicht, dass Sie keine Ausnahmen benutzen oder gar Ausnahmen maskieren sollten. Ausnahmen und insbesondere Stacktraces sind wertvolle, unverzichtbare Hilfen bei der Fehlersuche.

Folgender wenig vorbildhafter Code maskiert die Ausnahme und ersetzt die Fehlerbehandlung durch das Setzen eines Zustandes (`error!=null`).

```
InputStream in;
String error;

// in sei ein geöffneter InputStream

// Tun Sie dies nicht!
public int read() {
   int i=0;
   try {
      i = in.read();
   }
   catch(IOException ioe) {
      // hier geht der Stacktrace verloren
      error = ioe.toString();
   }
   return i;
}

// wird evtl. nie aufgerufen
public String getError() {
   return error;
}
```

Dieses Vorgehen erschwert die Fehlersuche aus folgenden zwei Gründen enorm:

- Der Stacktrace ist verloren. Alles, was Ihnen bleibt, ist eine textuelle Fehlermeldung.
- Ein Klient dieser Klasse ruft eventuell nie die Methode `getError()` auf. Das heißt, Sie erfahren unter Umständen nie, dass eine Ausnahme vorlag.

Benutzen Sie also Ausnahmen mit Bedacht. Weitere nützliche Anmerkungen über den Einsatz können Sie in Joshua Blochs Buch *Effektiv Java Programmieren* [Bloch02, S.173ff] finden.

7.2 Kosten von Try-Catch-Blöcken in Schleifen

Eine weit verbreitete Ansicht lautet, dass zu viele `try-catch`-Blöcke Programme langsam machen. Tatsächlich ist dies nicht unbedingt der Fall. Betrachten Sie folgende zwei Methoden und die normalisierten Ausführungszeiten in Tabelle 7.1:

```
private int loopWithoutTryCatch() {
   int a = 0;
   for (int i=0; i<100000; i++) {
      a += Integer.parseInt("1");
   }
   return a;
}
```

```
private int loopWithTryCatch() {
   int a = 0;
   for (int i=0; i<100000; i++) {
      try {
         a += Integer.parseInt("1");
      }
      catch (NumberFormatException nfe) {
         nfe.printStackTrace();
      }
   }
   return a;
}
```

Java VM	mit try-catch	ohne try-catch	Faktor
Sun JDK 1.3.1 Client	100%	97%	1,03
Sun JDK 1.3.1 Server	56%	58%	0,97
Sun JDK 1.4.0 Client	87%	86%	1,01
Sun JDK 1.4.0 Server	31%	31%	1,02
IBM JDK 1.3.0	73%	64%	1,13

Tabelle 7.1: Normalisierte Ausführungszeiten sowie der Faktor, den die Methode ohne try-catch-Block schneller ist als die Methode mit try-catch-Block

Nur im IBM JDK 1.3.0 führt der `try-catch`-Block zu einem signifikanten Performance-Verlust.

Nun haben wir in den beiden Methoden auf etwas seltsame Weise `a++` ausgeführt. Das Benutzen von `a += Integer.parseInt("1")` garantierte, dass in der Schleife eine nicht-triviale und somit schwer zu optimierende Methode ausgeführt wurde, die den größten Teil der Rechenzeit schluckt. Wenn wir stattdessen das funktional gleiche, sehr viel schnellere und wesentlich leichter zu optimierende `a++` ausführen, kommen wir zu anderen Ergebnissen.

Java VM	mit try-catch	ohne try-catch	Faktor
Sun JDK 1.3.1 Client	100% (entspricht 143 s)	57%	1,75
Sun JDK 1.3.1 Server	1,8%	1,8%	1,00
Sun JDK 1.4.0 Client	86%	79%	1,08
Sun JDK 1.4.0 Server	nicht messbar	nicht messbar	nicht messbar
IBM JDK 1.3.0	119%	29% (nicht messbar)	4,15 (nicht messbar)

Tabelle 7.2: Normalisierte Ausführungszeiten der a++-Variante sowie der Faktor, den die Methode ohne try-catch-Block schneller ist als die Methode mit try-catch-Block

Tabelle 7.2 zeigt, dass die normalisierten Ausführungszeiten für Sun JDK 1.3.1 Client und IBM JDK 1.3.0 ohne `try-catch`-Block wesentlich kürzer sind. Bei Sun JDK 1.3.1 Server und Sun JDK 1.4.0 Client ist jedoch kaum ein nennenswerter Unterschied zu beobachten. Interessant ist, dass Sun JDK 1.4.0 Server unsere triviale Methode so weit optimiert, dass die Ausführungszeit nicht hinreichend lang ist (weitaus kleiner als 1%), um zu einem messbaren Ergebnis zu kommen. Gleiches gilt für den zweiten Lauf der Variante ohne `try-catch`-Block im IBM JDK. Diese Optimierung gelang der IBM VM jedoch nicht bei der Variante mit `try-catch`-Block.

`try-catch`-Blöcke führen bei aktuellen Sun VMs also kaum mehr zu Performance-Verlusten, können jedoch bei IBM JDK 1.3.0 zu Einbußen führen und insbesondere Optimierungen verhindern, die andernfalls greifen würden.

Ein signifikanter Performance-Gewinn durch einen `try-catch`-Block außerhalb von Schleifen ist aber nur zu erwarten, wenn der Schleifenkörper hinreichend kurz ist und entsprechend oft ausgeführt wird. Denn wie wir oben gesehen haben, ist der zu erwartende Gewinn vernachlässigbar groß, wenn eine ausreichend komplexe Methode wie zum Beispiel `Integer.parseInt()` innerhalb des Blocks ausgeführt wird.

Mit anderen Worten:

Abhängig von der VM und dem Inhalt einer Schleife kann es sich lohnen, `try-catch`-Blöcke bevorzugt außerhalb von Schleifen zu platzieren, wenn dadurch der Programmablauf nicht beeinträchtigt wird.

In normalen Programmen scheinen die möglichen Performance-Gewinne jedoch so klein zu sein, dass es sich kaum lohnt, aus Geschwindigkeitsgründen auf `try-catch`-Blöcke zu verzichten.

7.3 Keine eigenen Ausnahme-Hierarchien

In vielen Projekten wird als Erstes eine eigene Ausnahme-Hierarchie erstellt. Schließlich, so das Argument, müsse man einheitlich mit Ausnahmen umgehen können und Regelungen finden, bevor der eigentliche Anwendungscode geschrieben werde. Also geht man daher und erstellt eine Ausnahme-Hierarchie wie in Abbildung 7.1.

Der Ansatz, sich Gedanken zu machen, bevor das Kind in den Brunnen gefallen ist, ist sicherlich lobenswert – an dieser Stelle jedoch völlig fehl am Platze.

Im JDK 1.4.0 sind 270 Ausnahme-Klassen enthalten. Vermutlich können Sie 95% aller Ausnahmen, die je von Ihrem Code ausgelöst werden, mit diesen Ausnahmen abbilden. Und die 5%, die nicht in den vorhandenen Ausnahmen enthalten sind, lassen sich meist durch Erben von einer vorhandenen Ausnahme erstellen.

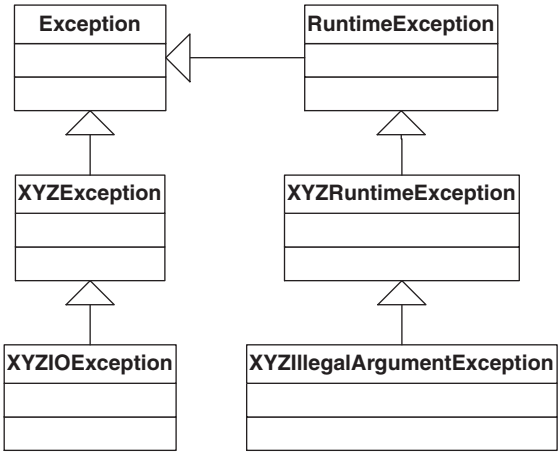

Abbildung 7.1: Ausschnitt einer Parallel-Ausnahme-Hierarchie der Firma XYZ, die eine eigene XYZIOException und eine eigene XYZIllegalArgumentException enthält

Was passiert, wenn Sie diese Regel nicht beherzigen, ist Folgendes: In Ihrem Code werden ständig Ausnahmen aus dem JDK gefangen und verpackt in Ihren eigenen Ausnahmen wieder geworfen.

```
InputStream in;

// in sei ein initialisierter InputStream

public int read() throws XYZIOException {
   // Tun Sie dies nicht!
   try {
      return in.read();
   }
   catch (IOException ioe) {
      throw new XYZIOException(ioe);
   }
}
```

Und dies sind die Konsequenzen:

- Schwierigere Fehlersuche, da die ursprüngliche Ausnahme nicht mehr so leicht oder gar überhaupt nicht mehr zugänglich ist.
- Statt nur eine Ausnahme erzeugen Sie mindestens zwei. Da Ausnahmen eine außerordentliche Last für die Java VM darstellen, wird Ihr Programm langsamer.

Also:

Erstellen Sie keine eigene Parallel-Ausnahme-Hierarchie.

7.4 Automatisch loggende Ausnahmen

Oftmals wird als Argument für eine eigene Ausnahme-Hierarchie angeführt, dass man das automatische Loggen von Ausnahmen in einer Basisklasse (`XYZException`) implementieren wolle. Gemeint ist, dass sich jede Ausnahme der Firma XYZ automatisch in einer Logdatei verewigt, ohne dass der Entwickler Code zu genau diesem Zweck schreiben müsste.

Wenn Sie auf eine solche Idee kommen und überzeugt davon sind, dass Sie unbedingt automatisch loggende Ausnahmen benötigen, implementieren Sie den nötigen Code in einer Hilfsklasse, an die Sie von Ihren eigenen loggenden Ausnahmen delegieren. Aber benutzen Sie auf jeden Fall die Ausnahme-Hierarchie des JDKs. Ihre Klasse `com.xyz.LogIOException` sollte also von `java.io.IOException` erben und nicht von `com.xyz.LogException`. Auf diese Weise ersparen Sie sich das lästige Fangen und Wiederauslösen von Ausnahmen.

Auf jeden Fall vorzuziehen ist es jedoch, dem Entwickler die Entscheidung zu überlassen, welche Ausnahme geloggt werden muss und welche nicht. Im Zweifelsfall kann ohnehin derjenige, der die Ausnahme fängt, besser beurteilen, ob sie geloggt werden muss, als der, der sie auslöst. Sie müssen nur Ihren Entwicklern vertrauen! Nur so können Sie wirklich performanten Ausnahme-Code schreiben.

Hierbei ist noch anzumerken, dass alle Ausnahmen, von denen man glaubt, dass sie garantiert nicht ausgelöst werden, die aber behandelt werden müssen, unbedingt auch geloggt werden müssen. Wenn sie nämlich doch mal ausgelöst werden, haben Sie ein riesiges Problem den Fehler zu finden.

```
String s;
// s sei so initialisiert, dass wir fest daran glauben,
// dass nichts schief gehen kann...
try {
   Class klass = Class.forName(s);
   ...
}
catch (ClassNotFoundException cnfe) {
   // so unwahrscheinlich es auch scheint: loggen!
   cnfe.printStackTrace();
}
```

7.5 Ausnahmen wieder verwenden

Um die beträchtlichen Kosten beim Erstellen einer Ausnahme zu umgehen, können Sie Ausnahmen wiederverwenden. Wir wollen versuchen abzuschätzen, wie groß der Performance-Gewinn ist. Betrachten wir folgende beiden Methoden:

```
private void normalException() {
   for (int i=0; i<10000; i++) {
      try {
         throw new Exception();
      }
      catch (Exception e) {
         // wir ignorieren diese Ausnahme bewusst
      }
   }
}

private final Exception REUSED_EXCEPTION
   = new Exception("Stacktrace ist ungültig.");

private void reusedException() {
   for (int i=0; i<10000; i++) {
      try {
         throw REUSED_EXCEPTION;
      }
      catch (Exception e) {
         // wir ignorieren diese Ausnahme bewusst
      }
   }
}
```

Java VM	normale Ausnahme	wieder verwendete Ausnahme	Faktor
Sun JDK 1.3.1 Client	100%	5%	19,44
Sun JDK 1.3.1 Server	75%	nicht messbar	nicht messbar
Sun JDK 1.4.0 Client	121%	7%	18,83
Sun JDK 1.4.0 Server	99%	nicht messbar	nicht messbar
IBM JDK 1.3.0	58%	9%	6,35

Tabelle 7.3: Normalisierte Ausführungszeit sowie der Faktor, den die Methode reusedException() schneller ist als normalException()

Die Methode `reusedException()` ist in allen Fällen deutlich schneller als die Methode `normalException()`. Bei den beiden Sun Server VMs lagen die Werte für `reusedException()` sogar deutlich unter einem Prozent.[1]

Rein unter Performance-Gesichtspunkten betrachtet, lohnt es sich also, Ausnahmen wiederzuverwenden. Falls Sie sich für diese Technik entscheiden, bedenken Sie jedoch, dass Sie sämtliche Informationen aus dem Stacktrace verlieren, da der Stacktrace beim Erzeugen der Ausnahme angelegt wird. Auch das Aufrufen von `fillInStackTrace()`

1 Dies kann evtl. jedoch auch daran liegen, dass der Compiler erkennt, dass die Ausnahme unbedeutend für die weitere Ausführung und deshalb wegoptimierbar ist.

führt nicht aus diesem Dilemma, da es genauso lange dauert wie das Auslösen einer neuen Ausnahme. Sie erschweren sich die Fehlersuche also ganz erheblich. Gewöhnlich sollte zudem in einer echten Ausnahmesituation Performance nicht mehr so wichtig sein.

8 Datenstrukturen und Algorithmen

Die Wahl der richtigen Datenstrukturen und Algorithmen kann leicht über Erfolg und Misserfolg eines Projekts entscheiden. Deshalb ist es notwendig, zumindest über ein Grundwissen in diesem Bereich zu verfügen. Tatsächlich kommt es in den meisten Fällen gar nicht so sehr darauf an, selbst großartige Datenstrukturen und Algorithmen zu schreiben, sondern vielmehr darauf, qualifiziert entscheiden zu können, welche Datenstruktur zu welchem Zweck am besten geeignet ist; zu wissen, was die Vor- und Nachteile verschiedener Datenstrukturen sind und wie performant die dazugehörigen Operationen ausgeführt werden können.

Daher wollen wir uns zunächst kurz die Groß-O-Notation anschauen, bevor die einzelnen Klassen und Schnittstellen des Collections-Framework beschrieben und mit Hilfe der Groß-O-Notation eingeordnet werden.

Anschließend möchte ich Ihnen noch zwei Beispiele für den intelligenten Einsatz von Datenstrukturen zeigen sowie Caches zur Beschleunigung von Speicherzugriffen diskutieren.

8.1 Groß-O-Notation

Die Groß-O-Notation geht auf das 1894 vom deutschen Mathematiker Paul Bachmann (1837–1920) publizierte Werk *Analytische Zahlentheorie* zurück. Edmund Landau (1877–1938), der eine Vielzahl mathematischer Schriften verfasste, sorgte für die Verbreitung dieser Notation, weswegen früher auch vom Landau-Symbol anstatt vom großen O die Rede war.

Die Notation dient dazu, obere Schranken für die Komplexität von Algorithmen anzugeben. Sie ist folgendermaßen definiert:[1]

$f(n) \in O(g(n))$ für $n \to \infty$ genau dann, wenn $\exists$ Konstante $c>0$, $n_0 \in \mathbb{N}$, so dass
$(\forall n \geq n_0) \; |f(n)| \leq c\,|g(n)|$

1 Eigentlich werden noch andere Schranken und Symbole in der Groß-O-Notation definiert. Für unsere Zwecke reicht jedoch ein Verständnis dieser einen Definition.

$f(n) \in O(g(n))$ bedeutet also auf gut Deutsch: die Laufzeit $f(n)$ eines Algorithmus wächst bis auf einen konstanten Faktor nicht schneller als die Funktion $g(n)$.

Üblicherweise bezieht man sich dabei nicht auf irgendwelche Funktionen $g(n)$, sondern auf einfache, allgemein bekannte Funktionen (Tabelle 8.1).

Eine sequenzielle Suche hat beispielsweise ein Laufzeitverhalten von $O(n)$. Eine binäre Suche hingegen hat ein Laufzeitverhalten von $O(\log n)$. Das bedeutet, dass für große n eine binäre Suche immer schneller ist als eine sequenzielle.

Funktion $g(n)$	Name	Beispiel
c	Konstant	Hashtabelle[1]
$\log n$	Logarithmisch	Binäre Suche
n	Linear	Sequentielle Suche
$n \log n$	-	Quicksort
n^2	Quadratisch	Multiplikation von zwei *n*-stelligen Zahlen
n^3	Kubisch	Matrizenmultiplikation
2^n	Exponenziell	Travelling Salesman Problem

Tabelle 8.1: Typische Funktionen $g(n)$

1 Gilt nur näherungsweise, sofern die Hashfunktion die Elemente gleichmäßig verteilt und die Kollisionsrate sehr gering ist.

Um zu verdeutlichen wie viel ein Algorithmus mit $O(a())$ schneller ist als ein Algorithmus mit $O(b())$, wollen wir annehmen, ein Algorithmus benötigt 10^{-6} Sekunden für eine Operation. Dann ergeben sich in Abhängigkeit von der Problemgröße n und $O(g(n))$ die Laufzeiten aus Tabelle 8.2.

n	$O(\log_2 n)$	$O(n)$	$O(n^2)$	$O(2^n)$
10	0,000003s	0,00001s	0,0001s	0,001s
100	0,000007s	0,0001s	0,01s	10^{16} Jahre
1.000	0,00001s	0,001s	1,0s	astronomisch
10.000	0,000013s	0,01s	1,7min	astronomisch
100.000	0,000017s	0,1s	2,8h	astronomisch

Tabelle 8.2: Ausführungszeiten von Algorithmen in Abhängigkeit von ihrer Komplexität und der Problemgröße

Offensichtlich lohnt es sich, Algorithmen einer geringen Komplexitätsklasse zu verwenden.

Nach diesem kurzen Ausflug in die Klassifizierung von Algorithmen haben wir nun das Rüstzeug uns mit den konkreten Datenstrukturen der Java 2 Plattform samt ihrer Algorithmen auseinander zu setzen.

8.2 Collections-Framework

Das *Collections-Framework* ist seit JDK 1.2 Teil von Java. Zuvor musste man sich mit Bibliotheken Dritter behelfen oder auf die Klassen `Hashtable` und `Vector` beschränken. Da beide Klassen voll synchronisiert sind, führte dies, vor allem bei älteren Java VMs, zu Performance-Einbußen. Mit dem Collections-Framework verfügt Java nun seit einiger Zeit über sehr gute Basisdatenstrukturen mit akzeptabler Performance. Es beinhaltet Schnittstellen der abstrakten Datentypen Liste, Menge (`Set`) und Tabelle (`Map`), verschiedene Implementierungen sowie eine nützliche Hilfsklasse namens `Collections`.

8.2.1 Collections, Sets und Listen

`Collection` ist die Superschnittstelle für Sets und Listen. Abbildung 8.1 zeigt die Klassenhierarchie inklusive Implementierungen.

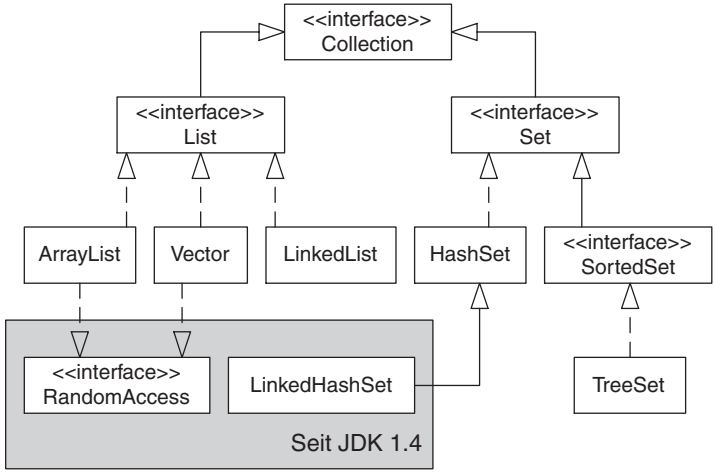

Abbildung 8.1: Klassenhierarchie von `Collection`

Hier eine kurze Beschreibung der Schnittstellen und der wichtigsten definierten Operationen:

- `Collection`

  Menge von Objekten.

  Die wichtigsten definierten Operationen sind hinzufügen (`add()`), entfernen (`remove()`) und ist-enthalten (`contains()`). Darüber hinaus gibt es eine Methode `iterator()`, die einen `Iterator` zurückgibt, mit dem man über die Elemente der `Collection` iterieren kann, sowie eine Methode `toArray()`, die eine Array-Repräsentation der `Collection` zurückgibt.

- `List`

  Geordnete[2] Menge von Objekten.

  `List` definiert, zusätzlich zu den in `Collection` bereits definierten Methoden, indexbasierte Operationen wie `get(index)`, `set(index, Object)`, `remove(index)`, `indexOf(Object)` und `lastIndexOf(Object)`. Zudem existiert eine Methode `subList()` zum Erstellen einer Sicht (View) auf die ursprüngliche Liste.

- `Set`

  Menge von Objekten, in der jedes Objekt nur einmal vorkommt.

  Die Gleichheit von Objekten ist gegeben, wenn `a.equals(b)`. Ein `Set` enthält also nur Objekte, für die paarweise gilt: `!a.equals(b)` (bzw. `!b.equals(a)`, falls `a==null`). Es sind keine Operationen zusätzlich zu denen aus `Collection` definiert

- `SortedSet`

  Sortierte Menge von Objekten, in der jedes Objekt nur einmal vorkommt.

  `SortedSet` definiert zusätzlich zu den Methoden aus `Collection` noch Methoden zum Erstellen von Sichten auf den vorderen (`headSet()`) oder hinteren Teil (`tailSet()`) sowie einen beliebigen Ausschnitt der Menge (`subSet()`). Zudem gibt es Methoden zum Zugriff auf das erste (`first()`) bzw. letzte Element (`last()`) sowie eine Methode `comparator()`, die den benutzten `Comparator` zurückgibt.

- `RandomAccess`

  Markierungs-Interface, das Listen kennzeichnet, die schnellen, direkten Zugriff (gewöhnlich ***O(c)***) auf ein beliebiges Objekt erlauben. Ab JDK 1.4.

Tabelle 8.3 gibt einen Überblick über das Laufzeitverhalten der Basisoperationen der Standard-Implementierungen von `List` und `Set`.

Klasse	add()	remove(Object)	contains()	get(int)	Synchronisiert
`ArrayList`	***O(c)***	***O(n)***	***O(n)***	***O(c)***	nein
`Vector`	***O(c)***	***O(n)***	***O(n)***	***O(c)***	ja
`LinkedList`	***O(c)***	***O(n)***	***O(n)***	***O(n)***	nein
`HashSet`	***O(c)***	***O(c)***	***O(c)***	-	nein
`TreeSet`	***O(log n)***	***O(log n)***	***O(log n)***	-	nein
`LinkedHashSet`	***O(c)***	***O(c)***	***O(c)***	-	nein

Tabelle 8.3: Laufzeitverhalten verschiedener Implementierungen von `List` *und* `Set`

2 D.h. es gibt eine Ordnung, nämlich genau die, in der Elemente angefügt wurden. *Geordnet* bedeutet also nicht unbedingt *sortiert*.

Natürlich gibt es besonders günstige und ungünstige Szenarien für den Einsatz einer Implementierung. Hier deshalb einige Tipps für den Gebrauch von Listen:

- Falls Sie häufig lesend über einen Index auf Elemente der Liste zugreifen, sollten Sie auf keinen Fall `LinkedList` benutzen, sondern `ArrayList` oder `Vector`.
- `ArrayList` ist grundsätzlich etwas schneller als `Vector`, da `ArrayList` nicht synchronisiert ist. Wie viel schneller hängt jedoch stark von der verwendeten VM ab. Ansonsten gleichen sich die Charakteristika der beiden Klassen. Im Übrigen lässt sich `ArrayList` (wie auch alle anderen `Collection`- und `Map`-Implementierungen) durch einen Synchronisation-Wrapper synchronisieren. Eine entsprechende Fabrikmethode befindet sich in der Hilfsklasse `java.util.Collections`.
- Wollen Sie eine Liste als Kellerspeicher (Stack) benutzen, sei Ihnen `ArrayList` oder `java.util.Stack` empfohlen. `Stack` erbt von `Vector` und weist dieselben Eigenschaften auf. Falls Sie sich für `ArrayList` entscheiden, fügen Sie Elemente immer am Ende der Liste an, nie am Anfang. Gleiches gilt fürs Entfernen. Der Grund dafür ist, dass das Einfügen und Entfernen proportional zur Anzahl der folgenden Elemente dauert, da diese im Array um eine Position verschoben werden müssen. Ein Element am Anfang einer `ArrayList` oder eines `Vectors` einzufügen oder zu entfernen, ist daher eine Operation mit einer Laufzeit von ***O(n)***.
- Das Einfügen oder Entfernen am Anfang oder Ende einer `LinkedList` ist eine Operation der Klasse ***O(c)***. Es müssen lediglich ein paar Referenzen umgebogen oder ausgenullt werden. Das macht `LinkedList` zu einer geeigneten Datenstruktur für Warteschlangen (Queues).

Tipps für den Gebrauch von Sets:

- `HashSet` ist wesentlich schneller als `TreeSet`. Falls die Elemente Ihres Sets nicht sortiert sein müssen, benutzen Sie lieber `HashSet`.
- Benutzen Sie `LinkedHashSet` (ab JDK 1.4) nur, wenn die Reihenfolge, in der Sie Elemente eingefügt haben, von Bedeutung ist oder Sie schnell über die Elemente des Sets iterieren müssen. Da die Elemente verlinkt sind, ist die Laufzeit der Iteration proportional zur Anzahl der Elemente im Set und nicht zur Kapazität der unterliegenden `HashMap`, wie dies bei `HashSet` der Fall ist. Das bedeutet, dass eine große Kapazität nicht die Performance der Iteration schmälert.
- `HashSet` und `LinkedHashSet` lassen sich genau wie `HashMap` und `Hashtable` mit einer Kapazität und einem Ladefaktor (Loadfactor) initialisieren und optimieren. Mehr dazu weiter unten.

8.2.2 Maps

Maps bilden die zweite Haupt-Klassenhierarchie des Collections-Frameworks. Abbildung 8.2 gibt einen Überblick über Klassen und Interfaces.

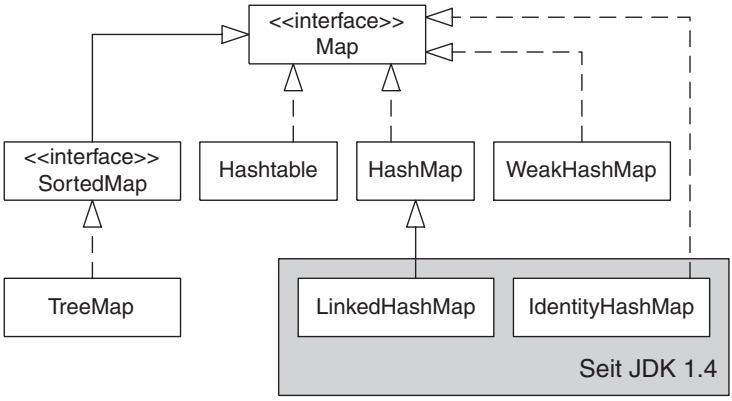

Abbildung 8.2: Klassenhierarchie von Map

Hier eine kurze Beschreibung der beiden Schnittstellen:

- `Map`

  Menge von Schlüssel-/Wert-Paaren, in der jeder Schlüssel nur einmal vorkommt.

  Die wichtigsten definierten Operationen sind hinzufügen (`put()`), entfernen (`remove()`) und entnehmen (`get()`). Darüber hinaus verfügt `Map` noch über Methoden zum Erzeugen von drei Sichten: `entrySet()` gibt ein `Set` von `Map.Entry`-Objekten zurück, `keySet()` ein `Set` der Schlüssel und `values()` eine `Collection` der Werte.

- `SortedMap`

  Nach Schlüsseln sortierte Menge von Schlüssel-Wert-Paaren, in der jeder Schlüssel nur einmal vorkommt.

  `SortedMap` definiert zusätzlich zu den Methoden aus `Map` noch Methoden zum Erstellen von Sichten auf den vorderen (`headMap()`) oder hinteren Teil (`tailMap()`) sowie einen beliebigen Ausschnitt der Tabelle (`subMap()`). Zudem gibt es Methoden zum Zugriff auf den ersten (`firstKey()`) und letzten Schlüssel (`lastKey()`) sowie eine Methode `comparator()`, die den benutzten `Comparator` zurückgibt.

Tabelle 8.4 gibt einen Überblick über das Laufzeitverhalten der Basisoperationen von Standard-Implementierungen des Interfaces `Map`.

Klasse	put()	remove()	get()	containsKey()	Synchronisiert
TreeMap	*O(log n)*	*O(log n)*	*O(log n)*	*O(log n)*	nein
HashMap	*O(c)*	*O(c)*	*O(c)*	*O(c)*	nein
Hashtable	*O(c)*	*O(c)*	*O(c)*	*O(c)*	ja
WeakHashMap	*O(c)*	*O(c)*	*O(c)*	*O(c)*	nein
LinkedHashMap	*O(c)*	*O(c)*	*O(c)*	*O(c)*	nein
IdentityHashMap	*O(c)*	*O(c)*	*O(c)*	*O(c)*	nein

Tabelle 8.4: Laufzeitverhalten verschiedener Implementierungen von `Map`

Natürlich gibt es auch für Maps Hinweise, wann welche Implementierung am geeignetsten ist:

- Genau wie `HashSet` schneller als `TreeSet` ist, sind `HashMap` und `Hashtable` wesentlich schneller als `TreeMap`. Wenn Ihre Map nicht sortiert sein muss, benutzen Sie also lieber `HashMap` oder `Hashtable`.
- Da `HashMap` im Gegensatz zu `Hashtable` nicht synchronisiert ist, ist `HashMap` etwas schneller.
- Wenn Sie `null` als Schlüssel oder Wert benutzen wollen, können Sie nicht `Hashtable` benutzen, da `null` nicht als Schlüssel oder Wert unterstütz wird.
- `LinkedHashMap` (ab JDK 1.4) ist langsamer als `HashMap`. Benutzen Sie die Klasse also nur, wenn die Reihenfolge, in der die Elemente eingefügt wurden, wichtig ist oder Sie schnell über die Elemente iterieren wollen. `LinkedHashMap` eignet sich außerdem dazu, Caches zu implementieren. Dazu muss die `removeEldestEntry()`-Methode überschrieben werden.
- Es gibt sehr wenige Gelegenheiten, in denen die Klasse `WeakHashMap` von Nutzen ist. Sie kann als Cache für Elemente dienen, die so lange im Speicher bleiben müssen, wie ihr Schlüssel-Objekt existiert. Beispielsweise können dies Metainformationen von Klassen sein, für die das Klassenobjekt selbst als Schlüssel fungiert. Im Sun JDK 1.4 werden `WeakHashMaps` genau zu diesem Zweck in den Klassen `java.beans.Introspector` und `java.lang.reflect.Proxy` verwendet.
- Genau wie `WeakHashMap` ist auch `IdentityHashMap` (ab JDK 1.4) eher ein Exot. Im Gegensatz zu allen anderen Map-Implementierungen, werden in einer `IdentityHashMap` zwei Objekte ausschließlich als gleich angesehen, wenn sie identisch sind, d.h. es muss gelten `o1==o2`. Anders ausgedrückt, `o1.equals(o2)` reicht nicht. Ist Identität als Vergleichsfunktion gewünscht, ist `IdentityHashMap` potenziell schneller als `HashMap`.

8.2.3 Hashbasierte Strukturen optimieren

Die Performance von hashbasierten Datenstrukturen wie `HashMap` und `HashSet` hängt im Wesentlichen von fünf Faktoren ab:

- Güte und Geschwindigkeit der Hashfunktion `hashCode()`
- Geschwindigkeit der Vergleichsmethode `equals()`
- Kapazität der Tabelle
- Ladefaktor der Tabelle
- Kollisionsauflösungs-Strategie

Bei jeder Basisoperation einer Hashtabelle muss der Hashcode eines Objektes berechnet werden. Dies geschieht in der Regel durch Ausführen der Methode `hashCode()`.[3] Es ist also essentiell für Hashtabellen, dass die `hashCode()`-Methode der verwendeten Schlüssel schnell ist. Für unveränderbare (immutable) Objekte empfiehlt es sich daher, den Hashcode einmal zu berechnen und dann zwischenzuspeichern, so dass er nicht immer wieder neu berechnet werden muss. Seit JDK 1.3 ist dies auch das Standardverhalten der String-Klasse. Zuvor war der Hashcode immer wieder neu berechnet worden, so dass es sich lohnte, spezielle String-Wrapper, die den Hashcode zwischenspeicherten, als Schlüssel zu benutzen.

Wenn Sie veränderbare Objekte als Schlüssel für eine Hashtabelle benutzen, sollten Sie darüber nachdenken, ob der Hashcode sich für die Zeiten cachen lässt, in denen das Objekt unverändert bleibt. Unabhängig davon sollten Sie jedoch auf jeden Fall sicherstellen, dass die Schlüssel nach dem Einfügen nicht mehr verändert werden, da sie sonst unauffindbar werden, verloren gehen und unnötig Speicherplatz belegen. Nachträglich von außen veränderte Schlüssel eignen sich also hervorragend für ein schwer zu lösendes Speicherproblem.

Neben der Geschwindigkeit von `hashCode()` ist die gleichmäßige Verteilung der Werte über den gesamten Wertebereich von `int` entscheidend. Nur dann ist tatsächlich ein Laufzeitverhalten von ***O(c)*** zu erwarten.

Hashtabellen basieren in der Regel auf einem Array bestimmter Länge, der so genannten Kapazität. Wenn ein Objekt in die Tabelle eingefügt werden soll, wird aus dem Hashcode und der Kapazität ein Array-Index berechnet. Haben das einzufügende Objekt und ein bereits in der Datenstruktur vorhandenes Objekt denselben Index, kommt es zu einer Kollision. Das bedeutet, das Objekt kann nicht einfach am berechneten Index eingefügt werden, sondern es muss ein anderer Platz gefunden werden. Dies bedeutet Extra-Aufwand. Gut verteilte Hashfunktionen minimieren die Anzahl von

3 Einzige Ausnahme hierzu ist die `IdentityHashMap`, die die Methode `System.identityHashCode()` benutzt.

Kollisionen und tragen daher erheblich zur Performance von Hashtabellen bei. Die Klasse `String` beispielsweise berechnet ihren Hashcode mittlerweile mit einer anerkannt guten Hashfunktion, die auch in Bibliotheken anderer Sprachen benutzt wird:

$$\sum_{i=0}^{s.length()-1} s.charAt(s.length() - i - 1) \cdot 31^i$$

`Integer` benutzt einfach seinen eigenen Wert und `Long` berechnet `(int)(value ^ (value >>> 32))`.

Um eine Kollision festzustellen, muss das evtl. bereits in der Datenstruktur vorhandene Objekt mit dem einzufügenden Objekt verglichen werden. Eine Kollision liegt dann vor, wenn es sich nicht um das gleiche Objekt handelt. Ist dies der Fall, muss das Objekt an der nächsten geeigneten Position mit dem einzufügenden Objekt verglichen werden usw., bis ein Platz für das Objekt gefunden wird. Alle hashbasierten Datenstrukturen außer `IdentityHashMap` benutzen für diese Vergleiche die `equals()`-Methode. Daher ist wichtig, dass auch diese möglichst effizient implementiert ist.

Ein weiterer Faktor für die Performance einer hashbasierten Datenstruktur ist die oben bereits erwähnte Kapazität sowie der Ladefaktor. Der Ladefaktor bezeichnet dabei einen Schwellwert im Verhältnis zwischen Kapazität und enthaltenen Elementen. Überschreitet der Quotient von enthaltenen Elementen und Kapazität diesen Schwellwert, so wird die Kapazität automatisch erhöht und die Elemente werden neu verteilt (Rehashing). Da das Neuverteilen eine sehr aufwändige Operation ist, empfiehlt es sich, die Datenstruktur mit einer angemessenen Kapazität zu initialisieren, die einein-halb- bis zweimal so groß ist wie die erwartete Anzahl an Elementen. Es spielt dabei keine Rolle, ob Sie eine gerade, ungerade oder Primzahl wählen. Außer bei `Hashtable` wird die angegebene Kapazität ohnehin zur nächsten Zweierpotenz aufgerundet, da dadurch zur Indexberechnung an Stelle des Modulo-Operators `%` das viel schnellere bitweise Und `&` benutzt werden kann.

Wichtig im Zusammenhang mit dem Ladefaktor ist, dass die Anzahl der Kollisionen bei höherem Ladefaktor zunimmt. `HashMap`, `HashSet` etc. haben mit 0,75 einen vernünftigen Ladefaktor, der zugunsten besserer Geschwindigkeit der Basisoperationen verringert werden kann. Da ein geringer Ladefaktor implizit die Kapazität erhöht und die Anzahl leerer Array-Positionen steigt, erhöht sich so der Speicherverbrauch und verschlechtert sich die Geschwindigkeit der Iteration über die Elemente. Eine Ausnahme von letzterem Effekt ist `LinkedHashMap`, da die Elemente untereinander verlinkt sind und deshalb mit einer Laufzeit proportional zur Anzahl der Elemente iteriert werden kann.

Im Gegensatz zu `HashMap` & Co ist der initiale Ladefaktor der `IdentityHashMap` nur 0,5 statt 0,75. Dies liegt daran, dass `IdentityHashMap` statt dem in `HashMap` verwendeten *direkten Verketten* (*Separate Chaining*) so genanntes *lineares Sondieren* (*Linear Probing*) als Kollisionsauflösungs-Strategie benutzt. Lineares Sondieren ist schneller als direktes

Verketten, führt jedoch bereits ab Ladefaktoren von 0,5 zu schlechterer Performance wegen zu vieler Kollisionen. Weitere Details zu Kollisionsauflösungsstrategien finden Sie beispielsweise in Mark Allen Weiss' Buch *Data Structures & Algorithms in Java* [Weiss99, S.155ff].

8.2.4 Collections

Die Klasse `java.util.Collections` ist eine Hilfsklasse mit einer Vielzahl statischer Methoden. Es lohnt sich, diese Methoden zu kennen, da sie performante Implementierungen für Operationen sind, die in den Collection-Klassen selbst nicht realisiert sind. Dazu gehören insbesondere Sortier- und Suchmethoden. Wenn eine dieser Methoden für Ihre Zwecke tatsächlich nicht schnell genug sein sollte, können Sie immer noch ein eigene Version schreiben. In den meisten Fällen lohnt sich dies jedoch nicht. Wenn Sie diese Methoden verwenden, können Sie außerdem an Verbesserungen des JDK unmittelbar teilhaben. Ein gutes Beispiel dafür ist die unten beschriebene `binarySearch()`-Methode.

Hier eine Kurzbeschreibung der Methoden. Eine vollständige Erläuterung finden Sie in der JDK-Dokumentation.

- `binarySearch()`

  Binäre Suche in Listen nach einem Objekt. Falls nötig, kann ein beliebiger `Comparator` benutzt werden. Trotz des Namens dieser Methode wurden in JDK 1.3.1 Erben von `AbstractSequentialList` (wie beispielsweise `LinkedList`) aus Performancegründen (s.o.) sequenziell statt binär durchsucht. Seit JDK 1.4.0 werden alle Listen binär durchsucht. Dabei wird bei kurzen oder `RandomAccess`-Listen über `get()` und bei anderen Listen über einen `ListIterator` zugegriffen.

- `sort()`

  Sortiert eine Liste, falls nötig mit einem beliebigen `Comparator`. Dabei wird auf die Methode `Arrays.sort(Object[])` zurückgegriffen, die einen modifizierten Mergesort-Algorithmus mit einer garantierten Laufzeit von ***O(n log n)*** verwendet. Die Laufzeit geht gegen ***O(n)*** für teilsortierte Listen.

- `shuffle()`

  Mischt eine Liste – falls nötig mit einem beliebigen `Random`-Objekt.

- `reverse()`

  Kehrt die Reihenfolge einer Liste um.

- `rotate()`

  Rotiert eine Liste um einen Wert. Seit JDK 1.4.

- `swap()`

  Tauscht zwei Elemente einer Liste. Seit JDK 1.4.

- `replaceAll()`

  Ersetzt ein Objekt in einer Liste durch ein anderes. Seit JDK 1.4.

- `copy()`

  Kopiert eine Liste in einer andere.

- `fill()`

  Füllt eine Liste mit einem Objekt.

- `max()/min()`

  Gibt das größte/kleinste Objekt einer `Collection` zurück. Dazu kann ein beliebiger `Comparator` benutzt werden.

- `nCopies()`

  Gibt eine unveränderbare Liste mit *n* Elementen zurück, die alle das gleiche Objekt referenzieren.

- `(last)indexOfSubList()`

  Gibt den Index an, an dem eine Liste in einer anderen Liste enthalten ist. Seit JDK 1.4.

- `enumeration()`

  Erstellt aus einer `Collection` eine `Enumeration`.

- `list()`

  Erstellt aus einer `Enumeration` eine `ArrayList`. Seit JDK 1.4.

- `reverseOrder()`

  Gibt einen `Comparator` zurück, der genau entgegengesetzt zur natürlichen Ordnung sortiert.

- `singleton()`

  Gibt ein unveränderbares Set zurück, das nur das übergebene Objekt beinhaltet.

- `singletonList()/singletonMap()`

  Gibt eine unveränderbare Liste/Map zurück, die nur das übergebene Objekt bzw. Schlüssel-/Wert-Paar enthält.

- `unmodifiableXXX()`

  Gibt einen Wrapper zurück, der die übergebene Datenstruktur enthält und nur lesenden Zugriff zulässt. Das heißt, es wird verhindert, dass die enthaltene Datenstruktur modifiziert wird. Mögliche Datenstrukturen sind `Collection`, `List`, `Map`, `Set`, `SortedMap` und `SortedSet`. Beispiel:

  ```
  Map unmodifiableMap = Collections.unmodifiableMap(new HashMap());
  ```

- `synchronizedXXX()`

  Gibt einen Synchronisations-Wrapper zurück, der die übergebene Datenstruktur enthält. Mögliche Datenstrukturen sind `Collection`, `List`, `Map`, `Set`, `SortedMap` und `SortedSet`. Beispiel:

  ```
  Map synchronizedMap = Collections.synchronizedMap(new HashMap());
  ```

  Die Synchronisations-Wrapper synchronisieren die gesamte Datenstruktur mit nur einem Objekt (Mutex). Das heißt, dass, wenn ein Thread eine Operation ausführt, kein anderer dies tun kann; selbst dann nicht, wenn die beiden Operationen sich nicht behindern würden. Dies ist ein sehr grobes Sperrverhalten, das beispielsweise für Datenbanken undenkbar wäre. Es gibt jedoch Datenstrukturen speziell für den Gebrauch mit mehreren Threads. Mehr dazu in *Kapitel 9.1.3 Threadsichere Datenstrukturen*.

8.3 Jenseits des Collections-Frameworks

Das Collections-Framework ist ein sehr wertvoller Teil der Java-Klassenbibliothek. Seine Existenz alleine verwandelt jedoch kein einziges Programm in ein Performance-Wunder. Das Auswählen bzw. Finden der am besten passenden Datenstruktur ist das Entscheidende. Im Folgenden werden wir für zwei Probleme verschiedene Lösungen ausprobieren und beurteilen.

8.3.1 Zahlen sortieren

Jon Bentley beschreibt in seinem großartigen Buch *Programming Pearls* [Bentley00, S.3ff] ein interessantes Problem: Stellen Sie sich vor, Sie müssten Millionen von paarweise verschiedenen, siebenstelligen natürlichen Zahlen aus einer Datei lesen und sortiert wieder ausgeben. Wie würden Sie vorgehen?

Der naive Ansatz sähe sicherlich wie folgt aus:

```
public void sort(String filename) throws IOException {
   long start = System.currentTimeMillis();
   DataInputStream in = new DataInputStream(
      new BufferedInputStream(new FileInputStream(filename))
   );
   SortedSet set = new TreeSet();
   for (int i=0, l=(int)new File(filename).length()/4; i<l; i++) {
      set.add(new Integer(in.readInt()));
   }
   in.close();
   DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
         new FileOutputStream(filename + ".sorted")
      )
```

```
    );
    for (Iterator i=set.iterator(); i.hasNext();) {
       out.writeInt(((Integer)i.next()).intValue());
    }
    out.close();
    System.out.println(System.currentTimeMillis()-start + "ms");
}
```

Alle Zahlen werden einfach der Reihe nach eingelesen, in `Integer`-Objekte umgewandelt und in ein `TreeSet` eingefügt. Anschließend werden alle Elemente des Sets mit einem `Iterator` wieder ausgegeben.

Um knapp drei Millionen Nummern zu sortieren, benötigt mein Rechner mit diesem Verfahren zwischen 35 und 45 Sekunden. Der Speicherverbrauch liegt bei rund 150 Mbyte.

Offensichtlich ist dieses Vorgehen suboptimal. Wir wollen daher ein paar Änderungen vornehmen. Zunächst einmal ersetzen wir das `TreeSet` durch einen Array. Da wir dadurch keine `Integer`-Objekte mehr benötigen, können wir einen `int`-Array an Stelle eines `Integer`-Arrays benutzen. Das bedeutet, wir ersparen uns das Erzeugen von Millionen von Objekten. Zum Sortieren bedienen wir uns der Methode `java.util.Arrays.sort(int[])`. Es handelt sich dabei um eine Adaption eines von Jon Bentley und M. Douglas McIlroy optimierten Quicksort-Algorithmus. Es ist anzunehmen, dass wir vermutlich keinen ebenbürtigen oder besseren Sortier-Algorithmus implementieren können, ohne sehr viel Zeit zu investieren. Also belassen wir es dabei und wagen einen Testlauf.

```
public void sort(String filename) throws IOException {
    long start = System.currentTimeMillis();
    DataInputStream in = new DataInputStream(
       new BufferedInputStream(new FileInputStream(filename))
    );
    int[] array = new int[(int)new File(filename).length()/4];
    for (int i=0; i<array.length; i++) {
       array[i] = in.readInt();
    }
    in.close();
    java.util.Arrays.sort(array);
    DataOutputStream out = new DataOutputStream(
       new BufferedOutputStream(
          new FileOutputStream(filename + ".sorted")
       )
    );
    for (int i=0; i<array.length; i++) {
       out.writeInt(array[i]);
    }
    out.close();
    System.out.println(System.currentTimeMillis()-start + "ms");
}
```

Für die drei Millionen Nummern benötigen wir jetzt nur noch knapp 10 Sekunden sowie knapp 19 Mbyte. Nicht schlecht. Aber es geht noch besser.

Hier noch einmal die drei wesentlichen Fakten:

- Zahlen von 1.000.000 bis 9.999.999 sollen sortiert werden
- Keine Zahl kommt zweimal vor
- Es gibt also 8.999.999 verschiedene Zahlen

8.999.999 verschiedene, aufeinander folgende Zahlen können auch als 8.999.999 verschiedene wahr/falsch-Zustände oder Bits repräsentiert werden. Für jede Zahl, die wir einlesen, setzen wir also einfach das entsprechende Bit. Auf diese Weise erhalten wir eine sehr platzsparende Repräsentation der Zahlen. Wir wollen uns das mal für den schlechtesten Fall von 8.999.999 Zahlen anschauen:

8.999.999 Bit entsprechen etwa 1.010 Kbyte. Da jedes `int` vier Bytes belegt, entsprechen 8.999.999 `int` etwa 35.000 Kbyte. Wir können den Speicherverbrauch für den Fall, dass wir wirklich alle 8.999.999 Zahlen sortieren müssen, somit um den Faktor 32 verringern. Hinzu kommt, dass wir gar nicht mehr sortieren müssen, da wir ja quasi immer sofort an der richtigen Stelle einfügen. Anstelle des `int`-Arrays benutzen wir also einen `boolean`-Array.

Der Code sähe folgendermaßen aus:

```
public void sort(String filename) throws IOException {
   long start = System.currentTimeMillis();
   DataInputStream in = new DataInputStream(
      new BufferedInputStream(new FileInputStream(filename))
   );
   boolean[] array = new boolean[8999999];
   for (int i=0, b=(int)new File(filename).length()/4; i<b; i++) {
      array[in.readInt()-1000000] = true;
   }
   in.close();
   DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
         new FileOutputStream(filename + ".sorted")
      )
   );
   for (int i=0; i<array.length; i++) {
      if (array[i]) out.writeInt(i+1000000);
   }
   out.close();
   System.out.println(System.currentTimeMillis()-start + "ms");
}
```

Das Sortieren dauert jetzt nur noch zwischen sechs und sieben Sekunden bei einem Speicherverbrauch von etwa 16 Mbyte. Ohne Frage ist das ein besserer Wert als 19 Mbyte. Bei 3 Millionen Nummern sollte der Unterschied Δ jedoch weit höher sein:

$$\Delta = 3.000.000 \cdot 4Byte - \frac{8.999.999}{8} Byte \approx 10Mbyte$$

Statt 3 Mbyte sollte der Unterschied etwa 10 Mbyte betragen. Wo stecken die fehlenden 7 Mbyte?

Ein Blick in die Ausgabe vom Profiler Hprof gibt den entscheidenden Hinweis:

```
SITES BEGIN (ordered by live bytes)
          percent          live       alloc'ed  stack class
 rank   self  accum    bytes objs   bytes objs trace name
    1 95.61% 95.61%  9000016    1 9000016    1   305 [Z <=
    2  0.78% 96.39%    73272  551   74696  576     1 [C
    3  0.69% 97.07%    64528  220   64528  220     0 [I
...
```

Offensichtlich belegt ein `boolean`-Array (in der Hprof-Ausgabe bezeichnet mit dem Klassennamen `[Z`) für jeden booleschen Wert nicht wie angenommen ein Bit, sondern gleich ein ganzes Byte.

Die Lösung liegt in `java.util.BitSet`. `BitSet` speichert einzelne Bits nicht in einem `boolean`-Array, sondern wesentlich effizienter in einem `long`-Array. Der entsprechende Code sähe etwa so aus:

```
public void sort(String filename) throws IOException {
   long start = System.currentTimeMillis();
   DataInputStream in = new DataInputStream(
      new BufferedInputStream(new FileInputStream(filename))
   );
   BitSet set = new BitSet(8999999);
   for (int i=0, b=(int) new File(filename).length()/4; i<b; i++) {
      set.set(in.readInt()-1000000);
   }
   in.close();
   DataOutputStream out = new DataOutputStream(
      new BufferedOutputStream(
         new FileOutputStream(filename + ".sorted")
      )
   );
   for (int i=0; i<8999999; i++) {
      if (set.get(i)) out.writeInt(i+1000000);
   }
   out.close();
   System.out.println(System.currentTimeMillis()-start + "ms");
}
```

Mit dieser Lösung steigt der Speicherverbrauch nur knapp über 8 Mbyte, während die Ausführungszeit weiterhin zwischen 6 und 7 Sekunden liegt. Bessere Ausführungszeiten lassen sich vermutlich nur noch mit Änderungen an der Ein- und Ausgabe erreichen. Dies soll hier jedoch nicht das Thema sein.

8.3.2 Große Tabellen

Tabellenkalkulationen sind mittlerweile Standardsoftware und somit allgegenwärtig. Und natürlich benutzt jede Tabelle eine Datenstruktur. Wenn wir eine solche Tabelle in Java implementieren wollten, müssten wir uns logischerweise auch einer entsprechenden Datenstruktur bedienen. Die naive Lösung wäre ein zweidimensionaler Array.

Heutige Tabellenkalkulationen sind jedoch nicht nur allgegenwärtig, sie sind auch in der Lage riesige Tabellen zu verarbeiten. Wir wollen einmal annehmen, die Tabelle hätte 100.000 Reihen und ebenso viele Spalten. Ein zweidimensionaler Array würde rund 10 Milliarden Felder enthalten und jedes dieser Felder würde vier Byte für eine Objekt-Referenz belegen. Macht also 40 Milliarden Byte. Für einen handelsüblichen Rechner ist dies sicherlich ein wenig viel, daher müssen wir einen anderen Weg finden. Als Erstes schauen wir, was das JDK zu bieten hat.

In Swing heißt die entsprechende Datenstruktur für ein `javax.swing.JTable`-Objekt `javax.swing.table.TableModel`. Es handelt sich dabei um ein Interface, zu dem eine Standardimplementierung `DefaultTableModel` existiert. `DefaultTableModel` benutzt zwar keinen zweidimensionalen `Object`-Array, aber einen `java.util.Vector` randvoll gefüllt mit `Vector`-Objekten, also quasi einen zweidimensionalen `Vector`. Das wäre eigentlich auch nicht so schlimm, wenn nicht während der Initialisierung die Größe aller Vektoren und somit ihrer internen `Object`-Arrays explizit mit der Methode `Vector.setSize()` auf die verlangte Größe gesetzt würde. Mit anderen Worten: 40 Milliarden Byte für `Object`-Arrays plus 100.001-`Vector`-Objekte. Das macht die Sache nicht gerade besser.

Tabellenmodell mit ArrayLists

Als ersten Ansatz könnten wir an Stelle der Vektoren `ArrayLists` benutzen, deren Größe nicht von Anfang an gesetzt, sondern nur bei Bedarf vergrößert wird. Als Basisklasse dient uns hierbei `javax.swing.table.AbstractTableModel`. Die Implementierung sieht wie folgt aus:

```
package com.tagtraum.perf.swing;

import javax.swing.table.AbstractTableModel;
import java.util.ArrayList;

public class ArrayListTableModel extends AbstractTableModel {
```

```
private int cols;
private int rows;
private ArrayList listOfRowLists;

public ArrayListTableModel(int rows, int cols) {
   listOfRowLists = new ArrayList();
   this.rows = rows;
   this.cols = cols;
}

public int getRowCount() {
   return rows;
}

public int getColumnCount() {
   return cols;
}

public Object getValueAt(int row, int col) {
   // Hole die Liste für eine Reihe aus listOfRowLists
   ArrayList rowList = (ArrayList) get(listOfRowLists, row);
   // Falls die Reihe existiert, gib den Eintrag in der
   // verlangten Spalte zurück.
   if (rowList != null) {
      return get(rowList, col);
   }
   return null;
}

public void setValueAt(Object object, int row, int col) {
   // Falls das Object null ist, lösche den Eintrag
   if (object == null) {
      removeValueAt(col, row);
   } else {
      // Hole die Liste für eine Reihe aus listOfRowLists
      ArrayList rowList = (ArrayList) get(listOfRowLists, row);
      // Lege die Liste notfalls an, falls nicht vorhanden
      if (rowList == null) {
         rowList = new ArrayList();
         set(listOfRowLists, row, rowList);
      }
      // Setze den Wert an der entsprechenden Stelle
      set(rowList, col, object);
   }
   // Benachrichtige evtl. vorhandene Listener
   fireTableCellUpdated(row, col);
}

private void removeValueAt(int row, int col) {
   // Hole die Liste für eine Reihe aus listOfRowLists
   ArrayList rowList = (ArrayList) get(listOfRowLists, row);
   if (rowList != null) {
```

```
            // Entferne das Element in col
            remove(rowList, col);
         }
      }

      // Vergrößert die Liste so, dass das zu setzende Element
      // Platz hat.
      private void set(ArrayList list, int index, Object object) {
         while (list.size() - 1 < index) list.add(null);
         list.set(index, object);
      }

      // Gibt null für Indizes zurück, die außerhalb des gültigen
      // Bereichs liegen.
      private Object get(ArrayList list, int index) {
         if (list.size() <= index) return null;
         return list.get(index);
      }

      private void remove(ArrayList list, int index) {
         // Setzt null, sofern der Index im gültigen Bereich liegt.
         if (list.size() > index) list.set(index, null);
         // Falls möglich, reduziere Kapazität der Liste,
         // um so Speicher zu sparen.
         while (!list.isEmpty() && list.get(list.size()-1) == null) {
            list.remove(list.size() - 1);
         }
         if (list.isEmpty())
            listOfRowLists.set(index, null);
         else
            list.trimToSize();
      }

   }
```

Listing 8.1: ArrayList-basiertes TableModel

Statt alle Listen auf die volle Größe zu initialisieren, werden die Listen in der Methode `set()` immer erst mittels `list.add(null)` vergrößert, wenn dies tatsächlich nötig ist. Auf diese Weise wird sehr viel Speicher gespart.

`ArrayListTableModel` skaliert besser als `DefaultTableModel`, ist jedoch auch nicht optimal. Wenn ein Wert in Reihe 0 und Spalte 100.000 gesetzt werden soll, muss `list.add(null)` 99.999-mal aufgerufen werden, um die Liste zu vergrößern. Dabei wird die Kapazität der `ArrayList` für die entsprechende Reihe und somit der entsprechende Objekt-Array vergrößert – und zwar automatisch in 50%-Schritten. Das bedeutet, dass ein einziges Element im schlechtesten Fall rund 600.000 Byte verbraucht. Tatsächlich wird der Array bei einer Anfangskapazität von 10 Objekten auf eine Kapazität von 132.385 Objekten (529.540 Byte) vergrößert.

Die überschüssigen 129.540 Byte ließen sich durch einen Aufruf von `list.trimToSize()` nachträglich entfernen. Die Methode `set()` sähe dann folgendermaßen aus:

```
private void set(ArrayList list, int index, Object object) {
   while (list.size() - 1 < index) list.add(null);
   list.set(index, object);
   list.trimToSize();
}
```

Bei 100.000 Reihen und ebenso vielen Spalten werden so potenziell 12 Milliarden Byte gespart. Obwohl dies eindrucksvoll ist, stehen dem im schlechtesten Fall immer noch rund 400.000 Byte für ein einzelnes Objekt gegenüber. Zudem wird bei jedem `trimToSize()`-Aufruf der gesamte Array kopiert.

Compressed Row Storage

Anscheinend ist unser `ArrayListTableModel` noch nicht der Weisheit letzter Schluss.

Wir wollen daher über mögliche Anwendungsfälle nachdenken. Die Wahrscheinlichkeit, dass ein Benutzer tatsächlich alle 10 Milliarden Felder unserer Tabelle benutzt, ist eher gering. Möglicherweise benötigt der Nutzer im Schnitt lediglich ein paar hundert Felder.

Unser Model entspräche somit einer *dünn besetzten Matrize* (*Sparse Matrix*). Ein Blick in die einschlägige Literatur zeigt: Für dünn besetzte Matrizen existieren bekannte Datenstrukturen. Eine davon ist die *Compressed Row Storage*-Matrize (CRS). Sie besteht aus drei Arrays – einem Zeilen-, einem Spalten- und einem Werte-Array (Abbildung 8.3).

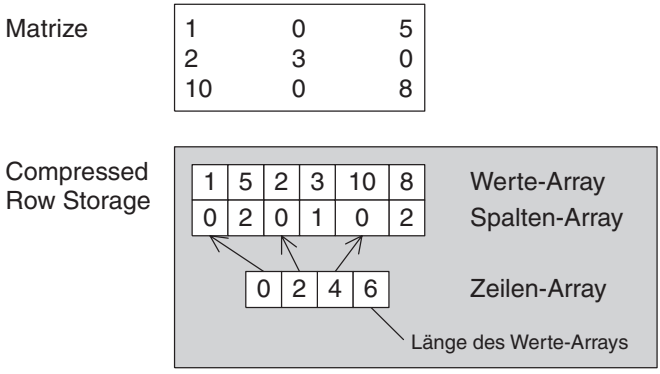

Abbildung 8.3: Abbildung einer Matrize im Compressed Row Storage-Format

Beim lesenden Zugriff wird dabei zunächst mit der Zeile als Index auf den Zeilen-Array zugegriffen. Der Wert des Zeilen-Arrays dient als Zeiger in den Spalten-Array. Ausgehend von dieser Position wird der Spalten-Array nach der verlangten Spaltenzahl durchsucht. Wird der Spalten-Wert gefunden, steht der gesuchte Matrizen-Wert an der gleichen Position im Werte-Array.

Im Grunde würden so die meisten unserer Probleme gelöst. Das CRS-Format hat jedoch zwei entscheidende Nachteile:

1. Wenn in der letzten Zeile ein Wert gesetzt ist, muss der Zeilen-Array zwangsläufig genauso groß sein wie die Zeilenzahl. Da der Zeilen-Array die Werte- und Spalten-Arrays referenziert, müssen auch diese mindestens so groß sein wie die Zeilenzahl. Angenommen wir würden zwei `int`-Arrays und einen `Object`-Array verwenden, dann hätten wir einen Mindest-Speicherverbrauch von 3 mal 4 Byte mal die Zeilenzahl. Bei 100.000 Zeilen wären dies 1.200.000 Byte.
2. Es kann nur mit ***O(log n)*** auf das gesuchte Element zugegriffen werden, da das verlangte Element innerhalb einer Zeile bestenfalls binär gesucht werden kann.

Mit ein wenig Aufwand ließe sich um das erste Problem herumprogrammieren. Das zweite Problem ist jedoch eine feste Eigenschaft der Datenstruktur.

Bessere Zugriffszeiten ließen sich vermutlich mit einer hashbasierten Datenstruktur erreichen.

Hashbasierte Matrize

Die Lösung mit einer hashbasierten Datenstruktur ist erfreulich simpel [vgl. Wilson00, S. 152f.]. Für eine effiziente Implementierung müssen wir lediglich eine Schlüssel-Klasse schreiben, die die Zeilen- und Spalten-Indizes enthält. Alternativ ließe sich hier auch einfach die `java.awt.Point`-Klasse benutzen.

```
package com.tagtraum.perf.swing;

import javax.swing.table.AbstractTableModel;
import java.util.HashMap;
import java.util.Map;

public class HashTableModel extends AbstractTableModel {

   private int cols;
   private int rows;
   private Map map;
   private Key searchKey;

   public HashTableModel(int rows, int cols) {
      map = new HashMap();
      this.rows = rows;
```

```
        this.cols = cols;
    }

    public int getRowCount() {
        return rows;
    }

    public int getColumnCount() {
        return cols;
    }

    public Object getValueAt(int row, int col) {
        return map.get(new Key(row, col));
    }

    public void setValueAt(Object object, int row, int col) {
        if (object == null) {
            if (map.remove(new Key(row, col)) != null) {
                fireTableCellUpdated(row, col);
            }
        }
        else {
            if (map.put(new Key(row, col), object) != object) {
                fireTableCellUpdated(row, col);
            }
        }
    }

    public boolean isCellEditable(int row, int col) {
        return true;
    }

    private static class Key {
        int col;
        int row;
        int hashCode;
        static final Class klass = Key.class;

        public Key(int row, int col) {
            this.row = row;
            this.col = col;
            this.hashCode = col ^ -row;
        }

        public int hashCode() {
            return hashCode;
        }

        public boolean equals(Object obj) {
            if (obj == this) return true;
            if (obj == null || klass != obj.getClass()) return false;
            Key other = (Key) obj;
```

```
            return other.col == col && other.row == row;
        }

        public String toString() {
            return col + ":" + row;
        }
    }
}
```

Listing 8.2: Hashbasierte TableModel-Klasse

Der minimale Speicherverbrauch ist gleich dem der `HashMap`. Die Zugriffszeit ist konstant. Das Einzige, was wir noch tun können, ist diese Zugriffszeit zu optimieren.

Der erste Schritt in diese Richtung ist die `equals()`-Methode von `Key`. Da wir die Schlüssel nirgendwo zwischenspeichern, ist die Chance, dass der Identitätsvergleich in der `equals()`-Methode erfolgreich ist, gleich null. Daher können wir auf ihn verzichten.

Da `Key` eine private Klasse ist, die wir nicht nach draußen geben, in der `HashMap` ausschließlich `Key`-Objekte sind, wir die `HashMap` ebenfalls nicht nach draußen geben und `null` nicht als Schlüssel benutzen, können wir auch auf den `null`-Vergleich verzichten.

Somit sieht die neue `equals()`-Methode folgendermaßen aus:

```
public boolean equals(Object obj) {
    Key other = (Key) obj;
    return other.col == col && other.row == row;
}
```

Tests ergeben, dass der Unterschied zwischen beiden Versionen vernachlässigbar gering ist. Diese Optimierung war also eher theoretischer Natur und somit überflüssig.

Unser nächster Optimierungskandidat ist die `getValueAt()`-Methode. Für jeden Aufruf wird ein neues `Key`-Objekt instanziiert. Wir könnten jedoch für `getValueAt()` immer das gleiche `Key`-Objekt benutzen und nur die Zeilen- und Spaltenwerte neu setzen. Gleiches gilt für `setValueAt()`, wenn das zu setzende Objekt `null` ist und wir eigentlich ein Objekt entfernen wollen. Wichtig ist, dass wir den sich ändernden Suchschlüssel nur zum Suchen benutzen und niemals zum Einfügen. Außerdem dürfen nicht zwei oder mehr Threads gleichzeitig auf die Methoden zugreifen. Da Swing jedoch nur einen Thread benutzt, können wir dies ruhigen Gewissens tun, sofern wir das Modell nicht selbst von einem anderen Thread aus manipulieren wollen.

Die modifizierte `Key`-Klasse sähe folgendermaßen aus:

```
...
public Key(int row, int col) {
    set(row, col);
}
```

```
public Key set(int row, int col) {
   this.row = row;
   this.col = col;
   this.hashCode = col ^ -row;
   return this;
}
...
```

Entsprechend müssen wir noch ein paar Änderungen in der `HashTableModel`-Klasse vornehmen:

```
private Key searchKey;

public HashTableModel(int rows, int cols) {
   map = new HashMap();
   this.rows = rows;
   this.cols = cols;
   searchKey = new Key(0, 0);
}

...

public Object getValueAt(int row, int col) {
   return map.get(searchKey.set(row, col));
}

public void setValueAt(Object object, int row, int col) {
   if (object == null) {
      if (map.remove(searchKey.set(row, col)) != null) {
         fireTableCellUpdated(row, col);
      }
   }
   else {
      if (map.put(new Key(row, col), object) != object) {
         fireTableCellUpdated(row, col);
      }
   }
}
...
```

Das Ergebnis überzeugt: Der lesende Zugriff ist nun etwa doppelt so schnell wie in der Version zuvor.

Ein bisher nicht behandeltes Problem bleibt jedoch. `JTable` legt für jede Spalte automatisch ein `TableColumn`-Objekt an – selbst für die Spalten, die nicht zu sehen sind. Das heißt, ob wir wollen oder nicht: zu einem Modell mit 100.000 Spalten werden automatisch auch 100.000 `TableColumn`-Objekte angelegt. Es lohnt sich also, Methoden zum Ändern der Modellgröße zur Verfügung zu stellen oder die Modellgröße an die größten Indizes, mit denen `setValueAt()` aufgerufen wurde, zu koppeln. Genau das haben wir im Folgenden getan:

```
public void setValueAt(Object object, int row, int col) {
   if (object == null) {
      if (map.remove(searchKey.set(row, col)) != null) {
         fireTableCellUpdated(row, col);
      }
   }
   else {
      if (map.put(new Key(row, col), object) != object) {
         if (row >= rows && col < cols) {
            rows = row;
            if (col < cols) {
               // nur die Zeilenanzahl hat sich geändert, daher
               // reicht das Data-Changed-Event
               fireTableDataChanged();
            }
            else {
               // auch die Spaltenanzahl hat sich geändert, daher
               // müssen wir das Structure-Changed-Event auslösen
               cols = col;
               fireTableStructureChanged();
            }
         }
         else fireTableCellUpdated(row, col);
      }
   }
}
```

Somit haben wir den Speicherverbrauch des `TableModels` drastisch vermindert.

8.4 Caches

Caches[4] sind eine lange bewährte Strategie zum schnellen Zugriff auf Daten, die räumlich oder zeitlich nah beieinander gespeichert sind. Prinzipiell handelt es sich dabei um eine schnelle Datenstruktur, die vor eine größere und langsamere Datenstruktur geschaltet ist, um so die Mehrzahl der Zugriffe auf die langsamere Datenstruktur zu beschleunigen. Zurzeit bietet Java von sich aus kaum Unterstützung für Caches. Dies soll sich jedoch mit dem *JCache*-API ändern, das aus Java Specification Request 107[5] hervorgehen wird.

Ein Beispiel für räumliche Lokalität sind Festplattenzugriffe. Gewöhnlich werden nicht nur die Daten gelesen, die gerade verlangt wurden, sondern auch die folgenden Daten. Da das Lesen sehr weniger Daten oft genauso lange dauert wie das Lesen eines ganzen Blocks von Daten, macht es Sinn den Block zu lesen und darauf zu hoffen, dass die zu

4 Das Wort Cache stammt vom französischen *cacher* (verstecken), da Caches in der Regel für den Benutzer unsichtbar sind.

5 JSR 107: *http://www.jcp.org/jsr/detail/107.jsp.*

viel gelesenen Daten kurze Zeit später benötigt werden. Da Programmcode meist und Daten oft sequenziell organisiert sind, trifft diese Annahme auch häufig zu. In Java hat dies seine Entsprechung in gepufferten Ein-/Ausgabe-Strömen.

Ein Beispiel für zeitliche Lokalität ist das so genannte Workingset von Programmen. Gemeint sind jene Speicherseiten, auf die immer wieder zugegriffen wird, während andere Speicherseiten kaum benötigt werden. Die nicht benötigten Seiten werden daher aus dem Hauptspeicher auf einen langsameren Speicher ausgelagert (Swapping).

Gewöhnlich ist mit einem Cache ein gewisser Aufwand verbunden, da überprüft werden muss, ob sein Inhalt noch korrekt ist. Damit sich dieser Aufwand lohnt, muss der Datenzugriff im Schnitt schneller sein als ohne Cache. Um die Effektivität eines Caches zu messen, betrachtet man daher seine Trefferrate, d.h. wie viele von x Datenzugriffen vom Cache bedient werden konnten.

Da ein Cache per Definition nur eine begrenzte Speicherkapazität hat, muss er über eine Austauschstrategie verfügen. Diese besagt, nach welchen Regeln alte Daten aus dem Cache entfernt und durch neue ersetzt werden.

8.4.1 Austauschstrategien

Austauschstrategien versuchen Daten so im Cache zu speichern bzw. aus dem Cache zu entfernen, dass die Trefferrate möglichst hoch ist.

Zufälliger Austausch

Die einfachste Strategie ist der zufällige Austausch. Beim Lesen eines Datums, das noch nicht im Cache enthalten ist, wird zufällig ein anderes Element aus dem Cache entfernt. Obwohl diese Strategie simpel und naiv klingt, muss sie keinesfalls schlecht sein. Wenn nämlich der Zugriff auf die Daten ebenfalls zufällig erfolgt und keinerlei Lokalität aufweist, ist jede aufwändigere Austausch-Strategie vergebene Liebesmüh.

Am längsten nicht benutztes Element

Hierbei wird immer das Element aus der Datenstruktur überschrieben, das am längsten nicht benutzt wurde. Der gängige Name dieses Verfahrens ist *Least Recently Used* (*LRU*) Gewöhnlich wird eine verkettete Liste benutzt, die die Elemente in Ihrer Zugriffsordnung enthält. Wird auf ein Element zugegriffen, so wird es aus der Liste entfernt und am Anfang der Liste wieder eingefügt. War das Element noch nicht in der Liste enthalten, so wird es ebenfalls am Anfang eingefügt und das letzte Element wird entfernt, sofern die Kapazität des Caches bereits erreicht ist.

LRU ist die wohl am häufigsten benutzte Austauschstrategie, weil sie die Charakteristika eines Workingsets am besten abbildet und gewöhnlich zu sehr guten Ergebnissen führt, sofern die Datenzugriffe zeitliche Lokalität aufweisen.

Ältestes Element

Anstatt bei jedem Zugriff auf die Datenstruktur ein Element an den Anfang der Liste verschieben zu müssen, kann man beim Einfügen auch einfach das älteste Element der Liste entfernen, unabhängig davon, wie oft oder wann es benutzt wurde. Diese Strategie führt in der Regel zu nicht so guten Resultaten wie LRU, hat jedoch weniger Verwaltungsaufwand.

8.4.2 Elementspezifische Invalidierung

Oft müssen Elemente eines Caches nicht nur aus dem Cache entfernt werden, weil kein Platz mehr für neue Elemente vorhanden ist, sondern weil das Element nicht mehr den korrekten Wert hat oder ein anderes Ereignis eingetreten ist.

So kann es vorkommen, dass der Benutzer explizit verlangt, dass ein Element aus einem Cache entfernt wird. Ebenso ist es denkbar, dass in einer Föderation von Caches ein Cache einen anderen anweist, ein bestimmtes Element zu entfernen, weil es ungültig ist. Weitere Auslöser für Invalidierung sind das Ablaufen der Lebensdauer eines Elements (*Time to Live*) oder das Verstreichen einer Zeit ohne Zugriff auf das Element (*Idle Time*).

8.4.3 Schreibverfahren

Um veränderte Elemente von einem Cache in die darunter liegende Datenstruktur zurückzuschreiben, gibt es verschiedene Strategien. Zwei allgemein benutzte sind *Write-Through* und *Write-Back*.

Beim Write-Through-Verfahren wird jeder schreibende Datenzugriff direkt auf der darunter liegenden Datenstruktur ausgeführt, so dass Cache und Datenstruktur immer kohärent sind. Dieses Verfahren ist insbesondere sinnvoll bei systemkritischen Daten. Write-Through hat seine Entsprechung in der Methode `force()` der Klasse `java.nio.channels.FileChannel` (seit JDK 1.4) . Sie sorgt dafür, dass alle Daten, die in den Channel geschrieben wurden, auch tatsächlich auf den Datenträger geschrieben werden und nicht nur in einen Cache.

Write-Through ist zudem sinnvoll, wenn die Anzahl der lesenden Zugriffe weitaus größer ist als die der schreibenden Zugriffe.

Beim Write-Back-Verfahren werden Änderungen nur an die darunter liegende Datenstruktur propagiert, wenn das gecachte Element aus dem Cache entfernt wird. Alle anderen Änderungen erfolgen ausschließlich im Cache.

8.4.4 Gecachte Map

Gewöhnlich erfolgt der Zugriff auf einen Cache genauso wie auf eine Tabelle mit einem Schlüssel. Nun macht es wenig Sinn, eine `HashMap` zu cachen, da diese selbst gut als sehr einfacher Cache benutzt werden kann. Stattdessen wollen wir versuchen eine `TreeMap` zu cachen. Natürlich könnten wir dies tun, indem wir einfach sowohl eine `HashMap` als auch eine `TreeMap` pflegen und für jene Operationen, bei denen die Ordnung der Elemente keine Rolle spielt, die schnellere `HashMap` benutzen. Die `HashMap` hätte jedoch unbegrenzte Kapazität. Wir würden somit den Speicherbedarf ungefähr verdoppeln.

Stattdessen wollen wir eine eigene Klasse schreiben, die gecachten Zugriff auf eine `Map` ermöglicht und dabei das Zufallsaustauschverfahren benutzt. Als Basis benutzen wir dazu eine `RandomCache`-Klasse, die die gecachten Schlüssel-/Wert-Paare jeweils in einem `CacheEntry`-Objekt hält und diese wiederum in einem Array speichern. Als Index in den Array benutzen wir die letzten Bits des Hashcodes des Schlüssels. Aus Geschwindigkeitsgründen sorgen wir dafür, dass die Arraygröße jeweils eine volle Zweierpotenz ist. So können wir den schnelleren bitweisen Und-Operator `&` anstelle vom Modulo-Operator `%` benutzen, um beliebige ganze Zahlen auf einen Array-Index abzubilden. Für diese Abbildung ist der Hashcode wie geschaffen. Das ist auch nicht weiter verwunderlich – denn im Endeffekt ist unser Cache nichts anderes als eine Hashtabelle ohne Kollisionsstrategie. Listing 8.4 zeigt die Schnittstelle unserer Cache-Klasse, Listing 8.4 die Implementierung. Listing 8.5 stellt die Klasse dar, die eine `RandomCache`-Instanz zum Cachen einer `Map` benutzt. Die Klasse könnte übrigens leicht erweitert werden, so dass auch sie das komplette `java.util.Map`-Interface implementiert.

```
package com.tagtraum.perf.datastructures;

// Interface für Caches.
public interface Cache {

   // Gibt die Kapazität dieses Caches an. Die exakte Bedeutung
   // dieses Wertes ist implementierungsabhängig.
   public int getCapacity();

   // Gibt ein Objekt aus diesem Cache zurück, sofern es enthalten
   // ist, ansonsten null.
   public Object get(Object key);

   // Registriert ein Objekt unter einem Schlüssel in diesem Cache.
   public Object put(Object key, Object value);

   // Gibt eine Zahl zwischen 0.0 und 1.0 zurück. 1.0 entspricht
   // einer 100-prozentigen Trefferquote.
   public float getHitRatio();
}
```

Listing 8.3: Einfaches Cache-Interface

```
package com.tagtraum.perf.datastructures;

// Cache mit zufälliger Austauschstrategie
public class RandomCache implements Cache {

   private CacheEntry[] entries;
   private int bitMask;
   private int hits;
   private int misses;

   public RandomCache(int initialCapacity) {
      // Finde eine Zweierpotenz >= initialCapacity
      int capacity = 1;
      while (capacity < initialCapacity)
         capacity <<= 1;
      entries = new CacheEntry[capacity];
      // Initialisiere mit leeren Entries
      for (int i = 0; i < capacity; i++) {
         entries[i] = new CacheEntry();
      }
      bitMask = capacity - 1;
   }

   public int getCapacity() {
      return entries.length;
   }

   public Object get(Object key) {
      int index = key.hashCode() & bitMask;
      CacheEntry entry = entries[index];
      if (entry.sameKey(key)) {
         // Treffer
         hits++;
         return entry.getValue();
      }
      // kein Treffer
      misses++;
      return null;
   }

   public Object put(Object key, Object value) {
      if (key != null) {
         return entries[key.hashCode() & bitMask].set(key, value);
      }
      return null;
   }

   public float getHitRatio() {
      return ((float) hits) / (float) (hits + misses);
   }
```

```
    private static class CacheEntry {
        private Object key;
        private Object value;

        public boolean sameKey(Object other) {
            return key != null && key.equals(other);
        }

        public Object getValue() {
            return value;
        }

        public Object set(Object key, Object value) {
            Object oldValue = value;
            this.key = key;
            this.value = value;
            return oldValue;
        }
    }
}
```

Listing 8.4: Simple Cache-Klasse, die das Zufallsaustauschverfahren benutzt

```
package com.tagtraum.perf.datastructures;

import java.util.Collections;
import java.util.Map;

public class RandomMapCache {

    private Map map;
    private Map unmodifiableMapView;
    private RandomCache cache;

    public RandomMapCache(Map map, int initialCapacity) {
        this.map = map;
        unmodifiableMapView = Collections.unmodifiableMap(map);
        cache = new RandomCache(initialCapacity);
    }

    public int getCapacity() {
        return cache.getCapacity();
    }

    public Map getMap() {
        return unmodifiableMapView;
    }

    public Object get(Object key) {
        Object value = cache.get(key);
        if (value != null) {
```

```
            return value;
         }
         value = map.get(key);
         if (key != null) {
            cache.put(key, value);
         }
         return value;
      }

      public Object put(Object key, Object value) {
         Object oldValue = map.put(key, value);
         if (key != null) {
            cache.put(key, value);
         }
         return oldValue;
      }
   }
```

Listing 8.5: Cache-Klasse für Maps

Natürlich wollen wir uns nicht mit der bloßen Existenz der oben beschriebenen Klasse zufrieden geben, sondern auch einen kleinen Test durchführen. Gecached werden soll eine `TreeMap` mit 1.280 Elementen, die Cache-Kapazität soll 128 betragen und wir greifen wiederholt auf eine begrenzte Anzahl der in der `TreeMap` enthaltenen Elemente zu. Genauer gesagt, die ersten 32, 64, 128 und 256 Elemente.

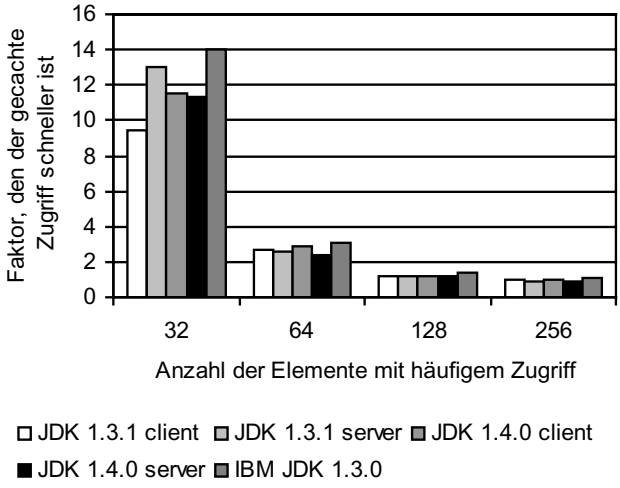

Abbildung 8.4: Vergleich zwischen einer ungecachten und einer gecachten TreeMap mit einer Cachegröße von 128 und einer Größe von 1.280 Elementen in Abhängigkeit von der Anzahl der Elemente, auf die regelmäßig zugegriffen wird

Abbildung 8.4 zeigt den Beschleunigungsfaktor für den Zugriff auf die `TreeMap` mit Cache. Offensichtlich sorgt der Cache nur für eine spürbare Beschleunigung, wenn die Cache-Kapazität mindestens doppelt so groß ist wie die Menge der häufig benutzten Elemente.

Abbildung 8.5 zeigt, warum dies so ist. Bei einer Cache-Kapazität von 128 und gleicher Anzahl häufig zugegriffener Elemente liegt die Cache-Trefferrate nur bei 34%. In 66% der Fälle musste auf die `TreeMap` zugegriffen werden.

Da wir den Hashcode als Basis für den Array-Index benutzen, steht die Trefferrate natürlich in direkter Beziehung zur Hashfunktion. Tatsächlich werden die Schlüssel der ersten 128 Elemente bei einer Cache-Kapazität von 128 durch die Hashfunktion `hashCode()` auf lediglich 43 Arraypositionen abgebildet – die restlichen 85 Positionen bleiben unbenutzt. Wesentlich besser sieht es dagegen bei 64 und 32 häufig zugegriffenen Elementen aus, daher auch der bessere Beschleunigungsfaktor.

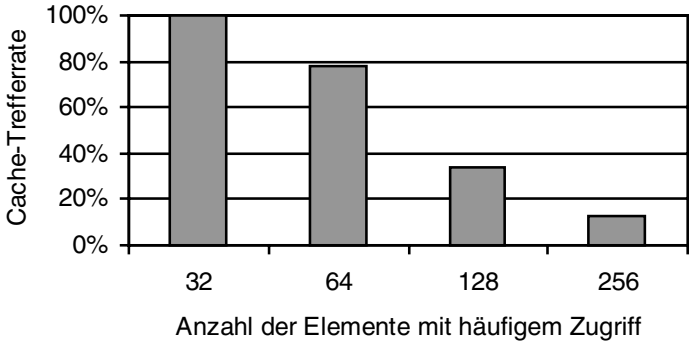

Abbildung 8.5: Trefferrate in Abhängigkeit von der Anzahl der Elemente, auf die regelmäßig zugegriffen wird

Für obigen Test habe ich die Schlüssel folgendermaßen erzeugt:

```
for (int i=0; i<keys.length; i++) {
   keys[i] = "key" + i;
}
```

Tests mit anderen Schlüsseln ergaben wesentlich bessere bzw. wesentlich schlechtere Resultate. Daher gilt:

Wenn Sie einen Cache einsetzen, messen Sie die Trefferrate und den Beschleunigungsfaktor. Überprüfen Sie zudem, ob die Methode `hashCode()` *für Ihre Schlüssel ausreichend gut verteilt ist.*

8.4.5 Caches mit LinkedHashMap

Seit JDK 1.4 gibt es die Klasse `java.util.LinkedHashMap`. Sie ist wie gemacht für LRU-Caches. Jedes Mal, wenn die Methoden `put()` oder `putAll()` benutzt werden, wird automatisch die Methode `removeEldestEntry()` aufgerufen. `removeEldestEntry()` gibt `true` zurück, wenn tatsächlich der älteste Eintrag der Map entfernt werden soll, was dann auch direkt anschließend passiert. Abhängig von einem booleschen Konstruktorparameter ist mit »ältestem Eintrag« dabei entweder das am längsten nicht benutzte Elemente oder das am längsten in der Map befindliche Element gemeint.

Um einen LRU-Cache zu schreiben müssen wir lediglich die Methode `removeEldestEntry()` überschreiben (Listing 8.6).

```
package com.tagtraum.perf.datastructures;

import java.util.LinkedHashMap;
import java.util.Map;

public class LRUCache extends LinkedHashMap {
   private int capacity;

   public LRUCache(int capacity) {
      // Propagiere Kapazität zur darunter liegenden HashMap,
      // so dass wir möglichst selten ein Rehashing benötigen.
      // Außerdem setzen wir die Ordnung der Map mit
      // true auf Zugriffs-Ordnung statt Einfüge-Ordnung.
      super((int)(capacity/0.75f), 0.75f, true);
      this.capacity = capacity;
   }

   // Gibt true zurück, wenn die Kapazität des Caches erreicht ist.
   protected boolean removeEldestEntry(Map.Entry eldest) {
      return size() > capacity;
   }

   public int getCapacity() {
      return capacity;
   }
}
```

Listing 8.6: LinkedHashMap basierter LRU-Cache

Abbildung 8.6 zeigt einen Vergleich zwischen dem oben vorgestellten Zufallscache und `LRUCache`. Im Test wird der Reihe nach wiederholt auf 32-256 Elemente zugegriffen.

Bei 256 Elementen und einer Cache-Kapazität von 128 wird beim LRU-Cache nur der zusätzliche Verwaltungsaufwand gemessen, da die Trefferrate gleich null ist (Abbildung 8.7). Bei 128 Elementen schneidet der LRU-Cache am besten ab. Die Trefferrate

liegt bei 100%. Damit ist aber auch die maximale Beschleunigung durch den `LRUCache` erreicht, während der Zufallscache für weniger Elemente schneller wird. Insbesondere für nur 32 Elemente ist der Zufallscache wesentlich schneller als der LRU-Cache. Mit anderen Worten:

Eine schlechte Austauschstrategie kann sich auszahlen, wenn sie sehr geringen Verwaltungsaufwand hat.

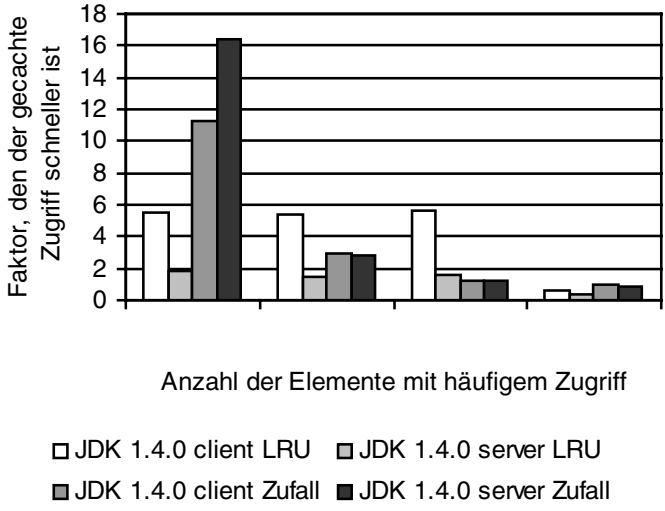

Abbildung 8.6: Vergleich zwischen einer LRU und einer mit Zufallsstrategie gecachten TreeMap mit einer Cachegröße von 128 und einer Größe von 1.280 Elementen in Abhängigkeit von der Anzahl der Elemente, auf die regelmäßig zugegriffen wird

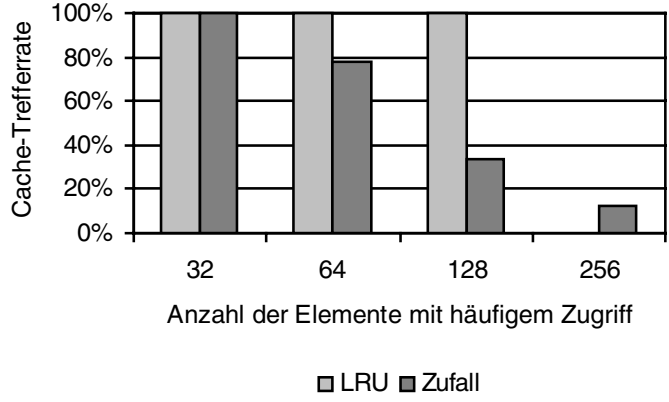

Abbildung 8.7: Trefferrate in Abhängigkeit von der Anzahl der Elemente, auf die regelmäßig zugegriffen wird, und der Austauschstrategie

8.4.6 Schwache Referenzen

Ein Objekt zu cachen heißt auch immer, Speicher dauerhaft zu belegen und so die automatische Speicherbereinigung zu verlangsamen. Oft ist es daher sinnvoll, dass ein Objekt, wenn es eine Weile nicht benutzt wurde, automatisch aus dem Speicher entfernt wird. Besonders einfach geht dies mit schwachen Referenzen aus dem Paket `java.lang.ref`.

Hier ein Beispiel für den entsprechenden Einsatz einer `SoftReference`:

```
private static SoftReference objectSoftReference = null;

public Object getObject() {
   Object object = null;
   if (objectSoftReference != null) {
      object = objectSoftReference.get();
   }
   if (object == null) {
      // Cachen des Objektes in einer SoftReference
      object = new Object();
      objectSoftReference = new SoftReference(object);
   }
   return object;
}

...
```

Auf das Objekt wird immer über die `getObject()`-Methode zugegriffen. Beim ersten Zugriff werden das Objekt und eine `SoftReference` auf dieses Objekt angelegt. Die `SoftReference` ist eine Klassen- oder Instanzvariable. Beim nächsten Aufruf von `getObject()` wird nun zunächst der Inhalt der `SoftReference` überprüft. Ist dieser ungleich `null`, wird das gecachte Objekt zurückgegeben. Ist er jedoch gleich `null`, wurde das zuvor erzeugte Objekt bereits von der Speicherbereinigung erfasst – das heißt wir müssen ein neues Objekt erzeugen.

Objekte, die durch eine `SoftReference` referenziert werden, können genau dann von der Speicherbereinigung erfasst werden, wenn sie nicht mehr durch eine normale Referenz referenziert werden. Die JDK-Dokumentation verspricht zudem, dass alle `SoftReferences` gelöscht werden, bevor die VM einen `OutOfMemoryError` auslöst. Das heißt, `SoftReferences` sind ideal für speicherempfindliche Caches.

Leider sind viele VMs jedoch übereifrig beim Löschen von `SoftReferences` und löschen diese ohne Not. Dies ist insbesondere der Fall bei Sun JDK 1.3.0. Spätere Versionen löschen `SoftReferences` erst einige Zeit nachdem die referenzierten Objekte nicht mehr durch normale Referenzen referenziert werden und erreichen so ein LRU-ähnliches Verhalten. Die Dauer dieser Gnadenfrist beträgt eine Sekunde pro freies Mbyte auf dem Heap. Die HotSpot-Server-Version benutzt für diese Berechnung die maximal

mögliche Heapgröße (VM Option `-Xmx`), während die Client-Version die aktuelle Heapgröße verwendet. Unabhängig davon, welche Version Sie benutzen, können Sie diese Frist manipulieren, indem Sie den VM-Parameter `-XX:SoftRefLRUPolicyMSPerMB` angeben:

```
java -XX:SoftRefLRUPolicyMSPerMB=2500 <Hauptklasse>
```

Dieser Parameter reguliert, wie viele Millisekunden pro freies Mbyte die Speicherbereinigung warten soll. Bitte beachten Sie, dass dies ein offiziell nicht unterstützter VM-Parameter ist. Er wird von Sun JDK 1.3.1 und 1.4.0 erkannt.

9 Threads

Java verfügt über eine sehr mächtige, einheitliche und vergleichsweise einfache Threadunterstützung. Richtig und mit Bedacht angewandt, kann sie zu sehr eleganten Designs führen. Gerade, wenn es um wahrgenommene Performance geht, ermöglichen Threads einfache Lösungen für ansonsten schwierige Probleme. Zudem sind Threads die Zutat, die dafür sorgt, dass Java Programme auf Mehrprozessormaschinen skalieren.

Doch Threads haben ihren Preis. Sie können die Komplexität von Programmen erheblich erhöhen. Die Programmausführung ist nicht mehr deterministisch und streng sequenziell, sondern parallel. Verschiedene Threads verhalten sich asynchron zueinander, es sei denn sie sind explizit synchronisiert. Und gerade korrekte Synchronisation ist nicht-trivial. Dies ist auch einer der Gründe, warum EJB von ihrem Container zwangssynchronisiert werden und keine weitergehende Synchronisierung erlaubt ist. Viele Fehler können so vermieden werden.

Zudem können Synchronisation und ständige Threadkontext-Wechsel zu schlechter Performance führen. Dies ist insbesondere der Fall, wenn mehr Threads als Prozessoren vorhanden sind. Dies, die erhöhte Komplexität und Nicht-Determinismus, sind Gründe, warum Swing nur einen Thread benutzt. Programme ohne Threads sind oft einfacher zu warten, schneller und Synchronisationsfehler sind nicht möglich. Daher gilt:

Wenn es keinen guten Grund für den Einsatz von Threads gibt, vermeiden Sie Threads lieber.

Wenn Sie jedoch glauben, dass Threads die Lösung Ihrer Probleme bedeuten, hilft Ihnen der Rest dieses Kapitels hoffentlich performanten Code zu schreiben.

9.1 Gefährlich lebt sich's schneller

Grundsätzlich gibt es im Zusammenhang mit Mehrthread-Programmen zwei wesentliche Zustände [vgl. Lea99, S. 38f.]:

- Sicherheit
  Der Zustand, in dem nichts Schlimmes passiert.
- Lebendigkeit
  Der Zustand, in dem überhaupt jemals etwas geschieht.

Die große Gefahr beim Programmieren mit mehreren Threads ist das gleichzeitige Manipulieren desselben Speicherbereichs durch zwei oder mehr Threads. Dies führt meistens zu unvorhersehbaren Ergebnissen, deren Ursache sehr schwierig herauszufinden ist. Will man dies verhindern, so muss man einem Thread exklusiven Zugriff auf eine Ressource zusichern. So erreichen Sie Sicherheit.

Meistens werden zu diesem Zweck jedoch andere Threads angehalten, die die Ressource ebenfalls manipulieren wollen. Natürlich sind Programme lebendiger, in denen immer alle Threads laufen.

Es ist offensichtlich, dass die Abwesenheit von Lebendigkeit zu Sicherheit führt. Oft, aber nicht zwingend, führt die Abwesenheit von Sicherheit auch zu Lebendigkeit.

Strategien für Sicherheit sind beispielsweise:

- Unveränderbarkeit (Immutability)
  Unveränderbare Objekte können ihren Zustand nicht ändern, daher können sie auch nicht manipuliert werden. Beispiel: `java.lang.String`
- (Vollständige) Synchronisation
  (Alle) Methoden, die den Objektzustand ändern, sind synchronisiert. Beispiel: `java.lang.StringBuffer()`
- Behälter (Containment)
  Auf das zu manipulierende Objekt kann nur durch ein anderes Objekt zugegriffen werden, das die Synchronisation übernimmt. Beispiel: `java.util.Collections.synchronizedMap(new HashMap())`

Wenn Sie ausschließlich diese drei Strategien verwenden, sind Sie auf der sicheren Seite. Leider heißt dies nicht, dass Ihr Programm auch nur einen Hauch von Leben in sich hat, von Performance ganz zu schweigen.

Die häufigsten Gründe hierfür sind:

- Verhungern (Starvation)
  Ein Thread wird nie ausgeführt, weil ein anderer Thread oder ein anderer Prozess sämtliche Prozessoren voll und ganz für sich in Anspruch nimmt.
- Dormancy
  Ein Thread wartet auf ein `resume()` oder `notify()`, das aber nie aufgerufen wird.
- Deadlock
  Thread *A* wartet auf ein Synchronisationsschloss, das Thread *B* besitzt, während Thread *B* auf ein Schloss wartet, das Thread *A* besitzt.

Im Folgenden werden wir uns mit den Kosten verschiedener Strategien zum Erreichen von Sicherheit befassen sowie einige andere Aspekte der Threadprogrammierung unter Performancegesichtspunkten betrachten.

9.1.1 Sicherheit durch Synchronisation

Um exklusiven Zugriff auf geteilten Ressourcen zu erlangen, können Sie in Java Programmteile mit einem `synchronized`-Block schützen. Dieser verhindert, dass derselbe Code gleichzeitig von mehr als einem Thread ausgeführt wird. Beachten Sie, dass `synchronized` keine Ressource schützt, sondern Code, der unter Umständen auf schützenswerte Ressourcen zugreift. Wollen Sie eine Ressource schützen, müssen Sie *jeglichen* Code, der diese Ressource manipuliert, mit einem `synchronized`-Block schützen, der mit demselben Objekt synchronisiert. Nicht mit `synchronized` geschützt werden müssen atomare Manipulationen. Das sind Operationen, die nicht vom Thread-Scheduler unterbrochen werden können. Dazu gehören Referenzzuweisungen sowie alle Zuweisungen von primitiven Datentypen außer `double` und `long`, es sei denn `double` oder `long` sind mit dem Schlüsselwort `volatile` gekennzeichnet [vgl. Gosling00, §17.4].

Beispiel:

```
private long time;
// Nicht threadsicher!
public void setTime(long time) {
   this.time = time;
}
```

Dieser Code ist *nicht* threadsicher, da das Zuweisen eines `longs` keine atomare Operation ist, sondern aus zwei 32-Bit-Operationen besteht. Wenn also zwei Threads gleichzeitig die Methode `setTime()` aufrufen, kann es passieren, dass `this.time` nachher als Wert die ersten 32 Bit vom einen Aufruf und die zweiten 32 Bit vom anderen Aufruf hat. Dies ließe sich durch `synchronized` verhindern:

```
private long time;
// Threadsicher!
public synchronized void setTime(long time) {
   this.time = time;
}
```

Bevor ein Thread die synchronisierte `setTime()`-Methode ausführen kann, muss er warten, bis kein anderer Thread diese Methode ausführt. `this.time` ist somit vor gleichzeitigem schreibenden Zugriff geschützt. Dieser Schutz ist jedoch nicht umsonst.

9.1.2 Synchronisationskosten

Bereits im Datenstrukturen-Kapitel (*Kapitel 8 Datenstrukturen und Algorithmen*) erwähnte ich, dass `java.util.Vector` etwas langsamer als `java.util.ArrayList` ist, weil `Vector` im Gegensatz zu `ArrayList` voll synchronisiert ist. Es stellt sich die Frage, wie viel ein `synchronized` tatsächlich kostet. Wir rufen daher die `setTime()`-Methode von Objekten der folgenden beiden Klassen wiederholt in einer Schleife auf und messen die Zeit.

```
// Klasse mit unsynchronisierter setTime()-Methode
class UnsynchronizedTime {
   private long time;

   public void setTime(long time) {
      this.time = time;
   }
}

// Klasse mit synchronisierter setTime()-Methode
class SynchronizedTime {
   private long time;

   public synchronized void setTime(long time) {
      this.time = time;
   }
}
```

Java VM	Synchronisiert	Unsynchronisiert	Faktor
Sun JDK 1.3.1 Client	100%	56%	1,79
Sun JDK 1.3.1 Server	65%	1,6%	40,41
Sun JDK 1.4.0 Client	110%	23%	4,69
Sun JDK 1.4.0 Server	65%	1,1%	58,91
IBM JDK 1.3.0	88%	12%	7,26

Tabelle 9.1: Normalisierte Ausführungszeiten der beiden setTime()-Methoden sowie der Faktor, den die unsynchronisierte Variante schneller war als die synchronisierte

Wie Tabelle 9.1 zeigt, ist die unsynchronisierte Variante in allen gemessenen VMs wesentlich schneller. Dies liegt jedoch nicht nur an den zusätzlichen Aufwänden der VM, die durch `synchronized` verursacht werden, sondern auch daran, dass bestimmte Optimierungen wegen des `synchronized`-Blocks nicht mehr durchführbar sind. Anders sind die riesigen Unterschiede zwischen der synchronisierten und der unsynchronisierten Variante in den Ergebnissen der beiden Sun-Server-VMs nicht zu erklären.

Daraus lässt sich schließen, dass es sich lohnen kann, dem Aufrufer die Synchronisation zu überlassen, anstatt selbst feingranular zu synchronisieren. Dieser Gedanke ist auch der Grund dafür, warum die Collection-Klassen alle unsynchronisiert sind. Selten entspricht die Granularität der notwendigen Synchronisation gerade dem Aufruf einer der Methoden der Collection-Klassen.

Zurück zu unserem kleinen Test. Wenn wir die Schleife, in der die `setTime()`-Methode des `UnsynchronizedTime`-Objektes aufgerufen wird, in einem `synchronized`-Block ausführen, wird sie beinahe genauso schnell ausgeführt wie die unsynchronisierte Variante und ist dennoch korrekt synchronisiert.

```
private UnsynchronizedTime unsynchronizedTime
   = new UnsynchronizedTime();

// Methode, in der setTime() in einer langen Schleife
// aufgerufen wird.
private void setUnsynchronizedTime() {
   for (int i = 0; i < 500000000; i++)
      unsynchronizedTime.setTime(i);
}

public void test() {
   // Aufuf der TestMethode und mit äußerer Synchronisation
   synchronized (unsynchronizedTime) {
      setUnsynchronizedTime();
   }
}
```

Ein etwas allgemeineres Beispiel. Nehmen wir einmal an, Sie schreiben eine Klasse `A`, die über zwei Methoden verfügt: `methode1()` und `methode2()`. Beide Methoden müssen synchronisiert sein, da sie eine von mehreren Threads geteilte Ressource manipulieren. Die Implementierung sieht folgendermaßen aus.

```
package xyz;
public class A {
   public synchronized void methode1() {
      // mache etwas
   }
   public synchronized void methode2() {
      // mache etwas anderes
   }
}
```

Nach einiger Zeit stellen Sie fest, dass häufig zunächst `methode1()` und direkt anschließend `methode2()` vom selben Thread ausgeführt wird. Daher schreiben Sie den Code ein wenig um und bieten eine Methode `methode1und2()` an, die jeweils unsynchronisierte Versionen von `methode1()` und `methode2()` aufruft. `methode1und2()` muss nur einmal statt zweimal einen `synchronized()`-Block betreten. Dafür muss bei jedem Methodenaufruf noch eine Extra-Methode aufgerufen werden. Da wir aber eine moderne VM benutzen, können wir annehmen, dass dieser zusätzliche Methodenaufruf vom VM-Compiler durch Inlining wegoptimiert wird. Ist dies der Fall, dann ist folgender Code schneller:

```
package xyz;
public class A {
   public synchronized void methode1() {
      unsyncMethode1();
   }
   private void unsyncMethode1() {
      // mache etwas
```

```
    }
    public synchronized void methode2() {
       unsyncMethode2();
    }
    private void unsyncMethode2() {
       // mache etwas anderes
    }
    // schneller als methode1() und methode2()
    private synchronized void methode1und2() {
       unsyncMethode1();
       unsyncMethode2();
    }
}
```

Als Nächstes stellen Sie fest, dass beide Methoden häufig von Instanzen einer Klasse `B` aufgerufen werden, die sich auch im Paket `xyz` befindet. Da Sie volle Kontrolle über beide Klassen haben und diese beiden Klassen sehr eng miteinander gekoppelt sind, entscheiden Sie sich dazu, Klasse `A` etwas zu öffnen und die Methoden `unsyncMethode1()` und `unsyncMethode2()` als `package`-privat zu deklarieren. Somit hat `B` unbeschränkten Zugriff auf `unsyncMethode1()` und `unsyncMethode2()`.

```
package xyz;
public class A {
   public synchronized void methode1() {
      unsyncMethode1();
   }
   // Jetzt package-privat!
   void unsyncMethode1() {
      // mache etwas
   }
   public synchronized void methode2() {
      unsyncMethode2();
   }
   // Jetzt package-privat!
   void unsyncMethode2() {
      // mache etwas anderes
   }
   // schneller als methode1() und methode2()
   private synchronized void methode1und2() {
      unsyncMethode1();
      unsyncMethode2();
   }
}

// Klasse B liegt im selben Paket wie Klasse A
package xyz;
class B {
   private A a;
   ...
   public void method3(int count) {
```

```
        synchronized(a) {
           for (int i=0; a<count; i++) a.unsyncMethode1();
           a.unsyncMethode2();
           a.unsyncMethode1();
           a.unsyncMethode1();
        }
     }
  }
```

Sie erlauben so dem HotSpot-Server-Compiler größere Code-Blöcke zu optimieren und erreichen eventuell eine bessere Performance. Dabei müssen Sie jedoch zwei Dinge bedenken:

1. Dies macht nur Sinn, wenn Sie *absolute* Kontrolle über alle beteiligten Klassen haben. Ist dies nicht der Fall, und Sie wollen trotzdem unsynchronisierten Zugriff auf die Klasse zulassen, verfahren Sie lieber nach dem Alles-Oder-Nichts-Prinzip. Entweder die öffentlichen Methoden der Klasse sind threadsicher oder sie sind es nicht. Teilweise threadsichere Klassen, sofern es so etwas überhaupt geben kann, führen zu Fehlern, die sehr schwierig zu finden sind.

2. Je größer `synchronized`-Blöcke sind, desto länger müssen andere Threads darauf warten, selbst einen `synchronized`-Block auszuführen, der mit demselben Objekt synchronisiert ist. Dies kann die Lebendigkeit verringern statt vergrößern.

9.1.3 Threadsichere Datenstrukturen

Bereits im Datenstrukturen-Kapitel klang an, dass das Synchronisieren mittels eines Synchronisationswrappers ziemlich grob ist. Falls Sie beispielsweise eine verlinkte Liste als Warteschlange benutzen, manipulieren Sie diese gewöhnlich nur am Anfang und am Ende. Sofern die Länge Ihrer Warteschlange größer eins ist, können Sie sie auch mit zwei verschiedenen Objekten synchronisieren – einem für den Anfang und einem fürs Ende. So erhöhen Sie auf einfachste Weise die Lebendigkeit.

Es gibt für viele Datenstrukturen maßgeschneiderte Synchronisationsstrategien. Eine ganz ausgezeichnete Sammlung von threadsicheren und dennoch lebendigen Datenstrukturen sowie anderen threadbezogenen Hilfsklassen bietet das freie `util.concurrent`-Paket von Doug Lea (*http://gee.cs.oswego.edu/dl/classes/EDU/oswego/cs/dl/util/concurrent/intro.html*). Wenn Ihre Software Lebendigkeitsprobleme im Zusammenhang mit grob sperrenden Datenstrukturen hat, sei Ihnen dieses Paket samt Doug Leas Buch *Concurrent Programming in Java™ – Design Principles and Patterns* [Lea99] wärmstens empfohlen. Alternativ gibt es noch ein kommerzielles Thread-Werkzeugklassen-Paket namens *JThreadKit* (*http://www.jthreadkit.com/*). In einer der nächsten Versionen des JDK wird es zudem ein Paket namens `java.util.concurrent` geben, das entsprechende Klassen definiert. Die dazugehörige Spezifikationsanforderung ist JSR 166 (*http://www.jcp.org/jsr/detail/166.jsp*).

9.1.4 Double-Check-Idiom

Wir kommen nun zu einem ganz anderen Thema. Nämlich wie Sie *nicht* um Synchronisation herumkommen. Gerade für Singletons [vgl. Gamma96, S.139] wird gerne eine besondere Form der *späten Initialisierung* (*lazy Initialization*) verwendet. Oft sieht der Code dazu folgendermaßen aus:

```
01 private static Singleton instance;
02 private Singleton() {
03 }
04 // Tun Sie dies nicht! Es funktioniert nicht!
05 public static Singleton getInstance() {
06    if (instance == null) {
07       synchronized (Singleton.class) {
08          if (instance == null) {
09             instance = new Singleton();
10          }
11       }
12    }
13    return instance;
14 }
```

Dies ist das so genannte *Double-Check-Idiom*. Man geht dabei davon aus, dass, sobald die Variable `instance` ungleich `null` ist, diese für alle Threads einen gültigen Wert hat. Daher wird der `synchronized`-Block in diesem Fall erst gar nicht betreten. Davon verspricht man sich Performance-Vorteile. Leider hat `instance`, wenn es ungleich `null` ist, nicht immer für alle Threads einen gültigen Wert ... [vgl. Pugh01]

Grundsätzlich arbeiten Threads in ihrem eigenen Speicher mit Kopien von Werten aus dem Hauptspeicher. Diese Kopien dürfen nicht ohne Anlass wieder in den Hauptspeicher geschrieben werden [Gosling00, §17.3]. Der Beginn oder das Ende eines `synchronized`-Blocks ist beispielsweise ein solcher Anlass. So wird das `instance`-Objekt zwar spätestens in Zeile 11 für andere Threads sichtbar, das garantiert aber nicht, dass auch alle von `instance` referenzierten Objekte schon wieder in den Hauptspeicher zurückkopiert worden sind. Erst nach Zeile 11 ist dies garantiert. Es gibt also eine Zeitspanne, in der `instance` zwar ungleich `null`, aber noch nicht komplett mit dem Hauptspeicher bzw. den anderen Threads synchronisiert ist. Wenn `instance` in diesem Zustand benutzt wird, kann dies zu sehr schwierig zu findenden Fehlern führen.

Folgender Code löst das Problem, da nicht nur beim schreibenden, sondern auch beim lesenden Zugriff auf `instance` synchronisiert wird:

```
private static Singleton instance;

private Singleton() {
}
```

```
public static synchronized Singleton getInstance() {
   if (instance == null) {
      instance = new Singleton();
   }
   return instance;
}
```

Grundsätzlich gilt, dass Sie Variablen, die von mehreren Threads benutzt werden, sowohl beim Schreiben als auch beim Lesen durch `synchronized` *schützen müssen.*

Alternativ zur Synchronisierung können Sie auch das *Initialize-on-Demand-Holder-Class*-Idiom [Bloch02, S.196] benutzen. Dabei macht man sich zunutze, dass eine Klasse erst initialisiert wird, wenn sie das erste Mal benutzt wird:

```
private static Singleton instanceHolder;

private Singleton() {
}

public static Singleton getInstance() {
   return SingletonHolder.instance;
}

private static class SingletonHolder {
   static final Singleton instance = new Singleton();
}
```

Java VM	Synchronisiert	Holderclass	Faktor
Sun JDK 1.3.1 Client	100%	37%	2,68
Sun JDK 1.3.1 Server	78%	nicht messbar	sehr hoch
Sun JDK 1.4.0 Client	110%	39%	2,86
Sun JDK 1.4.0 Server	77%	nicht messbar	sehr hoch
IBM JDK 1.3.0	104%	8%	12,64

Tabelle 9.2: Normalisierte Ausführungszeiten der synchronisierten und Holderclass-Variante sowie der Faktor, um den die Holderclass-Variante schneller ist

In den meisten Fällen ist die Holderclass-Variante sehr viel schneller als die synchronisierte Version. Die Sun-Server-VMs profitieren zudem stark davon, dass in der Holderclass-Variante kein `synchronized`-Block verwendet wird, und optimieren den Code so weit, dass der Test ad absurdum geführt wird. Es gilt:

Benutzen Sie auf keinen Fall das Double-Check-Idiom. Zudem sollten Sie der korrekt synchronisierten Variante das Initialize-on-Demand-Holder-Class-Idiom vorziehen.

9.1.5 Sprunghafte Variablen

Anstatt Variablen zwischen Threads durch `synchronized`-Blöcke zu synchronisieren, können Sie diese auch als `volatile` deklarieren. Dadurch wird die VM dazu genötigt, vor und nach jedem Benutzen dieser Variablen den Thread-Speicher mit dem Hauptspeicher abzugleichen. Zudem sind Zuweisungsoperationen von `longs` und `doubles`, die `volatile` sind, atomar. Es lohnt sich jedoch nur `volatile` zu benutzen, wenn Sie die Variable relativ selten manipulieren, da sonst ständige, zeitraubende Abgleiche mit dem Hauptspeicher erfolgen.

Vermutlich wird sich die genau Definition von `volatile` in Zukunft ändern, da §17 der Java Sprachspezifikation [Gosling00] im Rahmen der Java Spezifikationsanforderung 133 überarbeitet wird. Details hierzu können Sie im Web finden.

- Java Specification Request 133: *http://jcp.org/jsr/detail/133.jsp*

9.2 Allgemeine Threadprogrammierung

Ganz ohne Frage: Threads sind nicht wie alle anderen Java-Objekte. Dies soll für uns Grund genug sein, einige spezielle Aspekte ein wenig genauer zu beleuchten.

9.2.1 Threads starten

Einen Thread zu starten dauert etwas länger als ein normales Objekt zu instanziieren, da außer dem Objekt noch ein Java Virtual Machine Stack angelegt werden muss. Wir wollen testen, wie groß der Unterschied ist. Zu diesem Zweck führen wir die folgenden drei Methoden jeweils mehrmals nacheinander aus:

```
// Thread instanziieren und starten
private Object startThread() throws InterruptedException {
   Thread t = new Thread();
   t.start();
   return t;
}

// Thread instanziieren
private Object instantiateThread() {
   return new Thread();
}

// Objekt instanziieren
private Object instantiateObject() {
   return new Object();
}
```

Java VM	startThread()	instantiateThread()	instantiateObject()
Sun JDK 1.3.1 Client	100%	9%	nicht messbar
Sun JDK 1.3.1 Server	97%	7%	nicht messbar
Sun JDK 1.4.0 Client	112%	10%	nicht messbar
Sun JDK 1.4.0 Server	122%	8%	nicht messbar
IBM JDK 1.3.0	106%	7%	nicht messbar

Tabelle 9.3: Normalisierte Ausführungszeit der drei verschiedenen Methoden

Wie Tabelle 9.3 zeigt, dauert es rund zehnmal länger, einen Thread zu instanziieren und zu starten als ein `Thread`-Objekt nur zu instanziieren. Ein einfaches Objekt zu instanziieren führte beim 100.000fachen Aufruf lediglich zu Zeiten zwischen 10 und 50 Millisekunden. Zum Vergleich: Die Methode `startThread()` führte auf der Testmaschine beim 100.000fachen Aufruf zu Messzeiten von etwa 43 Sekunden.

Während es bei anderen Objekten meist nicht lohnt, so ist das Vorhalten von Thread-Objekten in einem Pool eine Ausnahme.

9.2.2 Threadpool

Objekt-Pools sind mittlerweile eher schädlich für die Performance eines lang laufenden Programms. Sie führen zu Objekten, die in der älteren Generation des Heaps gehalten werden. Da der Garbage Collection-Algorithmus für die ältere Generation meist auf Speicherverbrauch und Lokalität optimiert ist und nicht so sehr auf Geschwindigkeit wie der der jungen Generation, führen viele langlebige Objekte zu langen Speicherbereinigungszeiten. Es ist daher nicht unbedingt empfehlenswert, Objekt-Pools anzulegen. Eine Ausnahme hiervon sind Objekte, deren Erstellen – aus welchen Gründen auch immer – sehr aufwändig ist. Dies können neben Threads auch Objekte sein, die sehr groß sind oder bei der Initialisierung aufwändige Berechnungen erfordern (siehe *Kapitel 11.1.1 Datenmenge verkleinern*).

Die Klasse `Runner` (Listing 9.1) ist in der Lage, ein `java.lang.Runnable`-Objekt in einem Thread auszuführen und diesen Thread nach Beendigung der `Runnable.run()`-Methode für das nächste `Runnable`-Objekt wiederzuverwenden. Anstelle von normalen `Thread`-Objekten werden `RunnerThread`-Objekte (Listing 9.2) benutzt, die sich nach dem asynchronen Ausführen der `Runnable.run()`-Methode beim `Runner`-Objekt zur Wiederverwendung zurückmelden. Das Sequenzdiagramm in Abbildung 9.1 zeigt das Zusammenspiel der Objekte, wenn sich bereits `RunnerThread`-Objekte zurückgemeldet haben.

Ein Klient des `runner`-Objektes ruft die `run()`-Methode auf und übergibt das asynchron auszuführende Objekt `Runnable`. Aus dem Stack `stack` wird der oberste `RunnerThread` namens `rt` genommen und mit `setRunnable(runnable)` initialisiert. Anschließend wird die `work()`-Methode von `rt` aufgerufen, die mittels `mutex.notify()` den `rt`-eigenen Thread

benachrichtigt, der auf das `mutex`-Objekt wartet. `mutex` ist eine private Instanzvariable von `rt` und wird ausschließlich zum Synchronisieren der `RunnerThread`-Aktivität benutzt. Der einzige Thread, der jemals ein `mutex.wait()` aufrufen kann, ist ebenfalls ein privates Attribut von `rt`. Wenn `mutex.notify()` aufgerufen wird, können wir sicher sein, dass der Thread von `rt` gerade innerhalb der `run()`-Methode von `rt` genau darauf wartet. Dies resultiert in der asynchronen Ausführung der `run()`-Methode von `rt` und führt somit zur asynchronen Ausführung von `runnable.run()`. Anschließend registriert sich `rt` wieder bei `runner`, ruft `mutex.wait()` auf und wartet auf das nächste `mutex.notify()`.

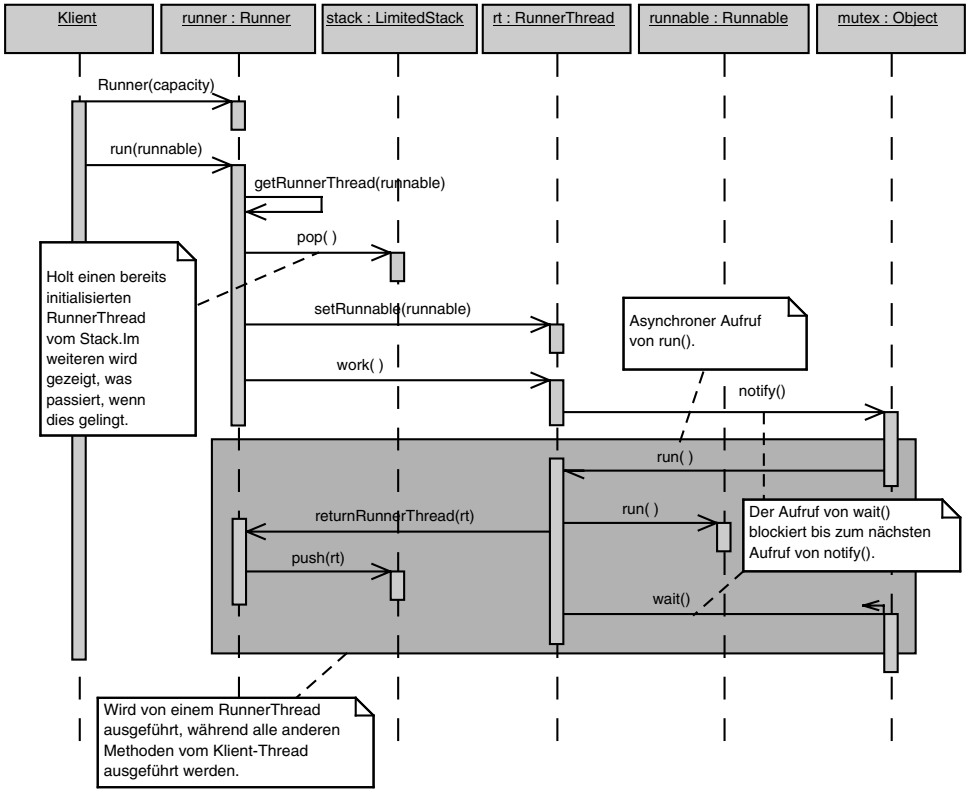

Abbildung 9.1: Sequenzdiagramm des Zusammenspiels von Runner, RunnerThread und einem Runnable-Objekt

Die Methoden `work()` und `run()` sind beide über `mutex` synchronisiert. Dies stellt sicher, dass `mutex.notify()` nicht aufgerufen wird, bevor `mutex.wait()` aufgerufen wurde. Dies ist essentiell, da `rt` sich zunächst bei `runner` zurückmeldet und dann erst `mutex.wait()` aufruft. Wäre dieser Block nicht mit `work()` synchronisiert, könnte sich ein `RunnerThread`-Objekt bei `runner` zurückmelden und `runner` könnte `work()` und somit `mutex.notify()` aufrufen, noch bevor `mutex.wait()` aufgerufen wurde. Dies würde dazu führen, dass der Thread des `RunnerThread`-Objektes ewig auf ein `mutex.notify()` wartete (Dormancy).

Der Code in Listing 9.1 und Listing 9.2 ist wegen der nötigen Poolverwaltung noch ein wenig komplizierter. Als Datenstruktur benutzen wir einen Stack, da dieser über die *FILO*-Eigenschaft verfügt. FILO steht für *First in, last out* und bedeutet, dass das zuletzt hinzugefügte Element als erstes wieder entnommen wird. Das heißt in unserem Fall, dass der zuletzt benutzte `RunnerThread` als erster wiederverwendet wird. Dies erhöht die Chance, dass er sich noch im Hauptspeicher befindet und noch nicht vom Betriebssystem ausgelagert wurde.

Natürlich wollen wir beliebig viele `Runnable`-Objekte ausführen können[1], aber nur eine begrenzte Anzahl im Pool halten. Unsere Stack-Implementierung, `LimitedStack`, hat daher die spezielle Eigenschaft, dass sie nicht mehr als eine vorgegebene Anzahl an Elementen halten kann. Dabei gibt die `push()`-Methode zurück, ob das Element erfolgreich auf den Stack gelegt werden konnte oder nicht. Dies ist wichtig für uns, da ein `RunnerThread`-Objekt, das nicht auf den Stack gelegt werden kann, beendet werden muss. Dies geschieht durch einen `quit()`-Aufruf in der `run()`-Methode von `RunnerThread`. Wäre dies nicht der Fall, würde das `RunnerThread`-Objekt `mutex.wait()` aufrufen, endlos auf ein `mutex.notify()` warten (Dormancy) und somit unnötig Ressourcen belegen.

Aus einem ähnlichen Grund benötigen wir die `Runner.destroy()`-Methode. Stellen Sie sich vor, Sie benutzen in einem Teil Ihrer Anwendung einen Thread-Pool mit einer Kapazität von 200 Threads. Dieser Teil der Applikation wird einmal am Tag für eine halbe Stunde ausgeführt. Die restlichen 23½ Stunden wird der Pool nicht benötigt. Wäre keine `destroy()`-Methode vorhanden, würden potenziell 200 Threads für 98 Prozent der Laufzeit sinnlos Ressourcen belegen.

```
package com.tagtraum.perf.threads;

public class Runner {

   private LimitedStack stack;
   private Class runnerThreadClass;
   private boolean destroyed;

   public Runner(int capacity) {
      this.stack = new LimitedStack(capacity);
      setRunnerThreadClass(RunnerThread.class);
   }

   // Führt die run()-Methode des Runnables asynchron mit einem
   // RunnerThread aus.
   public void run(Runnable runnable) {
      getRunnerThread(runnable).run();
   }
```

1 Es gibt sicherlich auch Fälle, in denen es sinnvoll ist, die Anzahl gleichzeitig ausgeführter Threads zu begrenzen. Dies würde das Beispiel aber noch komplizierter machen, als es ohnehin schon ist. Daher wollen wir darauf verzichten.

```
// Gibt einen initialisierten RunnerThread zurück. Der
// RunnerThread stammt entweder vom Stack oder wurde
// neu instanziiert.
protected synchronized RunnerThread
      getRunnerThread(Runnable runnable) {
   if (isDestroyed())
      throw new IllegalStateException("Runner is destroyed.");
   RunnerThread rt = (RunnerThread) stack.pop();
   if (rt == null) {
      rt = newRunnerThread();
      // Registriert den RunnerThread bei diesem Runner
      rt.setRunner(this);
   }
   rt.setRunnable(runnable);
   return rt;
}

// Instanziiert einen neuen RunnerTread.
protected RunnerThread newRunnerThread() {
   try {
      return (RunnerThread) getRunnerThreadClass()
         .newInstance();
   } catch (Exception e) {
      throw new InternalError(e.toString());
   }
}

public Class getRunnerThreadClass() {
   return runnerThreadClass;
}

protected void setRunnerThreadClass(Class runnerThreadClass) {
   this.runnerThreadClass = runnerThreadClass;
}

// Wird vom RunnerThread aufgerufen, sobald die run()-Methode
// des Runnables des RunnerThreads ausgeführt wurde.
synchronized boolean returnRunnerThread(RunnerThread rt) {
   // Nimmt den RunnerThread und legt ihn auf den Stack, sofern
   // der Runner noch nicht zerstört ist und der RunnerThread
   // noch lebendig ist. Gibt true zurück, wenn der RunnerThread
   // erfolgreich auf den Stack gelegt wurde.
   return rt.isAlive() && !destroyed && stack.push(rt);
}

public synchronized boolean isDestroyed() {
   return destroyed;
}

// Entfernt und stoppt alle RunnerThreads, die auf dem Stack
// liegen. Setzt anschließend das Flag destroyed. Dadurch wird
// verhindert, dass gerade laufende RunnerThreads sich wieder
```

```
    // bei Runner registrieren und so lebendig gehalten werden.
    public synchronized void destroy() {
       if (destroyed)
          throw new IllegalStateException("Runner is already"
             + " destroyed.");
       clear();
       destroyed = true;
    }

    // Entfernt und stoppt alle RunnerThreads, die auf dem Stack
    // liegen. Runner bleibt nach Aufruf dieser Methode weiterhin
    // benutzbar. Eventuell noch laufende RunnerThreads können sich
    // nach Aufruf dieser Methode wieder bei Runner zurückmelden.
    // Das heißt, dass der Stack nach Aufruf dieser Methode nicht
    // unbedingt leer sein muss, es sei denn es wurde mit this
    // synchronisiert.
    public synchronized void clear() {
       for (RunnerThread rt=stack.pop(); rt!=null; rt=stack.pop()) {
          rt.quit();
       }
    }

    // Stack, der nur eine bestimmte Anzahl an Elemente halten kann.
    private static class LimitedStack {
       private RunnerThread[] values;
       private int size;

       public LimitedStack(int capacity) {
          values = new RunnerThread [capacity];
       }

       public RunnerThread pop() {
          if (size == 0) return null;
          RunnerThread rt = values[--size];
          // Täten wir Folgendes nicht, würde dies zu einem
          // Speicherloch führen!
          values[size] = null;
          return rt;
       }

       // Legt ein Objekt auf den Stack, es sei denn er ist voll.
       // Ist dies der Fall, wird false zurückgegeben, sonst true.
       public boolean push(RunnerThread rt) {
          if (size == values.length - 1) return false;
          values[size++] = rt;
          return true;
       }
    }
}
```

Listing 9.1: Runner- und LimitedStack-Klassen

```
package com.tagtraum.perf.threads;

class RunnerThread implements Runnable {

   private static int count;
   private final Object mutex = new Object();
   private Runner runner;
   private Runnable runnable;
   private boolean running;
   private boolean working;
   private Thread thread;

   RunnerThread() {
      thread = new Thread("Runner-" + count++);
      // Sicherstellen, dass die VM terminieren kann, auch wenn
      // runner.destroy() nicht aufgerufen wurde.
      thread.setDaemon(true);
   }

   // Startet den Thread bzw. benachrichtigt ihn, dass er mit der
   // Ausführung fortfahren soll.
   public void work() {
      synchronized (mutex) {
         if (thread.isAlive()) {
            mutex.notify();
         } else {
            thread.start();
         }
      }
   }

   // Setzt running auf false und benachrichtigt den Thread, dass
   // er mit der Ausführung fortfahren soll. Dadurch wird er
   // beendet.
   public void quit() {
      synchronized (mutex) {
         running = false;
         mutex.notify();
      }
   }

   // Hauptschleife des RunnerThreads
   public void run() {
      synchronized (mutex) {
         try {
            running = true;
            while (running) {
               working = true;
               // Führt run() von runnable aus.
               runnable.run();
```

```
                working = false;
                // Stellt sicher, dass wir uns kein
                // Speicherloch einfangen.
                runnable = null;
                if (runner.returnRunnerThread(this)) {
                   mutex.wait();
                } else {
                   // Da wir this nicht wieder bei
                   // runner registrieren konnten, müssen wir quit()
                   // aufrufen, um belegte Ressourcen wieder
                   // freizugeben
                   quit();
                }
             }
          } catch (InterruptedException ie) {
             // Kann eigentlich nicht passieren, da
             // thread privat ist.
             ie.printStackTrace();
             quit();
          }
       }
    }

    // Wird von Runner vor work() aufgerufen.
    void setRunnable(Runnable runnable) {
       this.runnable = runnable;
    }

    // Wird von Runner aufgerufen.
    void setRunner(Runner runner) {
       this.runner = runner;
    }

    public boolean isAlive() {
       return thread.isAlive();
    }
}
```

Listing 9.2: RunnerThread-Klasse

Um zu vergleichen, ob das Benutzen von `Runner` sich wirklich positiv auswirkt, führen wir einen Test durch, in dem wir die Methode `runRunnable()` mehrfach ausführen. Die Methode ist so angelegt, dass sie mit der oben bereits beschriebenen `startThread()`-Methode vergleichbar ist.

```
private Runner runner = new Runner(100);

private Object runRunnable() {
   runner.run(new DummyRunnable());
   return null;
```

```
}

private static class DummyRunnable implements Runnable {
   public void run() {}
}
```

Wie Tabelle 9.4 zeigt, ist im beschriebenen Testfall die Methode `runRunnable()` mindestens sechsmal schneller als `startThread()`.

Java VM	startThread()	runRunnable()	Faktor
Sun JDK 1.3.1 Client	100%	15%	6,54
Sun JDK 1.3.1 Server	97%	13%	7,75
Sun JDK 1.4.0 Client	112%	16%	7,09
Sun JDK 1.4.0 Server	122%	10%	12,50
IBM JDK 1.3.0	106%	11%	9,85

Tabelle 9.4: Normalisierte Ausführungszeit der Methode startThread() und runRunnable() sowie der Faktor, um den runRunnable() schneller ist als startThread()

9.2.3 Kommunikation zwischen Threads

Oft muss ein Thread darauf warten, dass ein bestimmter Zustand eintritt. Folgender, naiver Code ist ein Negativ-Beispiel dafür, wie man dies anstellen kann.

```
volatile boolean condition = false;

public void waitForCondition() {
   // Furchtbarer Code! Tun Sie dies nicht!
   while (condition) {
      // Busy Wait
   }
}
```

Dieses Idiom wird auch *Busy Wait* genannt. Der aufrufende Thread verbleibt so lange in der `waitForCondition()`-Methode, bis ein anderer Thread die Variable `condition` auf `true` setzt. Abgesehen davon, dass Sie während des Wartens jede Menge CPU-Zyklen verbrauchen, kann es sein, dass nie ein anderer Thread zum Zuge kommt. Sie warten also vergebens.

Korrekt würde der Code etwa so lauten:

```
boolean condition = true;

public synchronized void waitForCondition()
      throws InterruptedException {
   while (condition) {
      wait();
```

```
    }
  }

  public synchronized void setCondition(boolean condition) {
    if (this.condition != condition) {
      this.condition = condition;
      // Alle wartenden Threads verständigen.
      if (!condition) notifyAll();
    }
  }
```

Anstatt also ohne Pause in einer Schleife eine Bedingung zu überprüfen, warten wir auf eine Benachrichtigung, überprüfen die Bedingung und warten gegebenenfalls auf die nächste Benachrichtigung oder fahren mit der Ausführung fort.

In obigem Code verwenden wir `notifyAll()`, um alle Threads zu benachrichtigen, die auf dasselbe Objekt warten. Falls Sie nur einen der wartenden Threads verständigen wollen, können Sie statt `notifyAll()` die `notify()`-Methode verwenden. Jedoch sollten Sie dann auch nicht mehr über `this`, sondern über ein eigenes Objekt synchronisieren. Schließlich können Sie nicht wissen, welche anderen Klassen Ihr Objekt ebenfalls zum Synchronisieren benutzen. Am Ende benachrichtigen Sie irgendeinen Thread, jedoch keinen, der `wait()` in `waitForCondition()` aufgerufen hat.

```
boolean condition = true;
static final Object lock = new Object();

public void waitForCondition()
      throws InterruptedException {
  synchronized (lock) {
    while (condition) {
      lock.wait();
    }
  }
}

public void setCondition(boolean condition) {
  synchronized (lock) {
    if (this.condition != condition) {
      this.condition = condition;
      // Einen wartenden Thread verständigen.
      if (!condition) lock.notify();
    }
  }
}
```

Die `notify()`-Methode ist gewöhnlich schneller als `notifyAll()`, wenn mehr als ein Thread benachrichtigt werden könnte. Das heißt aber auch, dass die Semantik eine ganz andere ist. Seien Sie also vorsichtig, wenn Sie hier versuchen zu optimieren. Gewöhnlich ist man mit `notifyAll()` auf der sicheren Seite. Noch einmal:

- `notify()` benachrichtigt *irgendeinen* Thread, der die `wait()`-Methode desselben Objekts aufgerufen hat. Die Laufzeit ist daher ***O(c)***.
- `notifyAll()` benachrichtigt *alle* Threads, die die `wait()`-Methode desselben Objekts aufgerufen haben. Die Laufzeit ist daher ***O(n)***.

9.2.4 Warten oder schlafen?

Es gibt keinen Performance-Unterschied zwischen `object.wait(time)` und `Thread.sleep(time)`. Jedoch gibt es einen semantischen Unterschied, der zu Laufzeit-Problemen führen kann.

Wenn Sie `Thread.sleep(time)` ausführen, behält der ausführende Thread alle Monitore. Dagegen wird der Monitor von `object` freigegeben, wenn Sie `object.wait(time)` ausführen. Falls Sie also `Thread.sleep(time)` in einem synchronisierten Block aufrufen, kann kein anderer Thread auf einen Block zugreifen, der über dasselbe Objekt synchronisiert ist. Dies gilt für die gesamte Schlafzeit. Rufen Sie dagegen `object.wait(time)` auf, so können andere Threads durchaus Code-Blöcke ausführen, die mit `object` synchronisiert sind.

```
// Andere Threads können während der Sekunde
// mit this synchronisierte Code-Blöcke ausführen.
public synchronized void waitASecond()
      throws InterruptedException {
   // Warte eine Sekunde.
   wait(1000);
}

// Niemand kann während der Sekunde mit this synchronisierte
// Code-Blöcke ausführen.
public synchronized void sleepASecond()
      throws InterruptedException {
   // Schlafe eine Sekunde.
   Thread.sleep(1000);
}
```

Es gilt:

Benutzen Sie in synchronisierten Code-Blöcken niemals `Thread.sleep()`, es sei denn, Sie wollen wirklich alle mit denselben Objekten synchronisierten Blöcke für die Dauer des Schlafes sperren.

9.2.5 Prioritäten setzen und Vorrang lassen

Die Methode `setPriority()` der Klasse `Thread` verspricht, dass Sie mit ihr die Priorität eines Threads beeinflussen können. Verlassen Sie sich nicht darauf. Das Setzen der Priorität ist eine der sehr schlecht portierbaren Fähigkeiten von Threads. Es gibt Plattformen, auf denen die Priorität eines Threads massive Auswirkungen hat, und es gibt Plattformen, auf denen die Priorität überhaupt nicht beachtet wird.

Gleiches gilt für `Thread.yield()`. Der Effekt von `yield()` ist von Plattform zu Plattform verschieden. Auch hier kann der Aufruf keinerlei, positive oder negative Auswirkungen haben. Es gilt:

`Thread.yield()` und `thread.setPriority()` sind nicht portabel. Zum Optimieren von Programmen sind sie daher nur eingeschränkt geeignet.

9.3 Skalieren mit Threads

Wenn Sie Ihre Hardware auf ein Multiprozessor-System hochrüsten, können Sie davon nur profitieren, wenn Sie die Rechenlast auf mehrere Threads verteilen. Benutzen Sie nur einen Thread oder eine so genannten Green-Thread-Implementierung der Java VM (*Kapitel 3.4 Die richtige VM auswählen*), wird maximal ein Prozessor ausgelastet. Falls Ihr Kunde sich einen teuren 16-Prozessor-Rechner angeschafft hat, wird er sich nicht gerade freuen, wenn Sie Software liefern, die nur ein Sechzehntel der potenziellen Rechenleistung nutzt.

Kandidaten für Multithreading sind Programme wie beispielsweise HTTP- oder FTP-Server, die viele parallele Verbindungen zu Klienten unterhalten. Auch auf Algorithmus-Ebene kann sich Parallelisierung lohnen. Die Crux an der Sache ist, dass parallele Algorithmen auf Einprozessormaschinen häufig langsamer sind als ihre sequenziellen Gegenstücke. Von daher macht es keinen Sinn, Algorithmen prinzipiell für mehrere Prozessoren auszulegen. Seit JDK 1.4.0 gibt es jedoch eine Methode, mit der Sie zur Laufzeit herausfinden können, über wie viele Prozessoren die Ausführungsumgebung verfügt. Die Methode heißt `Runtime.getRuntime().availableProcessors()`. Wir wollen an einem einfachen Beispiel ausprobieren, wie sich das Multiplizieren von Matrizen auf Mehrprozessormaschinen beschleunigen lässt.

Per Definition ist das Produkt zweier Matrizen A_{ik} und B_{ik} eine Matrix C_{ik}, deren Elemente c_{ik} die Skalarprodukte des *i*-ten Zeilenvektors von A und des *k*-ten Spaltenvektors von *B* sind.

$$\begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} c_{11} & c_{12} \\ c_{21} & c_{22} \\ c_{31} & c_{32} \end{pmatrix}$$

$$c_{ik} = \begin{pmatrix} a_{i1} & a_{i2} \end{pmatrix} \cdot \begin{pmatrix} b_{1k} \\ b_{2k} \end{pmatrix} = a_{i1}b_{1k} + a_{i2}b_{2k}$$

$$\Rightarrow \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \\ a_{31} & a_{32} \end{pmatrix} \cdot \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \end{pmatrix} = \begin{pmatrix} a_{11}b_{11} + a_{12}b_{21} & a_{11}b_{12} + a_{12}b_{22} \\ a_{21}b_{11} + a_{22}b_{21} & a_{21}b_{12} + a_{22}b_{22} \\ a_{31}b_{11} + a_{32}b_{21} & a_{31}b_{12} + a_{32}b_{22} \end{pmatrix}$$

Daraus ergibt sich folgender trivialer Algorithmus:

```
public static int[][] conventionalMultiply(int[][] a, int[][] b) {
   int[][] c = new int[a.length][b.length];
   for (int i = 0; i < a.length; i++) {
      // temporäre Variablen, um teure Array-Zugriffe zu sparen
      int[] cTemp = c[i];
      for (int k = 0; k < b.length; k++) {
         int temp = 0;
         int[] aTemp = a[i];
         for (int n = 0; n < b.length; n++) {
            temp += aTemp[n] * b[n][k];
         }
         cTemp[k] = temp; // c[i][k] = temp
      }
   }
   return c;
}
```

Die temporären Variablen dienen dazu, teure Array-Zugriffe zu sparen (*Kapitel 6.2.2 Teure Array-Zugriffe*). Da die Berechnung eines c_{ik} von allen anderen c_{ik} unabhängig ist, lässt sich dieser Algorithmus leicht parallelisieren, indem wir zum Berechnen von c_{ik} für verschiedene Wertebereiche von *i* jeweils einen eigenen Thread starten.

```
public static int[][] multithreadedMultiply(int[][] a, int[][] b,
      int threads) {
   // Falls nur ein Thread gestartet werden soll, benutzen wir
   // den aktuellen Thread und konventionelles Multiplizieren
   if (threads <= 1) {
      return conventionalMultiply(a, b);
   }
   int[][] c = new int[a.length][b.length];
   try {
      Thread[] t = new Thread[threads];
      int length = (int) Math.ceil(a.length / (double) threads);
      // Starten der Threads mit jeweils einem Teilproblem
      for (int i = 0; i < threads; i++) {
         t[i] = new Thread(new MultiprocessorMultiplier(a, b, c,
            length * i, length));
         t[i].start();
      }
      // Warten, bis alle Threads fertig sind
      for (int i = 0; i < threads; i++) {
         t[i].join();
      }
   } catch (InterruptedException ie) {
      // dürfte nicht passieren
      ie.printStackTrace();
   }
   return c;
```

```
}

private static class MultiprocessorMultiplier implements Runnable {
   private int[][] a;
   private int[][] b;
   private int[][] c;
   private int startRow;
   private int length;

   public MultiprocessorMultiplier(int[][] a, int[][] b, int[][] c,
         int startRow, int length) {
      this.a = a;
      this.b = b;
      this.c = c;
      this.startRow = startRow;
      this.length = length;
   }

   // Genau wie conventionalMultiply - nur die Grenzen der
   // i-Schleife (Zeile) sind anders.
   public void run() {
      for (int i = startRow, len = length + startRow; i < len
            && i < a.length; i++) {
         int[] cTemp = c[i];
         for (int k = 0; k < b.length; k++) {
            int temp = 0;
            int[] aTemp = a[i];
            for (int n = 0; n < b.length; n++) {
               temp += aTemp[n] * b[n][k];
            }
            cTemp[k] = temp;
         }
      }
   }
}
```

Das Ergebnis unserer Bemühungen zeigt Abbildung 9.2. Auf einer Zweiprozessormaschine (450 Mhz Intel Xeon) lässt sich die Ausführungszeit mit 32 Threads auf 55 Prozent der Ausführungszeit mit einem Thread senken.

Die optimalen 50 Prozent werden unter anderem deshalb nicht erreicht, weil Threads natürlich auch Verwaltungsaufwand mit sich bringen. Wie wir gesehen haben, ist insbesondere das Starten eines Threads keine billige Angelegenheit. Dies, sowie die Berechnung, welches Teilproblem von welchem Thread gelöst werden soll, muss sequenziell erfolgen. Genau dieser Aufwand ist auch der begrenzende Faktor für die Parallelisierung jedes Algorithmus. Gemäß Gene Amdahls Gesetz von 1967 sind es die sequenziellen Anteile eines parallelen Algorithmus, der die Laufzeitreduzierung durch mehr Prozessoren nach oben hin beschränkt.

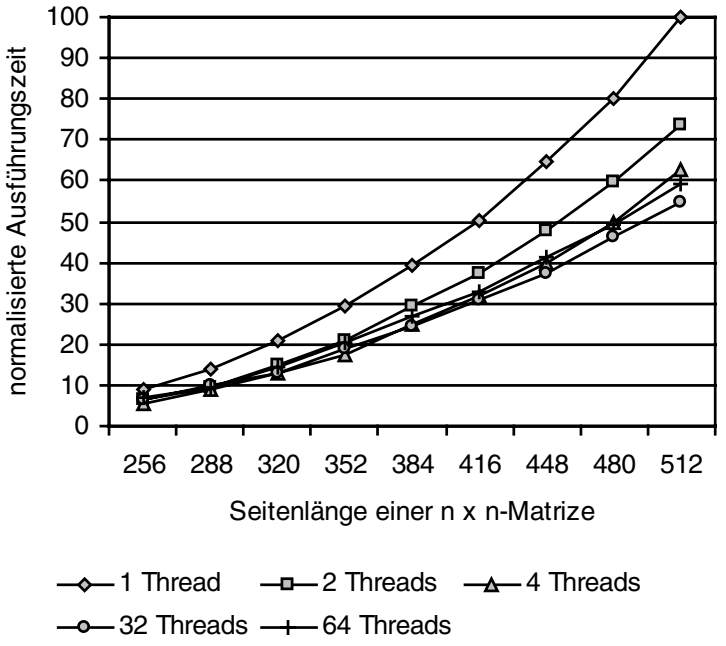

Abbildung 9.2: Ausführungszeit einer Matrizen-Multiplikation auf einer Zweiprozessormaschine in Abhängigkeit von der Anzahl verwendeter Threads und der Größe der Matrize

Bemerkenswert ist auch, dass die Laufzeit in unserem Experiment nicht bei zwei Threads, sondern bei 32 Threads optimal ist und bei weiter steigender Threadzahl wieder ansteigt. Es kann also durchaus Sinn machen, mehr Threads zu starten, als Prozessoren vorhanden sind – jedoch sollten es nicht zu viele sein. Letztlich hängt der Faktor aber von VM, Betriebssystem und Hardware ab.

Für unsere Testplattform wäre also folgender Code sinnvoll:

```
public static int[][] multiply(int[][] a, int[][] b) {
   int availableProcessors
      = Runtime.getRuntime().availableProcessors();
   if (availableProcessors == 1) {
      return conventionalMultiply(a, b);
   }
   return multithreadedMultiply(a, b, availableProcessors*16);
}
```

Dies ist jedoch nicht allgemeingültig. Häufig führt ein Faktor von 4 oder 8 zu vergleichbaren Laufzeiten.

9.4 Threads in Benutzeroberflächen

Gerade in Benutzeroberflächen spielt die wahrgenommene Performance eine große Rolle. In den folgenden beiden Abschnitten wollen wir uns damit beschäftigen, wie man Threads sinnvoll zusammen mit dem AWT (Abstract Window Toolkit) und Swing einsetzt.

9.4.1 Lebendige AWT-Oberflächen

Als Beispiel wollen wir ein kleines Programm anschauen, das den Benutzer eine URL eingeben lässt, die entsprechende Datei lädt und in einer `TextArea` anzeigt.

Der Code für die Oberfläche unserer Applikation sieht folgendermaßen aus:

```
package com.tagtraum.perf.threads;

import java.awt.*;
import java.awt.event.*;
import java.io.*;
import java.net.*;

public class URLLoaderDemo extends Frame {

   private TextField urlField;
   private TextArea textArea;
   private Label statusBar;

   public static void main(String[] args) {
      new URLLoaderDemo();
   }

   public URLLoaderDemo() {
      super("URLLoaderDemo");
      Panel panel = new Panel();
      Button loadButton = new Button("Load");
      urlField = new TextField("Enter URL here", 30);
      textArea = new TextArea(20, 65);
      statusBar = new Label();
      URLLoader urlLoader = new URLLoader();
      loadButton.addActionListener(urlLoader);
      urlField.addActionListener(urlLoader);
      panel.add(urlField, BorderLayout.NORTH);
      panel.add(loadButton, BorderLayout.NORTH);
      add(panel, BorderLayout.NORTH);
      add(textArea, BorderLayout.CENTER);
      add(statusBar, BorderLayout.SOUTH);
      setSize(300, 300);
      pack();
```

```
        setVisible(true);
    }
    ...
}
```

Was jetzt noch fehlt, ist die Klasse `URLLoader`, die als `ActionListener` für den LOAD-Button und das Textfeld `urlField` dient.

Zunächst einmal die Version ohne Threads:

```
// Innere Klasse von URLLoaderDemo
// Version ohne Threads - Gehen Sie so nicht vor!
private class URLLoader implements ActionListener {

   public void actionPerformed(ActionEvent e) {
      textArea.setText("");
      statusBar.setText("Loading " + urlField.getText());
      Reader in = null;
      try {
         in = new InputStreamReader(
           new URL(urlField.getText()).openStream()
         );
         char[] cbuf = new char[1024 * 8];
         for (int count = 0; (count = in.read(cbuf)) != -1;) {
           textArea.append(new String(cbuf, 0, count));
           repaint();
         }
         statusBar.setText("Done.");
      } catch (IOException ioe) {
         statusBar.setText(ioe.toString());
      } finally {
         if (in != null)
           try { in.close(); } catch (IOException ioe) {}
      }
      repaint();
   }
}
```

Ein kleiner Test zeigt, dass die Applikation ganz ordentlich arbeitet. Sie hat jedoch ein paar Nachteile:

- Während des Ladens reagiert die Applikation nicht auf Benutzereingaben.
- Während des Ladens lässt sich die Größe des Fensters nicht verändern.
- Der Benutzer wird nicht über den Fortschritt des Ladens informiert. Er kann nicht feststellen, wie lange es noch dauert.

Während der letzte Punkt leicht auch ohne Threads zu beheben ist, erfordern die beiden ersten Punkte Threads. Dies ist deshalb der Fall, weil das Laden in der Methode

`actionPerformed()` geschieht. `actionPerformed()` wird nämlich vom GUI-Thread ausgeführt. Und wenn der GUI-Thread mit dem Laden einer Datei beschäftigt ist, kann er sich logischerweise nicht den Benutzereingaben widmen.

Hier die zweite Version von `URLLoader`, diesmal mit Threadunterstützung:

```
// Innere Klasse von URLLoaderDemo
// Version mit Threads
private class URLLoader implements ActionListener, Runnable {
   private Thread t;
   private URL url;

   public void actionPerformed(ActionEvent e) {
      try {
         // Stoppe Thread, falls einer läuft.
         cancel();
         url = new URL(urlField.getText());
         // Starte neuen Thread
         t = new Thread(this);
         t.start();
      } catch (MalformedURLException mfue) {
         statusBar.setText(mfue.toString());
      }
   }

   // Stoppt evtl. schon vorhandenen Thread.
   public void cancel() {
      if (t != null && t.isAlive()) {
         try {
            t.interrupt();
            t.join();
            t = null;
         } catch (InterruptedException ie) {
            statusBar.setText(ie.toString());
         }
      }
   }

   public void run() {
      // Löscht die TextArea
      textArea.setText("");
      statusBar.setText("Loading " + url);
      repaint();
      Reader in = null;
      try {
         URLConnection connection = url.openConnection();
         in = new InputStreamReader(connection.getInputStream());
         // Wir merken uns die Gesamtlänge.
         int contentLength = connection.getContentLength();
         char[] cbuf = new char[1024 * 8];
         // Lese so lange, wie Daten da sind und der Thread nicht
```

```
            // unterbrochen wurde.
            for (int count = 0, sum = 0;
                  (count = in.read(cbuf)) != -1
                  && !t.isInterrupted();) {
               textArea.append(new String(cbuf, 0, count));
               // sum beinhaltet die Anzahl der bereits
               // gelesenen Bytes.
               sum += count;
               // Zeige an, wie weit wir sind.
               showProgress(contentLength, sum);
               repaint();
            }
            if (t.isInterrupted())
               statusBar.setText("Interrupted.");
            else
               statusBar.setText("Done.");
         } catch (IOException ioe) {
            statusBar.setText(ioe.toString());
         } finally {
            if (in != null)
               try { in.close(); } catch (IOException ioe) {}
         }
         repaint();
      }

      // Zeigt an, wie viel wir schon geladen haben.
      private void showProgress(int contentLength, int sum) {
         if (contentLength != -1) {
            statusBar.setText((sum / 1024) + "K/"
               + (contentLength / 1024) + "K - "
               + (sum * 100 / (contentLength)) + "%");
         } else {
            // Wenn contentLength -1 ist, wissen wir nicht, wie viel
            // wir insgesamt laden müssen. Wir zeigen aber zumindest
            // an, wie viel wir schon geladen haben.
            statusBar.setText((sum / 1024) + "K");
         }
      }
   }
```

Auch, wenn das Laden kein bisschen schneller erfolgt, so kann der Benutzer nach Belieben mit dem Fenster rumspielen und er wird über den Fortschritt des Ladens mit einer Prozentzahl und absoluten Werten informiert. Da beim erneuten Drücken des LOAD-Buttons ein bereits laufender Thread mit der `cancel()`-Methode gestoppt wird, kann der Benutzer sogar während des Ladens einfach eine andere URL eingeben und LOAD drücken. Der aktuelle Ladevorgang wird dann unterbrochen und ein neuer begonnen. In der ersten Version musste der Nutzer warten, bis die Datei vollständig geladen war.

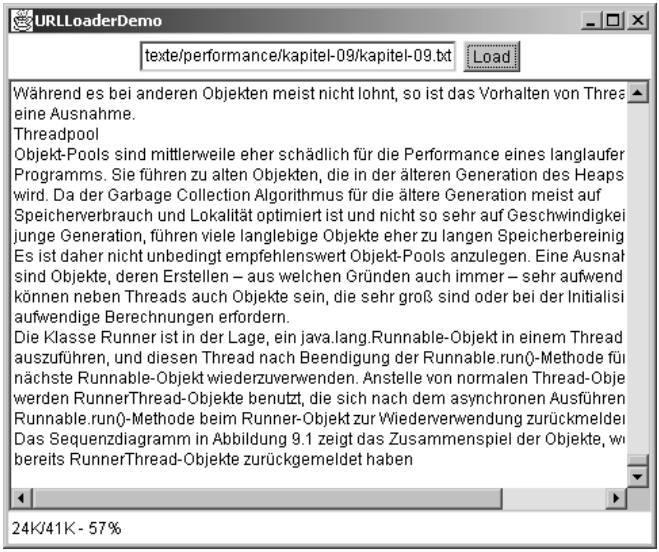

Abbildung 9.3: Screenshot der zweiten Version mit Fortschrittsanzeige

Da wir nun über eine `cancel()`-Methode verfügen, können wir dem Benutzer natürlich auch einen entsprechenden Knopf zur Verfügung stellen.

Insgesamt ist das Arbeiten mit der zweiten Version wesentlich befriedigender. Der Benutzer ist jederzeit informiert und hat die Kontrolle über die Applikation – und nicht umgekehrt wie in der ersten Version.[2]

Daraus folgt:

Aufwändige Operationen sollten, wenn es geht, in einem separaten Thread ausgeführt werden, um die Benutzeroberfläche nicht zu blockieren.

Und:

Wenn irgendwie möglich, informieren Sie den Benutzer immer über den Stand der von ihm angestoßenen Operationen.

9.4.2 Threads in Swing

Während AWT threadsicher ist, gilt dies nicht für Swing. So gut wie alle Aktionen, die die Benutzeroberfläche manipulieren, *müssen* im GUI-Thread ablaufen. Es gibt nur wenige Methoden innerhalb Swings, die trotzdem threadsicher sind. Sie sind als Ausnahmen entsprechend in der Dokumentation gekennzeichnet.

2 In Sun JDK 1.4.0 für Windows tauchen am Ende des Textes zufällige Textfetzen auf. Dies ist ein Fehler im JDK.

Wenn Sie das GUI trotzdem von außerhalb des GUI-Threads manipulieren wollen, müssen Sie sich einer der beiden Methoden `javax.swing.SwingUtilities.invokeLater()` oder `invokeAndWait()` bedienen.

`invokeLater()` führt die `run()`-Methode eines `java.lang.Runnable`-Objektes *asynchron* zum aufrufenden Thread auf. Das heißt, die Methode kehrt unmittelbar zurück und die `run()`-Methode wird später vom GUI-Thread ausgeführt, nachdem alle Ereignisse in der AWT-Ereignis-Warteschlange verarbeitet sind.

`invokeAndWait()` hingegen führt die `run()`-Methode eines `Runnable`-Objektes *synchron* zum aufrufenden Thread auf. Dies bedeutet, dass der aufrufende Thread warten muss, bis alle Ereignisse in der AWT-Ereignis-Queue verarbeitet sind und anschließend die `run()`-Methode des `Runnable`-Objektes vom GUI-Thread ausgeführt wurde. Der Vorteil gegenüber `invokeLater()` liegt darin, dass evtl. ausgelöste Ausnahmen in einer `InvocationTargetException` gekapselt und an den Aufrufer weitergeleitet werden. Somit ist eine robustere Ausnahmebehandlung möglich.

Es empfiehlt sich, für beide Methoden anonyme, innere Klassen zu verwenden. Beispiel:

```
SwingUtilities.invokeLater(new Runnable() {
   public void run() {
      // erledige, was auch immer erledigt werden muss.
   }
});
```

10 Effiziente Ein- und Ausgabe

Für eine effiziente Ein-/Ausgabe ist es wesentlich, zu verstehen, wie die einzelnen Klassen der Pakete `java.io` und seit JDK 1.4 auch `java.nio` funktionieren. Wir wollen zunächst an einem einfachen Beispiel illustrieren, wofür einige Klassen gut sind und wofür nicht.

10.1 Fallstudie Dateikopieren

Man könnte annehmen, dass so eine einfache Operation wie das Kopieren einer Datei schon von irgendeiner Klasse der Java-Klassenbibliothek implementiert wird. Dies ist jedoch nicht so. Daher müssen wir dies selbst erledigen.

Die erste Version unseres Dateikopierers sieht folgendermaßen aus:

```
package com.tagtraum.perf.io;

import java.io.*;

// Naiver Dateikopierer! Nicht nachmachen!
public class FileCopyDemo1 {
   public static void main(String[] args) throws IOException {
      Reader reader = new FileReader(args[0]);
      Writer writer = new FileWriter(args[1]);
      int c;
      while ((c = reader.read()) != -1) {
         writer.write(c);
      }
      reader.close();
      writer.close();
   }
}
```

Quell- und Zieldatei müssen als Argumente angegeben werden. Anschließend werden ein `FileReader` und ein `FileWriter` zum Lesen bzw. Schreiben eines Zeichens benutzt. So wird die gesamte Datei kopiert. Mit anderen Worten: Für jedes Zeichen wird einmal von der Festplatte gelesen und einmal auf sie geschrieben. Sie können sich ausmalen, welche katastrophalen Folgen dieses Vorgehen bei einer mehrere Mbyte großen Datei hat – die Tastatur würde Staub fangen, bevor die Kopie erstellt ist.

Die Lösung ist puffern. Statt des einfachen `FileReaders` bzw. `-Writers` benutzen wir also einen `BufferedReader` bzw. einen `BufferedWriter`. Diese beiden Klassen sorgen dafür, dass bei `read()`- bzw. `write()`-Aufrufen zunächst ein Pufferspeicher manipuliert wird und erst wenn dieser voll bzw. leer ist die Festplatte in Aktion tritt.

```
package com.tagtraum.perf.io;

import java.io.*;

// Leicht verbessert, aber nicht zu empfehlen!
public class FileCopyDemo2 {
   public static void main(String[] args) throws IOException {
      Reader reader = new BufferedReader(new FileReader(args[0]));
      Writer writer = new BufferedWriter(new FileWriter(args[1]));
      int c;
      while ((c = reader.read()) != -1) {
         writer.write(c);
      }
      reader.close();
      writer.close();
   }
}
```

Ein kleiner Schönheitsfehler hieran ist, dass wir für eine Datei mit einer Million Zeichen jeweils eine Million Mal `read()` und eine Million Mal `write()` aufrufen. Obwohl nicht jedes Mal auf die Festplatte durchgegriffen wird, ist dies nicht gerade effizient. Besser ist es, jeweils einen `char`-Array zu lesen und dann zu schreiben.

```
package com.tagtraum.perf.io;

import java.io.*;

// Wieder leicht verbessert, aber immer noch nicht
// zu empfehlen!
public class FileCopyDemo3 {
   public static void main(String[] args) throws IOException {
      Reader reader = new BufferedReader(new FileReader(args[0]));
      Writer writer = new BufferedWriter(new FileWriter(args[1]));
      int length;
      char[] cbuf = new char[1024 * 8];
      while ((length = reader.read(cbuf)) != -1) {
         writer.write(cbuf, 0, length);
      }
      reader.close();
      writer.close();
   }
}
```

Im Grunde übernehmen wir somit das Puffern selbst. `BufferedReader` und `BufferedWriter` – die, nebenbei bemerkt, auch noch voll synchronisiert sind – werden also überflüssig. Also weg damit!

```
package com.tagtraum.perf.io;

import java.io.*;

// Schon nicht schlecht. Aber etwas ist hier noch falsch!
public class FileCopyDemo4 {
   public static void main(String[] args) throws IOException {
      Reader reader = new FileReader(args[0]);
      Writer writer = new FileWriter(args[1]);
      int length;
      char[] cbuf = new char[1024 * 8];
      while ((length = reader.read(cbuf)) != -1) {
         writer.write(cbuf, 0, length);
      }
      reader.close();
      writer.close();
   }
}
```

Natürlich können Sie die Größe des `char`-Arrays variieren – je nachdem wie viel Speicher Sie für das Kopieren belegen wollen.

Nun sind `Reader` und `Writer` Klassen, die speziell für *Zeichen* ausgelegt sind. Und zum Kopieren von Zeichen ist unsere letzte Version schon ziemlich gut. Im Grunde wollen wir aber gar keine Zeichen kopieren, sondern *Bytes*. Da wir jedoch `Reader` und `Writer` benutzt haben, haben wir in allen bisherigen Versionen unseres Dateikopierers beim Lesen Bytes in Zeichen umgewandelt und beim Schreiben Zeichen in Bytes.

Offensichtlich ist es günstiger, auf dieses Hin- und Herwandeln zu verzichten. Statt `Reader` und `Writer` sollten wir also `InputStream` und `OutputStream` verwenden.

```
package com.tagtraum.perf.io;

import java.io.*;

// Performanter Dateikopierer
public class FileCopyDemo5 {
   public static void main(String[] args) throws IOException {
      InputStream in = new FileInputStream(args[0]);
      OutputStream out = new FileOutputStream(args[1]);
      int length;
      byte[] buf = new byte[1024 * 8];
      while ((length = in.read(buf)) != -1) {
         out.write(buf, 0, length);
      }
```

```
        in.close();
        out.close();
    }
}
```

Seit JDK 1.4 gibt es noch eine weitere, potenziell effizientere Möglichkeit eine Datei zu kopieren. Mit jedem Dateistrom ist ein `java.nio.channels.FileChannel` assoziiert. Dieser verfügt über die Methoden `transferTo()` und `transferFrom()`, die dazu optimiert sind, Daten zu oder von einer Datei zu übertragen. Dabei, so die Dokumentation, werden möglicherweise sehr effiziente Betriebssystemroutinen benutzt.

```
package com.tagtraum.perf.io;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.IOException;
import java.nio.channels.FileChannel;

// Dateikopierer für JDK >= 1.4.0
public class FileCopyDemo6 {
   public static void main(String[] args) throws IOException {
      FileInputStream in = new FileInputStream(args[0]);
      FileOutputStream out = new FileOutputStream(args[1]);
      FileChannel inChannel = in.getChannel();
      FileChannel outChannel = out.getChannel();
      long position = 0;
      long transferred;
      long remaining = inChannel.size();
      while (remaining > 0) {
         transferred = inChannel.transferTo(position, remaining,
            outChannel);
         position += transferred;
         remaining -= transferred;
      }
      in.close();
      out.close();
   }
}
```

Ein kleiner Test unter Windows 2000 mit Sun JDK 1.4.0 Client offenbart jedoch, dass in dieser Umgebung die `transferTo()`/`transferFrom()`-Methoden eher langsamer sind als die konventionelle Methode mit einem einfachen `byte`-Array. Dies wird sich jedoch vermutlich in späteren Versionen ändern.

Wir wollen noch einmal die wichtigsten Punkte dieser kleinen Fallstudie zusammenfassen:

- Wenn möglich, puffern Sie alle Ein- und Ausgaben.
- Eigene Puffer sind meist schneller als `BufferedReader`, `BufferedWriter`, `BufferedInputStream` und `BufferedOuputStream`, da sie nicht synchronisiert werden müssen.

- Benutzen Sie `Reader` und `Writer` ausschließlich, wenn Sie tatsächlich an Zeichen und nicht Bytes interessiert sind.
- Das neue Ein-/Ausgabe-API `java.nio` kann Operationen beschleunigen, muss aber nicht.

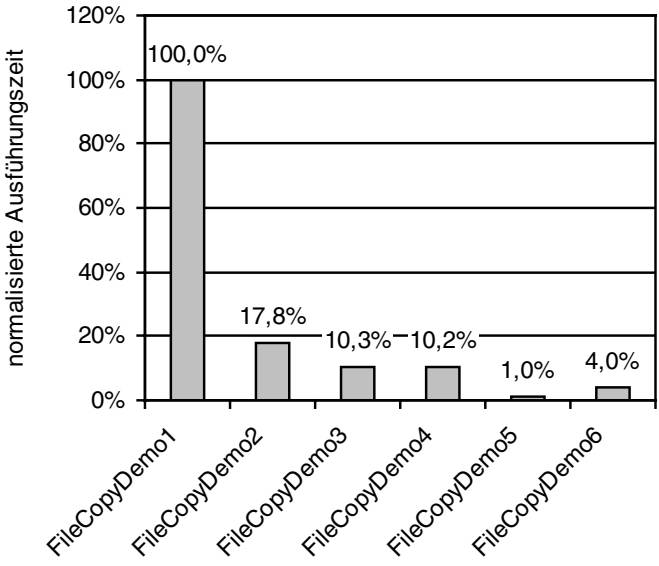

Abbildung 10.1: Kopieren einer 633 Kbyte-Datei unter Windows 2000 mit Sun JDK 1.4.0 Client auf verschiedene Weisen

10.2 Texte ausgeben

Wie oben bereits festgestellt, ist der korrekte Weg einen Text auszugeben, einen `Writer` zu verwenden. Wir wollen verschiedene `Writer` testen. Wir beginnen mit dem `PrintWriter`.

Unser Testprogramm schreibt die übergebenen Strings in eine Datei. Nach jedem String soll zudem ein Zeilenseparator ausgegeben werden. Mit dem `PrintWriter` lässt sich diese Aufgabe sehr leicht bewältigen. Natürlich puffern wir die Ausgabe, da wir ja wissen, welche fatalen Auswirkungen es hat, nicht zu puffern.

```
package com.tagtraum.perf.io;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;
```

```
// Nicht nachahmen!
public class CharWriterDemo1 {
   public static void main(String[] args) throws IOException {
      PrintWriter out = new PrintWriter(
         new BufferedWriter(new FileWriter("CharWriterDemo.tmp")),
         true
      );
      for (int i = 0; i < args.length; i++) {
         out.println(args[i]);
      }
      out.close();
   }
}
```

In ein paar Testläufen stellen wir fest, dass unser Programm nicht gerade sehr schnell ist (Sie finden die Ergebnisse am Ende dieses Abschnittes in Tabelle 10.1). Dies liegt daran, dass wir im Konstruktor des `PrintWriters` als zweiten Parameter ein `true` übergeben. Dies bedeutet, dass nach jedem `println()`-Aufruf die Methode `flush()` aufgerufen wird (Autoflush) und somit der Inhalt des Pufferspeichers in den darunter liegenden `Writer` geschrieben wird. In unserem Fall ist diese der `FileWriter`, der wiederum direkt auf die Festplatte schreibt. Kein Wunder also, dass das Programm ein wenig langsam ist. Der Puffer ist gänzlich nutzlos.

Übrigens erliegen Sie demselben Mechanismus, wenn Sie etwas in die Standard- oder Fehlerausgabe schreiben. Daher ist es für die Performance eines Programms sehr schädlich, wenn Sie beispielsweise in die Standardausgabe Protokollinformationen schreiben. Auf der anderen Seite wissen Sie nicht genau, wo das Programm steckt, wenn Sie die Standardausgabe puffern. Der Mittelweg ist, die Standardausgabe zu puffern, die Fehlerausgabe jedoch nicht. Wirklich wichtige Dinge erfahren Sie so sofort.

Versuchen wir es anders. Diesmal ohne Autoflush und auch ohne Puffer.

```
package com.tagtraum.perf.io;

import java.io.FileWriter;
import java.io.PrintWriter;
import java.io.IOException;

// Langsam, da ungepuffert
public class CharWriterDemo2 {
   public static void main(String[] args) throws IOException {
      PrintWriter out = new PrintWriter(
         new FileWriter("CharWriterDemo.tmp"));
      for (int i = 0; i < args.length; i++) {
         out.println(args[i]);
      }
      out.close();
   }
}
```

Das Ergebnis ist schon wesentlich besser, jedoch immer noch nicht zufrieden stellend. `PrintWriter` muss anscheinend doch gepuffert werden.

```
package com.tagtraum.perf.io;

import java.io.FileWriter;
import java.io.BufferedWriter;
import java.io.IOException;

public class CharWriterDemo3 {
   public static long main(String[] args) throws IOException {
      BufferedWriter out = new BufferedWriter(
         new FileWriter("CharWriterDemo.tmp"));
      for (int i = 0; i < args.length; i++) {
         out.write(args[i]);
         out.newLine();
      }
      out.close();
   }
}
```

Die gepufferte Version ist in der Tat etwas schneller. Wie viel, hängt von der Länge der ausgegebenen Zeilen ab.

Es ist nun tatsächlich recht schwierig, die letzte Version noch zu beschleunigen. Daher greifen wir zu einem Trick. Wir nehmen einfach mal an, dass alle Zeichen unseres Strings sich mit einem Byte darstellen lassen. Davon ausgehend schreiben wir unseren eigenen `Writer`, einen maskierten `Writer`, der nur jene Bits durchlässt, die zuvor in einer Bitmaske gesetzt wurden (Listing 10.1). Wenn wir das gesamte niedrige Byte eines Zeichens durchlassen, entspricht dies ISO 8859-1, dem in Westeuropa üblichen Zeichensatz. Unser Vorgehen ist dabei um einiges einfacher, als es normalerweise intern durch einen entsprechenden `OutputStreamWriter` geschehen würde.

```
package com.tagtraum.perf.io;

import java.io.CharConversionException;
import java.io.IOException;
import java.io.OutputStream;
import java.io.Writer;
import java.util.ResourceBundle;

public class MaskedStreamWriter extends Writer {

   // Vordefinierte Masken für ASCII und ISO 8859-1 (Latin 1)
   public static final int ASCII_MASK = 0x7f;
   public static final int ISO_8859_1_MASK = 0xff;

   private OutputStream out;
   private byte[] buf;
```

```
private int mask;
private static ResourceBundle localStrings
   = ResourceBundle.getBundle(
   "com.tagtraum.perf.io.localStrings");

public MaskedStreamWriter(OutputStream out, int bufsize,
      int mask) {
   this.out = out;
   buf = new byte[bufsize];
   // Masken, die mehr als die acht niedrigen Bits durchlassen,
   // werden nicht unterstützt.
   if (mask > 0xff)
      throw new IllegalArgumentException(
         localStrings.getString("illegal_mask"));
   this.mask = mask;
}

public void write(int c) throws IOException {
   // Prüft zunächst, ob das zu schreibende Zeichen durch
   // die Maske passt.
   if ((c & ~mask) != 0)
      throw new CharConversionException(
         localStrings.getString("unallowed_char") + " " + c);
   out.write(c);
}

public void write(char[] cbuf, int offset, int length)
      throws IOException {
   // Schreibt alle Zeichen zunächst in einen byte-Array
   int l = Math.min(buf.length, length);
   while (l > 0) {
      int end = offset + l;
      for (int i = offset; i < end; i++) {
         char c = cbuf[i];
         // Prüft, ob das zu schreibende Zeichen durch
         // die Maske passt.
         if ((c & ~mask) != 0)
            throw new CharConversionException(
               localStrings.getString("unallowed_char")
               + " " + c);
         else
            buf[i - offset] = (byte) c;
      }
      out.write(buf, 0, l);
      offset += l;
      length -= l;
      l = Math.min(buf.length, length);
   }
}
```

```
    public void write(String str, int off, int len)
          throws IOException {
       if (len < 0)
          throw new IndexOutOfBoundsException();
       char cbuf[] = new char[len];
       str.getChars(off, off + len, cbuf, 0);
       write(cbuf, 0, len);
    }

    public void close() throws IOException {
       out.close();
    }

    public void flush() throws IOException {
       out.flush();
    }

}
```

Listing 10.1: Maskierter Stream-Writer

Diesen `MaskedStreamWriter` bauen wir in unser Testprogramm ein und lassen den Test laufen. Und siehe da! Die Ausgabegeschwindigkeit verdoppelt sich (Tabelle 10.1) gegenüber der letzten Version.

```
package com.tagtraum.perf.io;

import java.io.BufferedWriter;
import java.io.FileOutputStream;
import java.io.IOException;

public class CharWriterDemo4 {

   public static void main(String[] args) throws IOException {
      long start = System.currentTimeMillis();
      BufferedWriter out = new BufferedWriter(
         new MaskedStreamWriter(
         new FileOutputStream("CharWriterDemo.tmp"),
         MaskedStreamWriter.ISO_8859_1_MASK));
      for (int i = 0; i < args.length; i++) {
         out.write(args[i]);
         out.newLine();
      }
      out.close();
   }
}
```

Java VM	CharWriter Demo1	CharWriter Demo2	CharWriter Demo3	CharWriter Demo4
Sun JDK 1.3.1 Client	100%	46%	40%	13%
Sun JDK 1.3.1 Server	79%	30%	24%	13%
Sun JDK 1.4.0 Client	90%	33%	25%	14%
Sun JDK 1.4.0 Server	84%	25%	19%	12%
IBM JDK 1.3.0	80%	32%	24%	12%

Tabelle 10.1: Normalisierte Ausführungszeiten der einzelnen CharWriterDemos

Aus unseren Experimenten folgt:

- Wenn möglich, sollten `PrintWriter` und `PrintStream` nicht im Autoflush-Modus benutzt werden.
- Puffern nutzt nur etwas, wenn `flush()` selten aufgerufen wird.
- Die Umwandlung von `char` in `byte` ist sehr teuer und kann gegebenenfalls schneller durch einen eigenen `Writer` erledigt werden.

10.3 Texte einlesen

Im Grunde funktioniert das Einlesen eines Textes genauso wie das Schreiben. Nur müssen diesmal Bytes in Zeichen umgewandelt werden und nicht umgekehrt. Unser Beispielprogramm soll eine Datei zeilenweise einlesen. Natürlich puffern wir das Einlesen entsprechend. Und glücklicherweise verfügt `BufferedReader` sogar über eine `readLine()`-Methode.

```
package com.tagtraum.perf.io;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class CharReaderDemo1 {
   public static void main(String[] args) throws IOException {
      long start = System.currentTimeMillis();
      BufferedReader in = new BufferedReader(
         new FileReader("CharWriterDemo.tmp"));
      String line;
      while ((line = in.readLine()) != null) {
      }
      in.close();
   }
}
```

Das Ergebnis (Tabelle 10.2) ist nicht schlecht und viel besser ist es wohl auch nicht möglich. Jedoch können wir den gleichen Kniff anwenden wie beim Schreiben – nämlich das höhere Byte ignorieren. Und das sogar ganz ohne Aufwand, da `DataInputStream` über eine entsprechende Methode verfügt.

```
package com.tagtraum.perf.io;

import java.io.BufferedInputStream;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.IOException;

public class CharReaderDemo2 {
   public static void main(String[] args) throws IOException {
      long start = System.currentTimeMillis();
      DataInputStream in = new DataInputStream(
         new BufferedInputStream(
         new FileInputStream("CharWriterDemo.tmp")));
      String line;
      while ((line = in.readLine()) != null) {
      }
      in.close();
   }
}
```

Die `readLine()`-Methode ist als veraltet (deprecated) markiert, gerade weil sie nur für wenige Zeichensätze geeignet ist. Für diese wenige Zeichensätze (ASCII, ISO 8859-1) ist sie jedoch mit Sun Client-VMs ein wenig schneller als ihr Gegenstück aus dem `BufferedReader`. In anderen VMs ist sie jedoch langsamer.

Java VM	CharReaderDemo1	CharReaderDemo2
Sun JDK 1.3.1 Client	100%	89%
Sun JDK 1.3.1 Server	56%	78%
Sun JDK 1.4.0 Client	90%	82%
Sun JDK 1.4.0 Server	59%	87%
IBM JDK 1.3.0	41%	97%

Tabelle 10.2: Normalisierte Ausführungszeiten der beiden CharReaderDemos

10.4 Dateicache

Gerade weil das Lesen von Dateien nicht gerade zu den allerschnellsten Operationen gehört, kann es sich manchmal lohnen, Dateien zu cachen. Anwendungen, die davon profitieren, sind beispielsweise HTTP- oder FTP-Server, da hier die Festplatte zum Nadelöhr wird und daher so viele parallele Festplattenzugriffe wie möglich vermieden werden müssen.

Um einen effizienten Cache zu implementieren, ist nicht nur die Cacheaustauschstrategie (LRU, Zufall etc. siehe *Kapitel 8.4 Caches*) wichtig, sondern auch, wie überprüft wird, ob eine gecachte Datei nicht mittlerweile manipuliert wurde. Dazu müssen in der Regel das Datum der letzten Veränderung und die Dateigröße überprüft werden. Nur eines von beiden zu überprüfen reicht meist nicht aus, da das Datum der letzten Veränderung oft nur eine Auflösung von einer Sekunde hat und eine Manipulation ja nicht unbedingt in einer Größenveränderung münden muss. Beide Eigenschaften zu überprüfen ist also angebracht. Leider ist dies jedoch nicht ganz umsonst. Abbildung 10.2 zeigt die Zeit, die vergeht, wenn Sie eine Datei vollständig lesen bzw. nur ihre Größe und das Datum der letzten Veränderung in Erfahrung bringen. Auf dem getesteten System (Windows 2000, Sun JDK 1.4.0 Server) musste die Datei mindestens eine Größe von 24 Kbyte haben, um den Zugriff auf die Metadaten zu rechtfertigen. Mit anderen Worten: Ein Cache, der bei *jedem* Zugriff die Gültigkeit einer Datei überprüft, ist auf diesem System dazu bestimmt, ineffizient zu sein, wenn er Dateien kleiner 24 Kbyte aufnimmt.

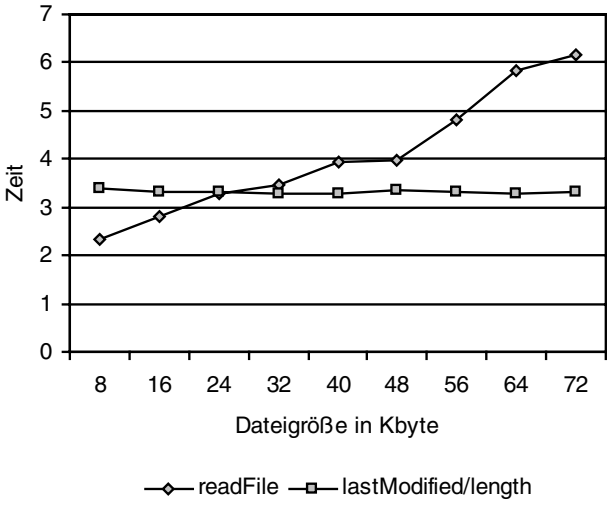

Abbildung 10.2: Vergleich des Zeitaufwandes für das vollständige Lesen einer Datei und lediglich für das Erfragen von Dateilänge sowie Datum der letzten Veränderung

Es lohnt sich also, jeder gecachten Datei eine Mindest-Lebenszeit zu garantieren, während der nicht überprüft wird, ob die Datei verändert wurde. Kleine Dateien sollten evtl. gar nicht erst gecached werden.

Ebenso sollten sehr große Dateien vermutlich nicht gecached werden, da sie zu viel Speicher belegen. Ein effizienter Dateicache muss also folgenden Kriterien genügen:

- Minimale und maximale Dateigröße muss spezifizierbar sein
- Caching-Strategie sollte austauschbar sein

- Cache-Kapazität muss manipulierbar sein
- Cache-Einträge müssen eine Lebensdauer haben, während der ihre Gültigkeit nicht überprüft wird

Ein einfacher, diesen Kriterien entsprechender Dateicache könnte aussehen wie in Listing 10.2.

```
package com.tagtraum.perf.io;

import com.tagtraum.perf.datastructures.Cache;

import java.io.File;
import java.io.IOException;
import java.io.InputStream;

public class FileCache {

   private Cache cache;
   private long minFilesize;
   private long maxFilesize = Long.MAX_VALUE;
   private int timeToLive = -1;

   public FileCache(Cache cache) {
      this.cache = cache;
   }

   public Entry get(File file) throws IOException {
      // Hole Entry aus dem Cache
      Entry entry = (Entry) cache.get(file);
      if (entry != null && !entry.isStale()) {
         // Entry ist nicht stale, also zurück an den Klient
         return entry;
      }
      // Falls es gecached werden soll, cache es
      long length = file.length();
      if (timeToLive != 0 && length > minFilesize
            && length < maxFilesize) {
         // Instanziiere ein FileCache.Entry-Objekt ...
         entry = new ByteArrayFile(file);

         // Alternativ könnten wir auch eine andere Klasse
         // instanziieren:
         // entry = new MemoryMappedFile(file);

         // ... setze seine Lebensdauer ...
         entry.setTimeToLive(timeToLive);
         // ... und registriere es im Cache
         cache.put(file, entry);
      }
      else {
```

```
            // Live-Objekte werden nicht gecached
            entry = new LiveFile(file);
        }
        // Und ab zum Klienten!
        return entry;
    }

    // Mindest-Dateigröße, um gecached zu werden.
    public long getMinFilesize() {
        return minFilesize;
    }

    public void setMinFilesize(long minFilesize) {
        this.minFilesize = minFilesize;
    }

    // Maximale Dateigröße, um gecached zu werden.
    public long getMaxFilesize() {
        return maxFilesize;
    }

    public void setMaxFilesize(long maxFilesize) {
        this.maxFilesize = maxFilesize;
    }

    // Standardzeit, während der die Validtät eines Eintrags
    // nicht überprüft wird.
    public int getTimeToLive() {
        return timeToLive;
    }

    public void setTimeToLive(int timeToLive) {
        this.timeToLive = timeToLive;
    }

    // Cache-Eintrag
    public static interface Entry {

        // Neuer InputStream
        public InputStream getNewInputStream() throws IOException;

        // File-Objekt der gecachten Datei
        public File getFile();

        // Länge des gecachten Inhalts
        public long length();

        // Letzte Modifikation des gecachten Inhalts
        public long lastModified();
```

```
        // Zeit, während der die Validität eines Eintrags
        // nicht überprüft wird.
        public void setTimeToLive(int timeToLive);
        public int getTimeToLive();

        // Zeigt an, ob dieser Cache-Eintrag bereits
        // 'verdorben ' ist.
        public boolean isStale();
    }
}
```

Listing 10.2: Einfacher Dateicache

In `FileCache` haben wir die Cache-Einträge lediglich als Schnittstelle definiert. Wir haben dies deshalb getan, da zum Cachen der Dateien grundsätzlich zwei Mechanismen in Frage kommen.

1. Das Speichern in einem `byte`-Array
2. Das Abbilden der Datei in den Speicher mittels `fileChannel.map()` (erst ab JDK 1.4.0 möglich)

Möglichkeit eins ist trivial. Entsprechender Code befindet sich in Listing 10.3. Möglichkeit Nummer zwei stellt ebenfalls keine allzu große Schwierigkeit dar. Wir müssen lediglich eine anonyme Hilfsklasse schreiben, die aus einem `java.nio.MappedByteBuffer` einen `InputStream` macht. Der Code befindet sich in Listing 10.4.

```
package com.tagtraum.perf.io;

import java.io.*;

class ByteArrayFile implements FileCache.Entry {

    private byte[] content;
    private File file;
    private long length;
    private long lastModified;
    private int timeToLive;
    private long creationTime;
    private long lastValidityCheck;

    public ByteArrayFile(File file) throws IOException {
        this.file = file;
        load(file);
        creationTime = System.currentTimeMillis();
        lastValidityCheck = creationTime;
    }

    // Lädt die Datei in einen byte-Array
    private void load(File file) throws IOException {
```

```
      this.length = file.length();
      this.lastModified = file.lastModified();
      content = new byte[(int)length];
      InputStream in = new FileInputStream(file);
      try {
         int offset = 0;
         while (offset < length) {
            offset += in.read(content, offset, (int)length-offset);
         }
      }
      finally {
         if (in != null) try { in.close(); } catch (IOException ioe) {}
      }
   }

   // Gibt einen neuen ByteInputStream zurück.
   public InputStream getNewInputStream() throws IOException {
      return new ByteArrayInputStream(content);
   }

   public File getFile() {
      return file;
   }

   public long length() {
      return length;
   }

   public long lastModified() {
      return lastModified;
   }

   public void setTimeToLive(int timeToLive) {
      this.timeToLive = timeToLive;
   }

   public int getTimeToLive() {
      return timeToLive;
   }

   public boolean isStale() {
      boolean stale = false;
      long now = System.currentTimeMillis();
      if (timeToLive >= 0
            && lastValidityCheck + (long)timeToLive < now) {
         // Zeit ist abgelaufen, aber vielleicht ist der Inhalt
         // noch nicht verdorben.
         stale = length != file.length()
            || lastModified != file.lastModified();
         if (!stale) lastValidityCheck = now;
```

```
        }
        return stale;
    }
}
```

Listing 10.3: byte-Array basierter Dateicache-Eintrag

```
package com.tagtraum.perf.io;

import java.io.*;
import java.nio.MappedByteBuffer;
import java.nio.channels.FileChannel;

class MemoryMappedFile implements FileCache.Entry {

    private MappedByteBuffer content;
    private File file;
    private long length;
    private long lastModified;
    private int timeToLive;
    private long creationTime;
    private long lastValidityCheck;

    public MemoryMappedFile(File file) throws IOException {
        this.file = file;
        map(file);
        creationTime = System.currentTimeMillis();
        lastValidityCheck = creationTime;
    }

    private void map(File file) throws IOException {
        this.length = file.length();
        this.lastModified = file.lastModified();
        FileInputStream in = new FileInputStream(file);
        try {
            content = in.getChannel().map(
                FileChannel.MapMode.READ_ONLY, 0, length);
        } finally {
            if (in != null) try {
                in.close();
            } catch (IOException ioe) {
                // ignorieren
            }
        }
    }

    public InputStream getNewInputStream() throws IOException {
        // Anonyme innere Klasse, die den MappedByteBuffer in einem
        // InputStream kapselt.
        return new InputStream() {
            private int pos;
```

```
            public int read() throws IOException {
               if (pos == content.limit()) return -1;
               return content.get(pos++);
            }

            public int read(byte b[], int off, int len)
                  throws IOException {
               if (pos == content.limit()) return -1;
               // Wir müssen hier synchronisieren, da mehr als ein
               // InputStream existieren, es aber nur einen
               // MappedByteBuffer gibt und absolute Bulk-Operationen
               // leider nicht unterstützt werden.
               synchronized (content) {
                  content.position(pos);
                  int readBytes = Math.min(len, content.remaining());
                  content.get(b, off, readBytes);
                  pos += readBytes;
                  return readBytes;
               }
            }
         };
      }

   // der Rest ist genau wie in ByteArrayFile (Listing 10.3)
   ...
   }
```

Listing 10.4: Memory-Map basierter Dateicache-Eintrag

Zusätzlich zu den beiden Cache-Eintrags-Klassen müssen wir noch eine Klasse definieren, die zwar die `FileCache.Entry`-Schnittstelle implementiert, aber eigentlich nur einen dünnen Mantel um eine echte Datei darstellt (Listing 10.5). Es wäre nämlich ziemlich ungünstig, wenn eine zwei Gbyte-Datei zunächst in einen `byte`-Array geladen würde ...

Alle drei Dateien würden vermutlich von einer Superklasse profitieren, aus Gründen der Übersichtlichkeit wollen wir darauf jedoch verzichten.

```
package com.tagtraum.perf.io;

import java.io.*;

class LiveFile implements FileCache.Entry {
   private File file;

   public LiveFile(File file) throws IOException {
      this.file = file;
   }
```

```
    // Da wir von einer gecachten Datei erwarten, dass man sie nicht
    // extra puffern muss, geben wir einen BufferedInputStream
    // zurück.
    public InputStream getNewInputStream() throws IOException {
       return new BufferedInputStream(new FileInputStream(file));
    }

    public File getFile() {
       return file;
    }

    public long length() {
       return file.length();
    }

    public long lastModified() {
       return file.lastModified();
    }

    public void setTimeToLive(int timeToLive) {
       throw new UnsupportedOperationException();
    }

    public int getTimeToLive() {
       return -1;
    }

    public boolean isStale() {
       return false;
    }
}
```

Listing 10.5: Mantel um eine echte, ungecachte Datei

Nun haben wir einen Dateicache und fragen uns natürlich, wie schnell er ist. Das Problem ist nur, dass die Mikro-Benchmarks, wie wir sie bisher verwendet haben, für einen Dateicache relativ wenig Aussagekraft haben. Denn jeder Cache sollte grandiose Performance-Zahlen liefern, wenn er innerhalb kurzer Zeit oft nach denselben Inhalten gefragt wird. Was wirklich zählt, ist, ob die Applikation, die den Cache benutzt, tatsächlich schneller wird.

Da wir jedoch keine Applikation zur Hand haben, messen wir die Performance dennoch mit einem Mikro-Benchmark, indem wir Dateien verschiedener Größe wiederholt vollständig lesen. Im ersten Lauf benutzen wir dazu einen gepufferten `FileInputStream`, im zweiten den Cache mit der Klasse `ByteArrayFile` und im dritten Durchlauf den Cache mit der Klasse `MemoryMappedFile`.

Erste Testläufe ergeben, dass für wenige Wiederholungen kein aussagekräftiger Wert messbar ist. Erst bei 1.000fachem Lesen liefert unser Testszenario einigermaßen sinn-

volle Werte. 1.000faches Lesen, das bedeutet, dass die Datei nur einmal *in* den Cache und danach immer wieder *aus* dem Cache gelesen wird. Es ist zweifelhaft, ob dies in einer echten Applikation der Fall wäre.

Wie auch immer. Unser Test ist nicht ganz fruchtlos. Wie Abbildung 10.3 zeigt, ist `ByteArrayFile` schneller als der ungecachte Zugriff und `MemoryMappedFile` wesentlich langsamer als der ungecachte Zugriff. Das ist ein Ergebnis, mit dem ich so nicht gerechnet hätte, da in den Speicher abgebildete Dateien den Zugriff normalerweise beschleunigen sollten.

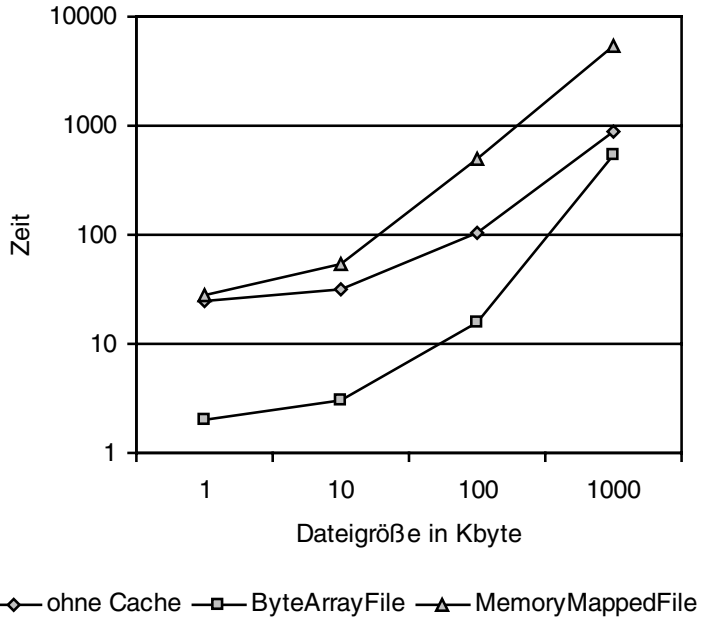

Abbildung 10.3: 1.000faches Lesen einer Datei mit und ohne Cache mit Sun JDK 1.4.0 Server und Windows 2000

Mit Hilfe des Profilers finden wir heraus, dass die `MemoryMappedFile`-Version die meiste Zeit in den Methoden `java.nio.ByteBuffer.get(byte[], int, int)` und `java.nio.DirectByteBuffer.get()` verbringt.

Den Grund dafür offenbart ein schneller Blick in den Quellcode:

```
// Aus Sun JDK 1.4.0, java.nio.ByteBuffer
public ByteBuffer get(byte[] dst, int offset, int length) {
   checkBounds(offset, length, dst.length);
   if (length > remaining())
      throw new BufferUnderflowException();
   int end = offset + length;
   for (int i = offset; i < end; i++)
```

```
        dst[i] = get();
    return this;
}

// Aus Sun JDK 1.4.0, java.nio.DirectByteBuffer
public byte get() {
    return (unsafe.getByte(ix(nextGetIndex())));
}
private long ix(int i) {
    return address + (i << 0);
}

// Aus Sun JDK 1.4.0, java.nio.Buffer
final int nextGetIndex() {
    if (position >= limit)
        throw new BufferUnderflowException();
    return position++;
}
```

Wie unschwer zu erkennen ist, wird jedes Byte einzeln aus dem `DirectByteBuffer` gelesen und in den `byte`-Array geschrieben (siehe *Kapitel 6.2.2 Teure Array-Zugriffe*). Dabei wird einmal vor der gesamten Operation getestet, ob es sich um gültige Indizes handelt, und dann noch einmal bei jedem einzelnen Byte. Kein Wunder, dass die `MemoryMappedFile`-Version so langsam ist. Dies kann sich jedoch in zukünftigen Java Versionen ändern.

10.5 Skalierbare Server

Mit JDK 1.4 verfügt Java endlich über ein besser skalierbares Ein-/Ausgabe-API. Nicht, dass das alte API ein absoluter Fehlschlag ist – gerade Javas eindrucksvoller Erfolg bei Applikationsservern widerlegt dies –, doch einige Eigenschaften des alten APIs führen zu drastischen Einschränkungen. Das größte Übel ist die blockierende Ein-/Ausgabe.

Um Daten über einen Socket zu schreiben, muss die `write()`-Methode eines assoziierten `OuputStreams` aufgerufen werden. Dieser Aufruf kehrt erst zurück, wenn alle zu schreibenden Bytes auch tatsächlich geschrieben sind. Bei vollen Puffern und langsamen Netzwerkverbindungen kann das schon seine Zeit dauern. Um dennoch performante Server mit Java zu realisieren, muss daher mit jedem Socket ein Thread assoziiert werden. So kann ein Thread arbeiten, während ein anderer wegen Ein-/Ausgaben blockiert ist.

Nun sind Threads nicht so schwergewichtig wie echte Prozesse. Abhängig von der Plattform sind sie aber auch keine Ressourcenschoner. Unter anderem implizieren viele Threads auch viele Thread-Kontext-Wechsel – und die sind auch nicht gerade billig.

Um also mit Java skalierbare Server zu bauen, musste ein API her, das die Ehe von Socket und Thread schied. Dies ist mit dem neuen Ein-/Ausgabe-API aus dem Paket `java.nio` nun geschehen.

Wir wollen beispielhaft zeigen, wie man mit dem alten und mit dem neuen API einen simplen Webserver programmiert. Da HTTP kein triviales Protokoll mehr ist, werden wir uns auf einige zentrale Features beschränken. Die vorgestellten Programme sind also weder sicher noch protokollkonform.

10.5.1 Httpd der alten Schule

Schauen wir uns zunächst den HTTP-Server mit dem alten API an (Listing 10.6). Da wir nur eine Klasse zur Implementierung benötigen, ist der grundsätzliche Aufbau schnell erklärt: In der `main()`-Methode wird zunächst ein `ServerSocket` instanziiert und an Port 8080 gebunden. Der designierte WWW-Port 80 ist auf Unix-Systemen dem Systemadministrator vorbehalten und führt leicht zu Konflikten beim Ausprobieren des Beispiels – daher also Port 8080.

Dann werden eine Reihe von `Httpd`-Objekten erzeugt und mit dem gemeinsamen `ServerSocket` initialisiert. Im `Httpd`-Konstruktor sorgen wir dafür, dass alle Instanzen sinnvolle Namen erhalten, setzen das Standard-Protokoll und starten den Server, indem wir die `start()`-Methode seiner Superklasse `Thread` aufrufen. Dies wiederum bewirkt, dass die `run()`-Methode aufgerufen wird, in der sich eine Endlosschleife befindet.

```
package com.tagtraum.perf.httpd;

import java.io.*;
import java.net.*;
import java.util.*;

public class Httpd extends Thread {

   private static int _no; // Instanz-Zähler
   private ServerSocket serverSocket;
   private byte[] buf = new byte[1024 * 8];
   private String protocol;;
   private InputStream in;
   private OutputStream out;
   private String uri;

   // Startet einen Httpd-Thread.
   public Httpd(ServerSocket serverSocket) throws IOException {
      super("Httpd " + (_no++));
      this.serverSocket = serverSocket;
      // default Protokoll-Version
      protocol = "HTTP/0.9";
      start();
   }
```

```
// Wartet am ServerSocket auf eine Verbindung und
// ruft dann handleRequest() auf.
public void run() {
   Socket socket = null;
   while (true) {
      try {
         socket = serverSocket.accept();
         // Nagles Algorithmus für bessere Performance
         // ausschalten
         socket.setTcpNoDelay(true);
         in = socket.getInputStream();
         out = socket.getOutputStream();
         handleRequest();
      } catch (Exception e) {
         // irgendetwas ist wirklich schief gegangen ...
         e.printStackTrace();
      } finally {
         // aufräumen
         if (socket != null) {
            try {
               // dies schließt auch gleichzeitig den In- und
               // Outputstream.
               socket.close();
            } catch (IOException ioe) {
               // ignorieren
            }
         }
         socket = null;
      }
   }
}

// Liest den Request und schickt entweder die Datei
// oder eine Fehlermeldung zurück.
private void handleRequest() throws IOException {
   try {
      // Nur 512 Byte lesen - länger sollte die Zeile ohnehin
      // nicht sein.
      int length = in.read(buf, 0, 512);
      if (length == 512) {
         sendError(414, "Request URI too long.");
         return;
      }

      // Wir nehmen ASCII als Zeichensatz an, daher können wir
      // den schnellen, aber veralteten String-Konstruktor
      // benutzen.
      String requestline = new String(buf, 0, 0, length);
      StringTokenizer st = new StringTokenizer(requestline,
         " \r\n");
      String method = st.nextToken();
      uri = st.nextToken();
```

```
            if (st.hasMoreTokens()) {
               protocol = st.nextToken();
            }
            File file = new File(uri.substring(1));
            if (!method.equals("GET")) {
               sendError(405, "Method " + method
                  + " is not supported.");
            } else if (!file.exists() || file.isDirectory()) {
               sendError(404, "Resource " + uri + " was not found.");
            } else if (!file.canRead()) {
               sendError(403, "Forbidden: " + uri);
            } else {
               sendFile(file);
            }
         } catch (NoSuchElementException nsee) {
            // Wir haben nicht genug Tokens lesen können.
            sendError(400, "Bad request.");
         } catch (Exception e) {
            try {
               sendError(500, "Internal Server Error.");
            } catch (IOException ioe) {
               // ignorieren
            }
         }
      }

      // Sendet eine Fehlermeldung an den Client.
      private void sendError(int httpStatus, String httpMessage)
            throws IOException {
         StringBuffer errorMessage = new StringBuffer(128);
         if (!protocol.equals("HTTP/0.9")) {
            errorMessage.append("HTTP/1.0 " + httpStatus + " "
               + httpMessage + "\r\n\r\n");
         }
         errorMessage.append("<HTML><BODY><H1>" + httpMessage
            + "</H1></BODY></HTML>");
         out.write(errorMessage.toString().getBytes("ASCII"));
         out.flush();
      }

      // Sendet die verlangte Datei an den Client.
      private void sendFile(File file) throws IOException {
         InputStream filein = null;
         try {
            filein = new FileInputStream(file);
            if (!protocol.equals("HTTP/0.9")) {
               // Status code und Header schreiben
               out.write(("HTTP/1.0 200 OK\r\nContent-Type: "
                  + Httpd.guessContentType(uri)
                  + "\r\n\r\n").getBytes("ASCII"));
            }
```

```
            int length = 0;
            while ((length = filein.read(buf)) != -1) {
               out.write(buf, 0, length);
            }
            out.flush();
         } finally {
            if (filein != null)
               try {
                  filein.close();
               } catch (IOException ioe) {
                  // ignorieren
               }
         }
      }

      // Gibt den ContentType der Ressource zurück.
      public static String guessContentType(String uri) {
         // Behelfslösung - sollte normalerweise
         // über eine Konfigurationsdatei erledigt werden.
         uri = uri.toLowerCase();
         if (uri.endsWith(".html") || uri.endsWith(".htm")) {
            return "text/html";
         } else if (uri.endsWith(".txt")) {
            return "text/plain";
         } else if (uri.endsWith(".jpg") || uri.endsWith(".jpeg")) {
            return "image/jpeg";
         } else {
            ...
         } else {
            return "unknown";
         }
      }

      public static void main(String[] args) throws IOException {
         ServerSocket serverSocket = new ServerSocket(8080);
         for (int i = 0; i < Integer.parseInt(args[0]); i++) {
            new Httpd(serverSocket);
         }
      }
   }
```

Listing 10.6: HTTP-Server der alten Schule

In dieser Endlosschleife wird die blockierende `accept()`-Methode des `ServerSockets` aufgerufen. Verbindet sich nun ein Client mit Port 8080 des Servers, gibt die `accept()`-Methode ein `Socket`-Objekt zurück. Mit jedem Socket sind ein `Input`- und ein `OutputStream` assoziiert. Beide werden in der anschließend aufgerufenen `handleRequest()`-Methode benutzt. In ihr wird zunächst der Client-Request gelesen, überprüft und dann eine angemessene Antwort zurückgeschickt. Handelt es sich um einen legitimen Request, wird also die verlangte Datei zurückgeschickt (`sendFile()`). Liegt hingegen

kein legitimer Request vor, erhält der Client eine entsprechende Fehlermeldung als Antwort (`sendError()`). Auf weitere Protokolldetails wollen wir hier aus Platzgründen nicht näher eingehen.

Es stellt sich die Frage, ob diese Art der Implementierung prinzipiell performant ist. Im Großen und Ganzen: Ja. Sicherlich könnte man das Interpretieren des Requests optimieren – die benutzten `StringTokenizer` ließen sich durch selbst geschriebene Tokenizer ersetzen (*Kapitel 4.6 Makro-Benchmarks*) und sicherlich könnte man einen Dateicache benutzen. Immerhin haben wir die für kurze Verbindungen ungeeignete TCP-Verzögerung (Slow-Start-Algorithmus) ausgeschaltet und auch das Senden der Datei erfolgt gepuffert. Doch viel wichtiger ist, dass alle Threads *völlig unabhängig* voneinander arbeiten. Welcher Thread eine neue Verbindung akzeptiert, wird über die `accept()`-Methode betriebssystemnah und somit schnell entschieden. Und über das `ServerSocket`-Objekt hinaus teilen die Threads keinerlei Ressourcen, die evtl. synchronisiert werden müssten. Schnell ist diese Lösung also – jedoch nicht beliebig skalierbar.

10.5.2 Nicht-blockierender Httpd

Schauen wir uns also die Lösung zwei mit dem neuen Ein-/Ausgabe-API an. Sie ist ein wenig komplizierter und erfordert das Zusammenspiel verschiedener Threads.

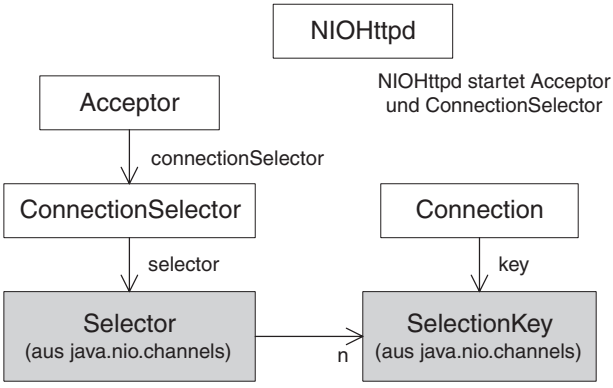

Abbildung 10.4: Klassendiagramm für NIOHttpd

Lösung zwei besteht aus vier Klassen (Abbildung 10.4):

- `NIOHttpd` (Listing 10.7)
- `Acceptor` (Listing 10.8)
- `ConnectionSelector` (Listing 10.9)
- `Connection` (Listing 10.10)

`NIOHttpd` dient hauptsächlich zum Starten des Servers. Genau wie in `Httpd` wird in der `main()`-Methode ein Server-Socket an Port 8080 gebunden (Listing 10.7, Zeile 25,26). Der wichtige Unterschied: Es handelt sich hier um einen `java.nio.channels.ServerSocketChannel`. Diesen öffnen wir zunächst über eine Fabrikmethode und binden ihn dann explizit per `bind()` an den Port. Anschließend erzeugen wir jeweils einen `ConnectionSelector` und einen `Acceptor`. Dabei wird der `ConnectionSelector` beim `Acceptor` registriert. Dem `Acceptor` wird außerdem der `ServerSocketChannel` übergeben (Listing 10.7, Zeile 28-32).

```
01 package com.tagtraum.perf.httpd;
02
03 import java.io.IOException;
04 import java.net.InetSocketAddress;
05 import java.nio.channels.ServerSocketChannel;
06
07 public class NIOHttpd {
08
09    // Hilfsmethode zum Ermitteln des ContentTyps einer Ressource
10    public static String guessContentType(String uri) {
11       uri = uri.toLowerCase();
12       if (uri.endsWith(".html") || uri.endsWith(".htm")) {
13          return "text/html";
14       } else if (uri.endsWith(".txt")) {
15          ...
16       } else {
17          return "unknown";
18       }
19    }
20
21    // Startet den Http-Daemon mit 2n Threads.
22    public static void main(String[] args) throws IOException {
23       ServerSocketChannel serverSocketChannel
24          = ServerSocketChannel.open();
25       serverSocketChannel.socket().bind(
26          new InetSocketAddress(8080));
27       for (int i = 0; i < Integer.parseInt(args[0]); i++) {
28          ConnectionSelector connectionSelector
29             = new ConnectionSelector();
30          Acceptor acceptor = new Acceptor(serverSocketChannel,
31             connectionSelector);
32       }
33    }
34 }
```

Listing 10.7: Klasse NIOHttpd

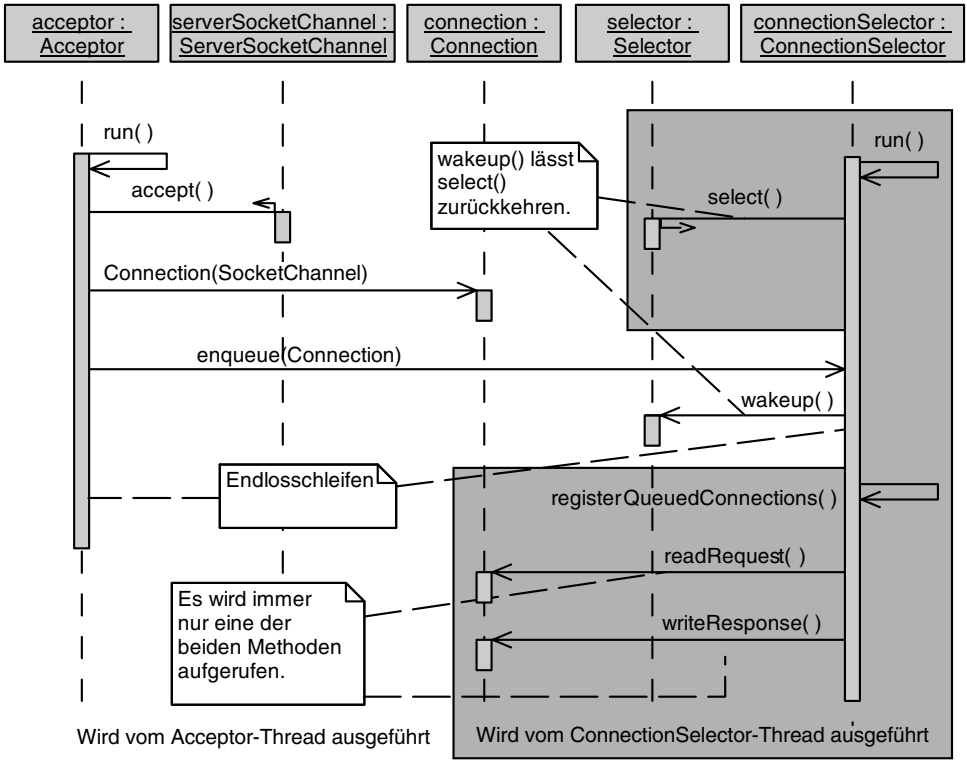

Abbildung 10.5: Sequenzdiagramm für NIOHttpd

Abbildung 10.5 zeigt das Zusammenspiel der beiden Threads `Acceptor` und `ConnectionSelector` im Überblick. Um es genau zu verstehen, wollen wir zunächst den `Acceptor` (Listing 10.8) näher betrachten. Seine Aufgabe ist es, eingehende Verbindungen anzunehmen und sie beim `ConnectionSelector` zu registrieren. Noch im Konstruktor wird daher die `start()`-Methode der Superklasse `Thread` aufgerufen (Listing 10.8, Zeile 18). In `run()` befindet sich die erforderliche Endlosschleife. In ihr wird genau wie in `Httpd` eine blockierende `accept()`-Methode aufgerufen (Listing 10.8, Zeile 27), die schließlich ein `SocketChannel`-Objekt zurückgibt. Nur ist es diesmal die `accept()`-Methode eines `ServerSocketChannels` anstatt eines `ServerSockets`. Mit dem zurückgegebenen `SocketChannel` als Argument wird ein `Connection`-Objekt erzeugt und mittels der `enqueue()`-Methode beim `ConnectionSelector` registriert (Listing 10.8, Zeile 28,29; Abbildung 10.6).

Um es noch einmal zusammenzufassen: der `Acceptor` macht nichts anderes als in einer Endlosschleife Verbindungen anzunehmen und diese beim `ConnectionSelector` zu registrieren.

```
01 package com.tagtraum.perf.httpd;
02
03 import java.net.Socket;
04 import java.nio.channels.ServerSocketChannel;
05 import java.nio.channels.SocketChannel;
06
07 class Acceptor extends Thread {
08
09    private static int _no; // Instanz-Zähler
10    private ServerSocketChannel serverSocketChannel;
11    private ConnectionSelector connectionSelector;
12
13    public Acceptor(ServerSocketChannel serverSocketChannel,
14          ConnectionSelector connectionSelector) {
15       super("Acceptor " + (_no++));
16       this.serverSocketChannel = serverSocketChannel;
17       this.connectionSelector = connectionSelector;
18       start();
19    }
20
21    // Akzeptiert Verbindungen und registriert diese mittels
22    // ConnectionSelector.enqueue(Connection).
23    public void run() {
24       while (true) {
25          SocketChannel socketChannel = null;
26          try {
27             socketChannel = serverSocketChannel.accept();
28             connectionSelector.enqueue(
29                new Connection(socketChannel));
30          } catch (Exception e) {
31             e.printStackTrace();
32             // aufräumen, falls nötig
33             if (socketChannel != null) {
34                try {
35                   socketChannel.close();
36                } catch (Exception ee) {
37                   // ignorieren
38                }
39             }
40          }
41       }
42    }
43 }
```

Listing 10.8: Klasse Acceptor

Genau wie der `Acceptor` ist der `ConnectionSelector` (Listing 10.9) ein Thread. Er dient dazu, Verbindungen auszuwählen, die gerade für Ein- bzw. Ausgaben bereit sind. In seinem Konstruktor werden eine Queue erzeugt und ein `java.nio.channels.Selector` mittels der Fabrikmethode `Selector.open()` geöffnet (Listing 10.9, Zeile 17,18). Dieser

`selector` ist der Dreh- und Angelpunkt unseres Servers. Bei ihm können wir Verbindungen registrieren und auf Anfrage eine Liste derjenigen Verbindungen zurückbekommen, die gerade zum Lesen oder Schreiben von Daten bereit sind. Der `ConnectionSelector` benutzt das `selector`-Objekt entsprechend.

```
01 package com.tagtraum.perf.httpd;
02
03 import java.io.IOException;
04 import java.nio.channels.SelectionKey;
05 import java.nio.channels.Selector;
06 import java.util.*;
07
08 class ConnectionSelector extends Thread {
09
10    private static int _no; // Instanz-Zähler
11    private Selector selector;
12    private List queue;
13
14    // Instanziiert und startet diesen ConnectionSelector.
15    public ConnectionSelector() throws IOException {
16       super("ConnectionSelector " + (_no++));
17       selector = Selector.open();
18       queue = new ArrayList();
19       start();
20    }
21
22    // Queuet eine Verbindung und ruft selector.wakeup() auf,
23    // damit ein SelectionKey für sie erzeugt und registriert
24    // werden kann.
25    public void enqueue(Connection connection) {
26       synchronized (queue) {
27          queue.add(connection);
28       }
29       // wakeup sorgt dafür, dass select() aufwacht und sich um
30       // gequeuete Verbindungen kümmert.
31       selector.wakeup();
32    }
33
34    // Registriert alle gequeueten Verbindungen beim Selector.
35    private void registerQueuedConnections() throws IOException {
36       // der synchronized Block ist ein Nadelöhr, daher sollte
37       // er wenn möglich vermieden werden.
38       if (!queue.isEmpty()) {
39          synchronized (queue) {
40             while (!queue.isEmpty()) {
41                Connection connection
42                   = (Connection) queue.remove(queue.size() - 1);
43                connection.register(selector);
44             }
45          }
```

```
46       }
47    }
48
49    // Ruft selector.select()in einer Endlosschleife auf.
50    // Kehrt der Aufruf von select() zurück, werden zunächst
51    // gequeuete Verbindungen beim Selector registriert.
52    // Anschließend werden für bereite Kanäle die entsprechenden
53    // Verbindungs-Arbeitsmethoden aufgerufen.
54    public void run() {
55       while (true) {
56          try {
57             int i = selector.select();
58             registerQueuedConnections();
59             if (i > 0) {
60                Set set = selector.selectedKeys();
61                Iterator connectionIterator = set.iterator();
62                while (connectionIterator.hasNext()) {
63                   SelectionKey key
64                      = (SelectionKey) connectionIterator.next();
65                   Connection connection
66                      = (Connection) key.attachment();
67                   try {
68                      if (key.interestOps()
69                            ==SelectionKey.OP_READ) {
70                         connection.readRequest();
71                      } else {
72                         connection.writeResponse();
73                      }
74                   } catch (IOException ioe) {
75                      connection.close();
76                   } catch (Throwable t) {
77                      connection.close();
78                      t.printStackTrace();
79                   }
80                }
81             }
82          } catch (Throwable t) {
83             t.printStackTrace();
84          }
85       }
86    }
87 }
```

Listing 10.9: Klasse ConnectionSelector

Nachdem im Konstruktor die `start()`-Methode aufgerufen wurde (Listing 10.9, Zeile 19), wird die Endlosschleife in der `run()`-Methode ausgeführt. In ihr rufen wir die `select()`-Methode des `selectors` auf (Listing 10.9, Zeile 57). Diese Methode blockiert so lange, bis entweder mindestens eine der registrierten Verbindungen für Ein-/Ausgabe-Operationen bereit ist oder ein Aufruf der `wakeup()`-Methode des `selector`-Objekts erfolgt.

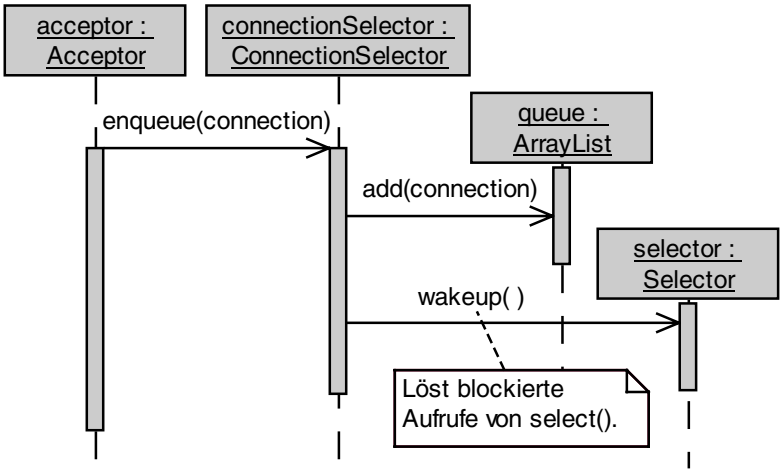

Abbildung 10.6: Sequenzdiagramm der enqueue()-Methode

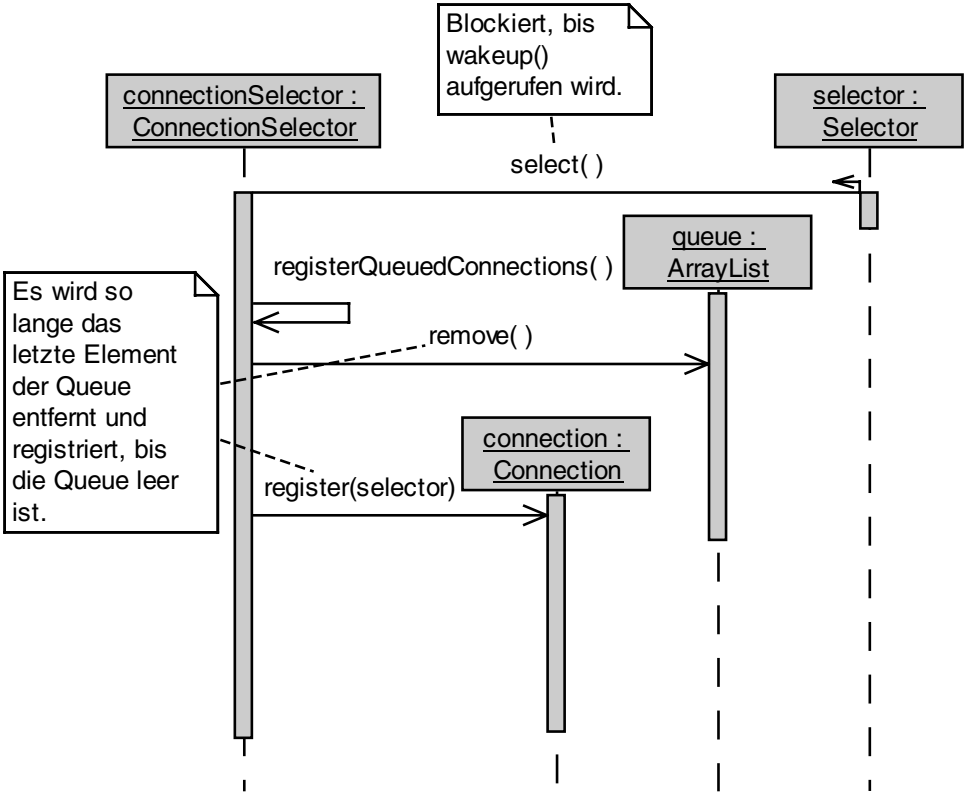

Abbildung 10.7: Sequenzdiagramm der registerQueuedConnections()-Methode

Es ist wichtig zu verstehen, dass kein `Acceptor`-Thread Verbindungen beim `selector` registrieren kann, während der `ConnectionSelector`-Thread die Methode `select()` ausführt, da die entsprechenden Methoden synchronisiert sind. Daher benutzen wir eine Queue, in die der `Acceptor`-Thread angenommene Verbindungen mit der Methode `enqueue()` einstellt (Listing 10.9, Zeile 25f.; Abbildung 10.6). Anschließend ruft er die `wakeup()`-Methode des `selectors` auf (Listing 10.9, Zeile 31). Dies wiederum löst den `ConnectionSelector`-Thread aus seiner `select()`-Blockade und erlaubt ihm, die Verbindungen aus der Queue beim nun nicht mehr blockierten `selector` zu registrieren. Genau dies geschieht in der `registerQueuedConnections()`-Methode (Listing 10.9, Zeile 35f.; Abbildung 10.7).

Selektor-Registrierung über Schlüssel

An dieser Stelle müssen wir ein wenig vorgreifen und einen kurzen Blick auf die `register()`-Methode der `Connection`-Klasse werfen (Listing 10.10, Zeile 45f.). Bisher haben wir vereinfachend davon gesprochen, dass eine Verbindung bei einem `selector` registriert wird. Tatsächlich wird jedoch ein `java.nio.channels.SocketChannel`-Objekt bei einem `selector` registriert. Und zwar nur für ausgewählte Ein-/Ausgabe-Operationen. Zurück erhält man einen `java.nio.channels.SelectionKey`. Diesem Schlüssel wiederum kann man mittels der `attach()`-Methode beliebige Objekte zuordnen. Um mit dem Schlüssel zur Verbindung zu gelangen, hängen wir das `Connection`-Objekt selbst an den Schlüssel. Somit können wir indirekt über den Schlüssel tatsächlich das Verbindungs-Objekt vom `Selector` erhalten.

Zurück zum `ConnectionSelector`. Der Rückgabewert der `select()`-Methode gibt an, wie viele Verbindungen für Ein-/Ausgabe-Operationen bereit sind. Ist dies bei keiner Verbindung der Fall, sparen wir uns den Rest und kehren zurück in die `select()`-Endlosschleife. Andernfalls iterieren wir über die Selektionsschlüssel (Listing 10.9, Zeile 60f.). Diese erhalten wir im Set von der Methode `selectedKeys()`. Über die `attachment()`-Methode des Schlüssels gelangen wir an das zugehörige `Connection`-Objekt und rufen dessen `readRequest()`- bzw. `writeResponse()`-Methode auf. Um welche Methode es sich handelt, hängt davon ab, ob sich die Verbindung für Lese- oder Schreib-Operationen registriert hat (Listing 10.9, Zeile 68f.).

```
01 package com.tagtraum.perf.httpd;
02
03 import java.io.*;
04 import java.net.Socket;
05 import java.nio.ByteBuffer;
06 import java.nio.channels.*;
07 import java.util.*;
08
09 // Repräsentiert die Verbindung, in deren Verlauf zunächst
10 // einmal der Request gelesen und dann ein Response
```

```
11 // geschrieben wird.
12 class Connection {
13
14    private SocketChannel socketChannel;
15    private ByteBuffer requestLineBuffer;
16    private ByteBuffer responseLineBuffer;
17    private int endOfLineIndex;
18    private SelectionKey key;
19    private FileChannel fileChannel;
20    private long filePos;
21    private long fileLength;
22    private int httpStatus;
23    private String httpMessage;
24    private String uri;
25    private String protocol;
26
27    // Initialisiert diese Verbindung mit einem SocketChannel.
28    public Connection(SocketChannel socketChannel)
29          throws IOException {
30       // Nagles Algorithmus für bessere Performance
31       // ausschalten
32       socketChannel.socket().setTcpNoDelay(true);
33       // der Kanal soll nicht blockieren
34       socketChannel.configureBlocking(false);
35       requestLineBuffer = ByteBuffer.allocate(512);
36       // Default http status code: OK
37       httpStatus = 200;
38       // Default http status message
39       httpMessage = "OK";
40       // Default Protokoll-Version
41       protocol = "HTTP/0.9";
42    }
43
44    // Registriert diese Verbindung bei dem übergebenen Selector.
45    public void register(Selector selector) throws IOException {
46       key = socketChannel.register(selector,
47          SelectionKey.OP_READ);
48       // Hinterlege die Verbindung im Schlüssel
49       key.attach(this);
50    }
51
52    // Liest den Request. Falls etwas schief geht, wird ein
53    // Fehlercode gesetzt. Ist der Request vollstandig gelesen,
54    // wird prepareForResponse() aufgerufen.
55    public void readRequest() throws IOException {
56       try {
57          if (!requestLineBuffer.hasRemaining()) {
58             setError(414, "Request URI too long.");
59             prepareForResponse();
60             return;
61          }
```

```
62          socketChannel.read(requestLineBuffer);
63          if (!isRequestLineRead()) {
64             return;
65          }
66          requestLineBuffer.flip();
67          byte[] b = new byte[endOfLineIndex];
68          requestLineBuffer.get(b);
69          String requestline = new String(b, 0);
70          StringTokenizer st
71             = new StringTokenizer(requestline, " \r\n");
72          String method = st.nextToken();
73          uri = st.nextToken();
74          File file = new File(uri.substring(1));
75          if (st.hasMoreTokens()) {
76             protocol = st.nextToken();
77          }
78          if (!method.equals("GET")) {
79             setError(405, "Method " + method
80                + " is not supported.");
81          } else if (!file.exists() || file.isDirectory()) {
82             setError(404, "Resource " + uri
83                + " was not found.");
84          } else if (!file.canRead()) {
85             setError(403, "Forbidden: " + uri);
86          } else {
87             fileLength = file.length();
88             fileChannel
89                = new FileInputStream(file).getChannel();
90          }
91          prepareForResponse();
92       } catch (NoSuchElementException nsee) {
93          // Wir haben nicht genug Tokens lesen können.
94          setError(400, "Bad request.");
95       } catch (Exception e) {
96          // Es ist etwas außerplanmäßig schief gegangen
97          setError(500, "Internal Server Error.");
98          prepareForResponse();
99          e.printStackTrace();
100      }
101   }
102
103   // Legt einen Buffer an, der die Response-Zeile, die Header
104   // und im Falle eines Fehlers eine HTML-Nachricht enthält.
105   private void prepareForResponse() throws IOException {
106      StringBuffer responseLine = new StringBuffer(128);
107      // Response-Zeile nur bei Http >= 1.0 schreiben
108      if (!protocol.equals("HTTP/0.9")) {
109         responseLine.append("HTTP/1.0 " + httpStatus + " "
110            + httpMessage + "\r\n");
111         // Im Fehlerfall benötigen wir keine Header
112         if (httpStatus != 200) {
```

```
113             responseLine.append("\r\n");
114          } else {
115             // Header für die Datei
116             responseLine.append("Content-Type: "
117                + NIOHttpd.guessContentType(uri) + "\r\n\r\n");
118          }
119       }
120       if (httpStatus != 200) {
121          // Fehlernachricht für den Nutzer
122          responseLine.append("<HTML><BODY><H1>" + httpMessage
123             + "</H1></BODY></HTML>");
124       }
125       responseLineBuffer = ByteBuffer.wrap(responseLine
126          .toString().getBytes("ASCII"));
127       key.interestOps(SelectionKey.OP_WRITE);
128       key.selector().wakeup();
129    }
130
131    // Gibt an, ob die Request-Zeile bereits vollständig gelesen
132    // wurde.
133    private boolean isRequestLineRead() {
134       for (; endOfLineIndex < requestLineBuffer.limit();
135             endOfLineIndex++) {
136          if (requestLineBuffer.get(endOfLineIndex) == '\r')
137             return true;
138       }
139       return false;
140    }
141
142    // Schreibt zunächst den responseLineBuffer und dann ggfs.
143    // die verlangte Datei zum Client. Nachdem alle Daten
144    // geschrieben wurden, wird der Selektions-Schlüssel
145    // gecancelt und der Kanal geschlossen.
146    public void writeResponse() throws IOException {
147       // Zunächst mal den response buffer schreiben
148       if (responseLineBuffer.hasRemaining()) {
149          socketChannel.write(responseLineBuffer);
150       }
151       // Wenn der Buffer vollständig geschrieben wurde,
152       // sind wir entweder fertig (im Fehlerfall) oder
153       // müssen noch die Datei hinterherschicken
154       if (!responseLineBuffer.hasRemaining()) {
155          if (httpStatus != 200) {
156             close();
157          } else {
158             filePos += fileChannel.transferTo(filePos,
159                (int) Math.min(64 * 1024, fileLength - filePos),
160                socketChannel);
161             if (filePos == fileLength) {
162                close();
163             }
```

```
164          }
165       }
166    }
167
168    // Setzt einen Fehler.
169    private void setError(int httpStatus, String httpMessage) {
170       this.httpStatus = httpStatus;
171       this.httpMessage = httpMessage;
172    }
173
174    // Cancelt den Selektions-Schlüssel und schließt alle
175    //offenen Kanäle.
176    public void close() {
177       try {
178          if (key != null) key.cancel();
179       } catch (Exception e) {
180          // ignorieren
181       }
182       try {
183          if (socketChannel != null) socketChannel.close();
184       } catch (Exception e) {
185          // ignorieren
186       }
187       try {
188          if (fileChannel != null) fileChannel.close();
189       } catch (Exception e) {
190          // ignorieren
191       }
192    }
193 }
```

Listing 10.10: Klasse Connection

Und damit kommen wir nun endgültig zur `Connection`-Klasse (Listing 10.10). Sie repräsentiert die Verbindung und kapselt zudem alle Protokollspezifika. Im Konstruktor wird zunächst `SocketChannel` in den nicht-blockierenden Modus versetzt (Listing 10.10, Zeile 34). *Dies ist essentiell für diesen Server!* Anschließend werden noch ein paar Standardwerte gesetzt sowie der Puffer `requestLineBuffer` für den Request alloziert. Da das Allozieren von systemnahen, direkten Puffern vergleichsweise teuer ist und wir für jede Verbindung einen neuen Puffer erzeugen, benutzen wir `java.nio.ByteBuffer.allocate()` anstelle von `ByteBuffer.allocateDirect()`. Würden wir die Puffer wieder verwenden, könnte sich jedoch ein direkter Puffer bezahlt machen.

Nachdem die Initialisierung erledigt und der `SocketChannel` zum Lesen bereit ist, wird vom `ConnectionSelector` die `readRequest()`-Methode aufgerufen. Mit `socketChannel.read(requestLineBuffer)` werden so viele Bytes in den Puffer gelesen, wie gerade verfügbar sind. Falls die gesamte Request-Zeile nicht gelesen werden kann, kehren wir zum aufrufenden `ConnectionSelector`-Objekt zurück und lassen so eine andere Verbin-

dung zum Zuge kommen. Ist jedoch die gesamte Zeile gelesen, interpretieren wir sie wie schon in `Httpd`. Handelt es sich um einen legitimen Request, erzeugen wir einen `java.nio.Channels.FileChannel` für die verlangte Datei und rufen die Methode `prepareForResponse()` auf.

`prepareForResponse()` bastelt die Response-Zeile, evtl. benötigte Header sowie – falls nötig – eine Fehlermeldung zusammen und hinterlegt diese Daten in `responseLineBuffer`. Hierbei handelt es sich wiederum um einen `ByteBuffer`, der jedoch lediglich ein dünner Wrapper um einen `byte`-Array ist und mit der Fabrikmethode `ByteBuffer.wrap(byte[])` erzeugt wurde. Nachdem wir die zu schreibenden Daten erzeugt haben, müssen wir dem `ConnectionSelector` noch mitteilen, dass wir von nun an Daten schreiben anstatt lesen wollen. Dies erreichen wir, indem wir die Methode `interestOps(SelectionKey.OP_WRITE)` des Selektions-Schlüssels aufrufen. Um sicherzugehen, dass der Selektor dies möglichst schnell mitbekommt, rufen wir anschließend noch die `wakeup()`-Methode auf.

Nun ruft der `ConnectionSelector` die `writeResponse()`-Methode auf. Zuerst wird der `responseLineBuffer` in den Socket-Kanal geschrieben. Gelingt dies vollständig und müssen wir noch die verlangte Datei hinterherschicken, rufen wir die `transferTo()`-Methode des zuvor geöffneten `FileChannels` auf. Übertragen werden in jedem Fall nur so viele Bytes, wie gerade in den Zielkanal geschrieben werden können. Dennoch muss hier eine Grenze gesetzt werden, um für Fairness zwischen verschiedenen Verbindungen zu sorgen.

Sind alle Daten übertragen, wird mit der `close()`-Methode aufgeräumt. Wichtig ist hier das De-Registrieren der Verbindung beim `ConnectionSelector`. Dies geschieht durch Aufruf der `cancel()`-Methode des Selektions-Schlüssels.

Wiederum stellt sich die Frage, ob die Implementierung grundsätzlich performant ist. Und wiederum können wir klar antworten: Ja.

Im Prinzip reichen je ein `Acceptor`- und ein `ConnectionSelector`-Thread, um gleichzeitig beliebig viele Verbindungen aufrechtzuerhalten. Damit glänzt diese Implementierung in der Kategorie Skalierbarkeit. Da jedoch die beiden Threads über die synchronisierte Methode `enqueue()` miteinander kommunizieren, können sie sich gegeneinander ausbremsen. Es bieten sich zwei Auswege aus dieser Situation an:

1. Eine bessere Implementierung der Queue
2. Mehrere `Acceptor`/`ConnectionSelector`-Paare

Lösung eins ließe sich durch eine `LinkedQueue` nach Doug Lea [Lea99, S.130] verwirklichen. Diese Datenstruktur zeichnet sich dadurch aus, dass sie Anfang und Ende der Warteschlange mit verschiedenen Locks sichert und sich daher einfügende und leerende Threads nicht gegenseitig blockieren. Nur wenn die Schlange leer ist, besteht die Möglichkeit der gegenseitigen Blockade. Die ließe sich jedoch durch eine Extra-Abfrage umgehen.

Im Vergleich zu diesem eleganten Ansatz fällt Lösung zwei schon fast in die Kategorie »Rohe Gewalt«. Über mehrere `Acceptor`/`ConnectionSelector`-Paare wird die Last verteilt und das Synchronisierungsproblem zwar nicht beseitigt, jedoch gelindert. Leider entstehen dabei auch zusätzliche Kosten für Kontext-Wechsel. Im Vergleich zu `Httpd` benötigt man jedoch bei weitem nicht so viele Threads. Und wenn man `NIOHttpd` auf einem Multiprozessor-System betreiben möchte, empfiehlt es sich sogar, mehrere Paare zu starten.

Nachteilig wirkt sich außerdem für `NIOHttpd` aus, dass immer wieder neue `Connection`-Objekte samt ihrer Puffer erzeugt werden. Dies führt zu einer Mehrbelastung verursacht durch die Speicherbereinigung.

10.5.3 Vergleichende Rechenspiele

Es stellt sich die Frage, wie viel besser `NIOHttpd` gegenüber `Httpd` skaliert. Statt zu messen, wollen wir dazu ein paar Überlegungen anstellen. Vorweggeschickt: Auch wenn die Zahlen und Formeln einen präzisen Eindruck machen – dies wird bei weitem kein exakter Vergleich. Es werden lediglich die zugrunde liegenden Konzepte gegeneinander abgeschätzt. Dabei lassen wir einflussreiche Randbedingungen wie Thread-Synchronisierung, Kontext-Switches, Paging, Festplattengeschwindigkeit und Caches völlig außer Acht.

Zunächst schätzen wir ab, wie lange es wohl dauert, r gleichzeitige Requests nach Dateien der Größe s Bytes bei einer Client-Anbindung mit einer Bandbreite von b Byte/Sekunde zu verarbeiten. Es ist offensichtlich, dass dies bei `Httpd` unmittelbar von der Anzahl der Threads t abhängt, da nur t Requests gleichzeitig behandelt werden können. Über den Daumen gepeilt, könnte die Rechnung aussehen wie in folgender Formel: c seien Fixkosten wie Parsen etc., die bei jedem Request bezahlt werden müssen. Wir nehmen zudem an, dass wir die Daten schneller von der Platte lesen als über den Socket schreiben können und die CPU nicht voll ausgelastet wird. Daher fließen die Geschwindigkeiten von Festplatte/Cache und Prozessor nicht in die Rechung ein.

$$l_{Httpd} = \frac{s \times r}{b \times \min(t, r)} + r \times c$$

`NIOHttpd` ist hingegen nicht von t abhängig. Die Zeit l hängt bei idealisierten Randbedingungen also hauptsächlich von der Anbindung des Clients b, der Größe der Datei s sowie von den bereits erwähnten Fixkosten c ab. Daraus resultiert:

$$l_{NIOHttpd} = \frac{s}{b} + r \times c$$

$$d = \frac{l_{NIOHttpd}}{l_{Httpd}}$$

Interessant als Maßzahl für uns ist nun der Quotient d. Er charakterisiert das Verhältnis zwischen `NIOHttpd` und `Httpd`.

Nach genauerem Hinsehen (... und ein paar Datenreihen) fällt auf, dass d bei konstanten s, b, t und c für große r gegen einen Grenzwert wächst. Dieser lässt sich leicht nach folgender Formel berechnen.

$$\lim_{r\to\infty} d_r = \frac{c}{c + \frac{s}{bt}}$$

Hieraus folgt, dass – neben der Anzahl der Threads und den fixen Kosten – die Dauer s/b der Verbindung erheblichen Einfluss auf d hat. Je länger die Verbindungen bestehen, desto kleiner ist d und umso größer ist der Vorteil von `NIOHttpd` gegenüber `Httpd`. So kann `NIOHttpd` rechnerisch unter bestimmten Bedingungen (c=10 ms, t=100, s=1 Mbyte) für einen einfachen ISDN-Kanal mit 8 Kbyte/s Bandbreite bis zu 126-mal schneller sein als `Httpd`. Dies gilt insbesondere für große Dateien und damit lang andauernde Verbindungen. Ist die Verbindung hingegen schnell, beispielsweise ein lokales 100-Mbit-Netz, lassen sich bei großen Dateien gerade mal 10% herausschlagen und bei kleinen Dateien gibt es kaum einen Unterschied.

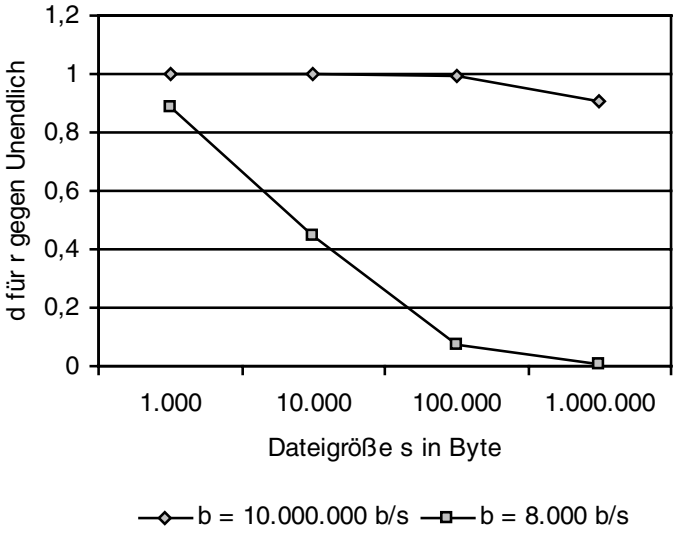

Abbildung 10.8: d bei c=10ms und t=100

Diese Berechnungen setzen voraus, dass die Fixkosten bei `NIOHttpd` und `Httpd` in etwa gleich sind und keine neuen Kosten durch die Art und Weise der Implementierung eingeführt wurden. Wie oben erwähnt: Der angeführte Vergleich gilt nur unter stark idealisierten Bedingungen.

Er reicht jedoch aus, ein Gefühl dafür zu vermitteln, unter welchen Bedingungen sich eher das eine oder andere Konzept lohnt bzw. wie groß die Unterschiede sein können. Hierbei sei noch angeführt, dass zwar die meisten Dateien im WWW eher klein sind, dass HTTP-1.1-Clients jedoch standardmäßig versuchen, eine Verbindung für mehr als eine Datei zu benutzen und entsprechend länger offen zu halten (Keep-Alive bzw. persistente Verbindungen). Nicht selten werden deshalb Verbindungen aufrechterhalten, über die nie wieder Daten übertragen werden. Hierdurch würden bei einem Server mit einem Thread pro Verbindung massiv Threads und somit kostbare Ressourcen gebunden. Das heißt gerade für HTTP kann die Skalierbarkeit von Servern durch das neue Ein-/Ausgabe-API dramatisch erhöht werden kann.

11 RMI und Serialisierung

RMI ist eine jener Technologien, die das Leben in einer verteilten Umgebung wesentlich leichter machen. Wie jede Erleichterung hat jedoch auch RMI seinen Preis. Gegenüber einem eigenen, selbst geschriebenen Protokoll ist das mehr oder minder generische RMI meist etwas schwerfällig. Dafür bietet es einen gewissen Komfort. Es stellt zudem den Standard dar, weswegen wir hier keine selbst geschriebenen Protokolle diskutieren wollen.

Es gibt im Wesentlichen drei Aspekte in Zusammenhang mit RMI, die sich zu betrachten lohnen, um die Performance zu steigern.

1. Serialisierung
2. Latenzzeiten
3. Verteilte Speicherbereinigung (Distributed Garbage Collection)

Auf alle drei Aspekte werden wir im Folgenden eingehen.

11.1 Effiziente Serialisierung

Wenn Sie eine Methode eines entfernten Objektes (*RemoteObject*) aufrufen, müssen die Argumente und der Rückgabewert für die Übertragung serialisiert und anschließend wieder deserialisiert werden. Dies ist zumindest immer dann nötig, wenn diese Objekte nicht selbst entfernte Objekte sind. Zum Serialisieren und Deserialisieren verfügt Java über einen eingebauten Mechanismus, der sich leicht benutzen lässt, indem Sie einfach die Schnittstelle `java.io.Serializable` in Ihren zu serialisierenden Klassen implementieren. Bei `Serializable` handelt es sich übrigens um ein Markierungs-Interface, das keine Methoden vorgibt. Allein die Tatsache, dass eine Klasse dieses Interface implementiert, führt zu einer speziellen Behandlung.

Wenn ein `Serializable`-Objekte serialisiert wird, erstellt der Serialisierungsmechanismus automatisch eine Darstellung des Objektes inklusive aller Attribute, die wiederum auch `Serializable` implementieren müssen. Es wird also die serielle Darstellung eines ganzen Objektbaumes erzeugt.

Wir wollen das an zwei einfachen Beispielen genauer untersuchen.

11.1.1 Datenmenge verkleinern

`InternationalDate` sei ein unveränderbares Objekt, das ein Datum in allen unterstützten Sprachen formatiert und die String-Darstellung in einer Tabelle vorhält (Listing 11.1).

```
package com.tagtraum.perf.serialization;

import java.io.Serializable;
import java.text.SimpleDateFormat;
import java.util.*;

public class InternationalDate1 implements Serializable {
   private Date date;
   private Map map;

   public InternationalDate1() {
      this(System.currentTimeMillis());
   }

   public InternationalDate1(long time) {
      date = new Date(time);
      buildInternationalStrings();
   }

   private void buildInternationalStrings() {
      map = new HashMap();
      Locale locales[] = Locale.getAvailableLocales();
      for (int i = 0; i < locales.length; i++) {
         SimpleDateFormat format = new SimpleDateFormat(
            "EEEE d MMM yyyy G HH:mm:ss,SSS zzzz", locales[i]);
         String formattedTime = format.format(date);
         map.put(locales[i], formattedTime);
      }
   }

   public Date getDate() {
      return date;
   }

   public String get(Locale locale) {
      return (String) map.get(locale);
   }
}
```

Listing 11.1: Internationales Datum

Wenn wir dieses Objekt serialisieren, werden 13.907 Byte[1] geschrieben. Das ist ein bisschen viel für ein einfaches Datum. Offensichtlich liegt dies daran, dass sämtliche Datumsstrings mitserialisiert werden. Natürlich ist das nicht nötig, da wir die Strings leicht neu berechnen können. Listing 11.2 zeigt eine verbesserte Version, in der die Tabelle `map` mit dem Schlüsselwort `transient` gekennzeichnet und die Methode `readObject()` implementiert ist. Alle Attribute, die mit `transient` gekennzeichnet sind, werden beim Serialisieren übersprungen. In der Regel ist es Aufgabe der privaten `readObject()`-Methode, transiente (flüchtige) Attribute aus den persistenten Attributen wiederherzustellen.

Das sorgfältige Kennzeichnen von Attributen als transient ist eine der wichtigsten Techniken, um die Größe serialisierter Objekte zu verringern.

```
package com.tagtraum.perf.serialization;

import java.io.Serializable;
import java.io.ObjectInputStream;
import java.io.IOException;
import java.util.*;
import java.text.SimpleDateFormat;

public class InternationalDate2 implements Serializable {
   private Date date;
   private transient Map map;

// genau wie in InternationalDate1
...

   private void readObject(ObjectInputStream in)
      throws IOException, ClassNotFoundException {
      // Liest die serialisierten Attribute
      in.defaultReadObject();
      // Erstellt die Strings
      buildInternationalStrings();
   }
}
```

Listing 11.2: Internationales Datum mit verbesserter serialisierter Darstellung

Das Ergebnis kann sich sehen lassen: Statt 13.907 Byte verbraucht die serialisierte Darstellung von `InternationalDate2` nur 139 Byte – ein Hundertstel des ursprünglichen Werts. Und das lässt sich sogar noch verbessern. Anstelle des `Date`-Objekts können wir ja auch einfach nur den Zeitwert als `long` schreiben (Listing 11.3).

1 Diese Größe variiert von VM zu VM, da fast jede VM andere Locales unterstützt.

```
package com.tagtraum.perf.serialization;

import java.io.*;
import java.text.SimpleDateFormat;
import java.util.*;

public class InternationalDate3 implements Serializable {
   private transient Date date;
   private transient Map map;

// genau wie in InternationalDate1
...

   private void writeObject(ObjectOutputStream out)
         throws IOException {
      // Schreibt den Zeitwert als long
      out.writeLong(date.getTime());
   }

   private void readObject(ObjectInputStream in)
         throws IOException, ClassNotFoundException {
      // Liest den Zeitwert als long…
      date = new Date(in.readLong());
      // …und initialisiert die Strings
      buildInternationalStrings();
   }
}
```

Listing 11.3: Internationales Datum, das lediglich den Zeitwert als long serialisiert

Auch dies führt zu einer Verbesserung. Es ist zwar nicht mehr um den Faktor hundert, aber 82 Byte statt 139 Byte ist immerhin auch schon eine Verbesserung um 41 Prozent. Wir wollen sehen, ob sich das noch verbessern lässt, indem wir das `java.io.Externalizable` anstelle von `Serializable` verwenden.

```
package com.tagtraum.perf.serialization;

import java.io.*;
import java.text.SimpleDateFormat;
import java.util.*;

public class InternationalDate4 implements Externalizable {
   // transient ist nicht mehr nötig!
   private Date date;
   private Map map;

// genau wie in InternationalDate1
...

   public void writeExternal(ObjectOutput out)
         throws IOException {
```

```
        out.writeLong(date.getTime());
    }

    public void readExternal(ObjectInput in)
          throws IOException, ClassNotFoundException {
        date = new Date(in.readLong());
        buildInternationalStrings();
    }
}
```

Listing 11.4: Externalisierbares internationales Datum

`Externalizable` bedeutet, dass die gesamte Serialisierung vom Programmierer übernommen wird. Insbesondere werden nicht automatisch Attribute von Superklassen geschrieben. Da die Superklasse von `InternationalDate` die Klasse `Object` ist, macht sich dies jedoch in unserem Fall nicht im Ergebnis bemerkbar. Das Datum benötigt immer noch 82 Byte.

Natürlich ließe sich jedes Datum auch mit einem selbst geschriebenen Protokoll in 8 Byte ausdrücken. Dies soll hier jedoch nicht zu Debatte stehen.

Stattdessen wollen wir untersuchen, wie hoch der Preis für die Größenreduktion der serialisierten Form ist. Denn gratis waren unsere Optimierungen leider nicht. Tabelle 11.1 und Tabelle 11.2 zeigen die normalisierten Ausführungszeiten. Wie nicht anders zu erwarten, ist die benötigte Zeit zum Schreiben drastisch gesunken. Jedoch ist die Zeit fürs Lesen auch stark gestiegen. Das Erstellen der formatierten Daten ist halt nicht ganz umsonst. Insbesondere Sun JDK 1.3.1 Client hatte daran schwer zu schlucken.

Java VM	Version 1	Version 2	Version 3	Version 4	Version 5
Sun JDK 1.3.1 Client	100%	1,13%	0,60%	0,41%	0,45%
Sun JDK 1.3.1 Server	77%	1,81%	1,36%	1,58%	0,99%
Sun JDK 1.4.0 Client	78%	1,44%	1,21%	1,13%	0,68%
Sun JDK 1.4.0 Server	72%	2,94%	1,89%	1,51%	0,60%
IBM JDK 1.3.0	95%	0,75%	0,53%	0,30%	0,68%

Tabelle 11.1: Normalisierte Ausführungszeit fürs Schreiben der serialisierten Form

Java VM	Version 1	Version 2	Version 3	Version 4	Version 5
Sun JDK 1.3.1 Client	100%	7.284%	7.157%	14.293%	1,97%
Sun JDK 1.3.1 Server	88%	195%	170%	331%	3,56%
Sun JDK 1.4.0 Client	89%	223%	214%	423%	2,36%
Sun JDK 1.4.0 Server	85%	196%	169%	334%	2,84%
IBM JDK 1.3.0	128%	253%	224%	440%	1,64%

Tabelle 11.2: Normalisierte Ausführungszeit fürs Lesen der serialisierten Form

In beiden Tabellen sehen Sie jedoch Werte einer *Version 5* von `InternationalDate`, die mit geradezu unglaublich guten Werten aufwarten kann. Zugegeben, hier habe ich etwas geschummelt, da diese Version nur unter günstigsten Umständen auf die angegebenen Werte kommt. Nichtsdestoweniger ist das natürlich besser, als diese Werte nie zu erreichen.

Die Idee für `InternationalDate5` ist folgende: `InternationalDate` ist unveränderbar. Das bedeutet, dass wir einen Pool von Objekten anlegen können, die immer wieder verwendet werden. Somit ersparen wir uns das ständige Neuerstellen gleicher Objekte. Dies macht insbesondere Sinn für `InternationalDate`, da jede Instanz sehr aufwändig zu erzeugen ist. Um sicherzustellen, dass nicht mehrere Instanzen eines Datums existieren, deklarieren wir den Konstruktor als `private` und fügen eine Fabrikmethode hinzu. Dies entspricht dem Singleton-Muster [Gamma96 S.139]. Als Pool verwenden wir einen Cache aus *Kapitel 8.4 Caches*. Und dank der `readResolve()`-Methode können wir während des Deserialisierens die `getInstance()`-Methode benutzen, um indirekt auf den Cache zuzugreifen. Somit kontrollieren und limitieren wir die Anzahl an `InternationalDate`-Objekten und sorgen so dafür, dass nicht zu viel Speicher verschwendet wird. Der Cache wird an die Klasse gebunden und ist somit pro Klassenobjekt der `InternationalDate`-Klasse[2] eindeutig.

Mit der `Externalizable`-Schnittstelle können wir diese Version übrigens nicht verwirklichen, da diese einen öffentlichen, argumentlosen Konstruktor voraussetzt. Aus diesem Grund setzen wir die Lösung mit `Serializable` um und serialisieren den Zeitwert.

Die Zeiten in Tabelle 11.1 und Tabelle 11.2 für Version 5 sind Testzeiten, die mit demselben Klassenobjekt gemessen wurden. Im Test lag die Cache-Trefferrate also bei 100 Prozent. Dies ist der Grund dafür, dass die Lese-Zeiten so kurz sind – das Objekt musste nicht extra instanziiert werden, da es sich bereits im Cache befand. Dies ist natürlich nicht immer der Fall. Die Lösung macht also nicht für alle Anwendungen Sinn. Diejenigen, für die sie geeignet ist, profitieren jedoch stark.

```
package com.tagtraum.perf.serialization;

import com.tagtraum.perf.datastructures.Cache;
import com.tagtraum.perf.datastructures.RandomCache;

import java.io.*;
import java.text.SimpleDateFormat;
import java.util.*;
```

2 Die Identität einer Klasse besteht aus dem Paar Klassenobjekt und `ClassLoader`-Objekt. Das heißt, wenn eine Klasse von zwei verschiedenen `ClassLoader` geladen wird, sind die resultierenden `Class`-Objekte nicht identisch. Somit ist der verwendete Cache nicht pro VM, sondern pro `Class`-Objekt eindeutig.

```
public class InternationalDate5 implements Serializable {
   private transient Date date;
   private transient Map map;
   // Cache für InternationalDate5-Objekte
   private static Cache cache = new RandomCache(512);

   // Privater Konstruktor
   private InternationalDate5(Date date) {
      this.date = date;
      buildInternationalStrings();
   }

   // Fabrikmethode
   public static synchronized InternationalDate5
         getInstance(Date date) {
      InternationalDate5 iDate
         = (InternationalDate5) cache.get(date);
      if (iDate == null) {
         iDate = new InternationalDate5(date);
         cache.put(date, iDate);
      }
      return iDate;
   }

   public static InternationalDate5 getInstance() {
      return getInstance(new Date());
   }

// genau wie in InternationalDate1
...

   // Auflösen des date-Attributes zu einer Instanz
   private Object readResolve()
         throws ObjectStreamException {
      return getInstance(date);
   }

   private void writeObject(ObjectOutputStream out)
         throws IOException {
      out.writeLong(date.getTime());
   }

   private void readObject(ObjectInputStream in)
         throws IOException, ClassNotFoundException {
      date = new Date(in.readLong());
   }
}
```

Listing 11.5: Internationales Datum mit Cache

11.1.2 Optimierte logische Darstellung

Im obigen Beispiel haben Sie gesehen, wie sich die Datenmenge effizient verkleinern lässt. Wir wollen uns ein weiteres Beispiel anschauen. Listing 11.6 zeigt eine einfachverknüpfte Liste für Strings, die `Serializable` implementiert.

```
package com.tagtraum.perf.serialization;

import java.io.Serializable;

public class LinkedList1 implements Serializable {

   private Entry head;
   private int size;

   public LinkedList1() {
      head = new Entry();
   }

   public void add(String value) {
      Entry e = head;
      while (e.getNext() != null) {
         e = e.getNext();
      }
      Entry newEntry = new Entry();
      newEntry.setValue(value);
      e.setNext(newEntry);
      size++;
   }

   public String get(int index) {
      if (index >= size) throw new IndexOutOfBoundsException();
      Entry e = head.getNext();
      for (int i = 0; i < index; i++) {
         e = e.getNext();
      }
      return e.getValue();
   }

   public String remove(int index) {
      if (index >= size) throw new IndexOutOfBoundsException();
      Entry e = head.getNext();
      for (int i = 0; i < index; i++) {
         e = e.getNext();
      }
      e.getPrev().setNext(e.getNext());
      if (e.getNext() != null)
         e.getNext().setPrev(e.getPrev());
      return e.getValue();
   }
```

```
    private static class Entry implements Serializable {
        private Entry next;
        private Entry prev;
        private String value;

        public String getValue() {
            return value;
        }

        public void setValue(String value) {
            this.value = value;
        }

        public Entry getNext() {
            return next;
        }

        public void setNext(Entry next) {
            this.next = next;
        }

        public Entry getPrev() {
            return prev;
        }

        public void setPrev(Entry prev) {
            this.prev = prev;
        }
    }
}
```

Listing 11.6: Simple Implementierung einer einfach verlinkten Liste

Naiv betrachtet, ist an dieser Implementierung wenig auszusetzen. Wenn Sie diese Liste jedoch benutzen, werden Sie evtl. feststellen, dass das Serialisieren und Deserialisieren ab einer bestimmten Länge zu einem `StackOverflowError` führt. Der Grund hierfür ist die simple Tatsache, dass der automatische Serialisierungsmechanismus den Objektbaum rekursiv traversiert. Die Länge unserer Liste steht somit in direkter Beziehung zur Rekursionstiefe – und die ist bei den meisten Systemen begrenzt. Auf einem Windows-2000-System mit Sun JDK 1.4.0 lag die maximale Länge der Liste bei 730.

Statt sich also auf den automatischen Mechanismus zu verlassen, müssen wir uns ein wenig anstrengen. Die offensichtliche Lösung ist es, zunächst die Länge der Liste zu schreiben und dann ebenso viele Werte einzulesen. Entsprechend fügen wir `readObject()`- und `writeObject()`-Methoden hinzu und deklarieren `head` und `size` als `transient` (Listing 11.7).

```
package com.tagtraum.perf.serialization;

import java.io.*;

public class LinkedList2 implements Serializable {

   private transient Entry head;
   private transient int size;

   public LinkedList2() {
      head = new Entry();
   }

// genau wie LinkedList1
...

   private void writeObject(ObjectOutputStream out)
         throws IOException {
      out.writeInt(size);
      Entry e = head;
      while (e.getNext() != null) {
         e = e.getNext();
         out.writeObject(e.getValue());
      }
   }

   private void readObject(ObjectInputStream in)
         throws IOException, ClassNotFoundException {
      size = in.readInt();
      head = new Entry();
      Entry e = head;
      for (int i = 0; i < size; i++) {
         Entry newEntry = new Entry();
         newEntry.setValue((String) in.readObject());
         e.setNext(newEntry);
         e = newEntry;
      }
   }

   private static class Entry {

   // genau wie LinkedList1
   ...

   }
}
```

Listing 11.7: Bessere Serialisierung einer verknüpften Liste

Nicht nur, dass wir jetzt beliebig lange Listen serialisieren können, die Größe der serialisierten Form sinkt auch leicht. Genauer gesagt, sie sinkt von 14.848 Byte auf 9.761 Byte bei identischem Inhalt (700 Strings der Länge 9-11). Dies lässt sich leicht noch ein wenig verbessern, indem wir etwas sorgfältiger serialisieren. Anstatt nämlich Strings mit `writeUTF()` und `readUTF()` zu schreiben und zu lesen, haben wir uns die Freiheit genommen, die Methoden `readObject()` und `writeObject()` zu benutzen. Diese Nachlässigkeit rächt sich in der resultierenden Größe. Mit `writeUTF()` und `readUTF()` (Listing 11.8) lässt sich diese immerhin von 9.761 Byte auf 9.104 Byte verringern.

Es lohnt sich, genau passende Methoden aus `DataInput` bzw. `DataOutput` zu benutzen.

```
package com.tagtraum.perf.serialization;

import java.io.*;

public class LinkedList3 implements Serializable {

   private transient Entry head;
   private transient int size;

// genau wie LinkedList1
...

   private void writeObject(ObjectOutputStream out)
         throws IOException {
      out.writeInt(size);
      Entry e = head;
      while (e.getNext() != null) {
         e = e.getNext();
         out.writeUTF(e.getValue());
      }
   }

   private void readObject(ObjectInputStream in) throws IOException,
ClassNotFoundException {
      size = in.readInt();
      head = new Entry();
      Entry e = head;
      for (int i = 0; i < size; i++) {
         Entry newEntry = new Entry();
         newEntry.setValue(in.readUTF());
         e.setNext(newEntry);
         e = newEntry;
      }
   }

   private static class Entry {
```

```
    // genau wie LinkedList1
    ...

    }
}
```

Listing 11.8: LinkedList mit readUTF() und writeUTF()

11.2 Latenzzeiten und Overhead

Für jeden entfernten Methodenaufruf gibt es fixe Kosten, die insbesondere mit dem Netzwerk zu tun haben. Jeder Verbindungsaufbau dauert halt ein wenig. Die simple Regel, die sich daraus für RMI ableiten lässt, lautet:

Wenn Sie die Wahl haben, eine entfernte Methode oft mit wenigen Argumenten oder selten mit vielen Argumenten bzw. großen Objekten aufzurufen, rufen Sie sie selten auf.

Die Erklärung dafür ist sehr einfach. Jeder Methodenaufruf habe fixe Kosten c und jedes Objekt in einem dieser Aufrufe habe Kosten von o. Die Gesamtkosten für n Aufrufe mit m Objekten errechnen sich aus $nc + mo$. Es ist klar, dass die Kosten bei konstantem m und steigendem n steigen.

Wir wollen dies durch ein kleines Experiment untermauern. Eine entfernte Methode habe folgende Signatur:

```
public void send(String[] s) throws RemoteException;
```

In unserem Test rufen wir die Methode auf, um 1.000 Strings zu übertragen, und messen die Zeit. Dabei variieren wir die Anzahl der Strings, die pro Aufruf übertragen werden, während die Gesamtanzahl der übertragenen Strings gleich bleibt.

Methodenaufrufe	1.000	100	10	1
Zeit	100%	14,8%	4,5%	3,0%

Tabelle 11.3: Normalisierte Ausführungszeit für das Übertragen von 1.000 Strings

Das Ergebnis in Tabelle 11.3 zeigt deutlich, dass die Übertragung umso schneller vonstatten geht, je weniger Methodenaufrufe wir verwenden.

Eine direkte Anwendung dieses Wissen ist das *Value-Object*-Muster, das besonders im J2EE/EJB-Umfeld beliebt ist. Statt über die einzelnen `get()`-Methoden eines entfernten Objektes einzelne Werte abzufragen, fragt man nach einem einzigen Objekt, das alle Attribute eines Objekts beinhaltet.

11.3 Verteilte Speicherbereinigung

In verteilten Systemen reicht die lokale Speicherverwaltung nicht aus. Daher verfügt RMI über eine verteilte Speicherverwaltung, die sicherstellt, dass Objekte, die von anderen Java VM referenziert werden, nicht vorzeitig beseitigt werden. Sie stellt zudem sicher, dass entfernte Objekte überhaupt beseitigt werden.

Wenn Sie also ein entferntes Objekt benutzen, wird dem Objekt-Server mitgeteilt, dass Sie dieses Objekt referenzieren. Es wird zudem automatisch periodisch signalisiert, dass Sie das Objekt noch für eine Weile länger benutzen wollen. Wenn Sie das entfernte Objekt nicht mehr benutzen, wird dies ebenso signalisiert.

Natürlich ist das Signalisieren nicht gratis, denn schließlich werden Nachrichten über das Netzwerk übermittelt. *Dass* diese Nachrichten übermittelt werden, ist Teil des Systems und steht außer Frage. Interessant ist jedoch, wie *häufig* diese Nachrichten übermittelt werden. Und genau dieser Parameter lässt sich konfigurieren.

Sie können beim Start der VM, die Ihr entferntes Objekt ausführt, den Parameter `java.rmi.dgc.leaseValue` setzen. Dieser Parameter spezifiziert die Zeit in Millisekunden, die ein entferntes Objekt seinen Klienten garantiert, dass es noch existiert. Gewöhnlich wird von Klienten nach Ablauf der Hälfte dieser Zeit eine neue Garantie angefordert. Der voreingestellte Wert liegt bei zehn Minuten. Wenn Sie diesen Wert erhöhen, verringern Sie also die Netzbelastung. Die Kehrseite ist jedoch, dass entfernte Objekte länger als nötig im Speicher verbleiben. Eventuell kann es sich also auch lohnen, genau umgekehrt zu verfahren und den Wert zu verringern.

Beispiel:

```
java -Djava.rmi.dgc.leaseValue=120000000 <mainclass>
```

Letztendlich müssen sowohl Referenzen auf entfernte Objekte als auch die entfernten Objekte irgendwann von der Speicherbereinigung beseitigt werden. Um sicherzustellen, dass dies auch in endlicher Zeit passiert, ruft die Sun-Implementierung von RMI die Speicherbereinigung periodisch mittels `System.gc()` auf. Dabei handelt es sich um eine vollständige Speicherbereinigung. Die Dauer dieser Periode lässt sich mit den Parametern `sun.rmi.dgc.client.gcInterval` bzw. `sun.rmi.dgc.server.gcInterval` beim Start der VM setzen. Der Server-Parameter sollte für VMs gesetzt werden, die entfernte Objekte ausführen, der Client-Parameter für VMs, die entfernte Objekte benutzen. Die voreingestellten Werte für beide Parameter sind 60.000 Millisekunden, also eine Minute. Diesen Wert zu erhöhen macht insbesondere Sinn bei Systemen, die über einen großen Heap verfügen, da eine vollständige Speicherbereinigung sehr viel Zeit kosten kann. Dies ist vergeudete Zeit, wenn der Heap noch nicht voll ist. Es kann sich also lohnen, die Speicherbereinigungsaktivität zu verfolgen und die Werte der beiden Parameter entsprechend zu verändern.

12 XML

Seit JDK 1.4.0 ist Unterstützung für *XML* (*Extensible Markup Language*) Teil der Java-Entwicklungsumgebung. Bereits vorher war XML-Unterstützung als optionales Paket mit Namen *JAXP* (*Java API for XML Processing*) von Sun erhältlich. JAXP ist im Wesentlichen eine Schnittstelle zu XML-Parsern sowie eine Schnittstelle zu *XSLT* (*Extensible Stylesheet Language Transformations*). In diesem Kapitel werden wir uns mit Performance-Aspekten von Parsern sowie einigen anderen XML-bezogenen Problemen auseinander setzen.

12.1 SAX, DOM & Co

JAXP unterstützt zwei verschiedene Parser-Modelle, deren Performance in Hinblick auf Speicherverbrauch und Geschwindigkeit sehr unterschiedlich sein kann: *SAX* (*Simple API for XML*) und *DOM* (*Document Object Model*). Zudem existiert noch eine dritte, nennenswerte Parsergattung namens Pull-Parser, die jedoch nicht von JAXP unterstützt wird. Wir werden kurz alle drei Gattungen erläutern.

12.1.1 SAX

SAX ist eine Schnittstelle, die von Mitgliedern der XML-DEV Mailingliste entwickelt wurde. Sie ist einfach, leichtgewichtig und gehört zur Gattung der Push-Parser. Um SAX zu benutzen, müssen Sie eine `org.xml.sax.ContentHandler`-Klasse implementieren, eine Instanz dieser Klasse bei einem `XMLReader` registrieren und anschließend dessen `parse()`-Methode aufrufen. Der Parser ruft dann für jedes Element die entsprechende Methode ihres `ContentHandlers` auf (Abbildung 12.1). Und dies ist auch genau die wesentliche Eigenschaft von SAX: Der Parser ruft die Methoden eines Handlers auf. Das heißt der Parser hat die Kontrolle und der Handler reagiert. Solche Parser werden auch als Push-Parser oder ereignisorientierte Parser bezeichnet.

Da der SAX-Parser für alle XML-Elemente der Reihe nach Methoden eines Handlers aufruft, eignet sich SAX hervorragend für einen stromorientierten, sequenziellen Zugriff auf XML-Dokumente. Es ist jedoch ungeeignet für wahlfreien Zugriff.

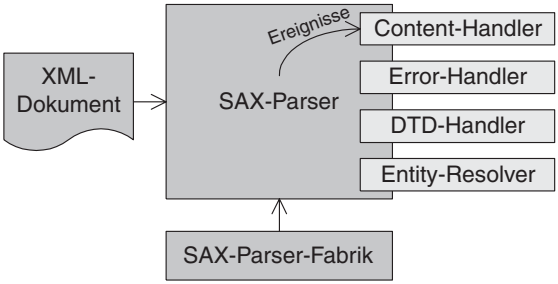

Abbildung 12.1: Verarbeitungskonzept von SAX

Um die Benutzung von SAX ein wenig zu vereinfachen, existiert eine Klasse `org.xml.sax.helpers.DefaultHandler`, die bereits die Methoden von `ContentHandler` sowie einigen anderen Schnittstellen implementiert. Falls Sie SAX benutzen wollen, bietet es sich also an, von `DefaultHandler` zu erben und die benötigten Methoden zu überschreiben. Zudem wird die `XMLReader`-Klasse von JAXP durch eine `SAXParser`-Klasse gekapselt, die wiederum von einer `SAXParserFactory` erzeugt wird. Somit kommen Sie mit dem `XMLReader`-Interface kaum in Berührung.

```
package com.tagtraum.perf.xml;

import org.xml.sax.Attributes;
import org.xml.sax.SAXException;
import org.xml.sax.helpers.DefaultHandler;
import javax.xml.parsers.SAXParser;
import javax.xml.parsers.SAXParserFactory;
import java.io.File;

public class SimpleSAXDemo {

   // Gibt die Namen aller Elemente einer XML-Datei aus.
   public static void main(String[] args) throws Exception {
      SAXParserFactory parserFactory
         = SAXParserFactory.newInstance();
      SAXParser parser = parserFactory.newSAXParser();
      parser.parse(new File(args[0]), new DefaultHandler() {

         public void startElement(String namespaceURI,
               String localName, String qName, Attributes atts)
                  throws SAXException {
            System.out.println(qName);
         }

      });
   }
}
```

Listing 12.1: SAX-basierter Parser, der die Namen aller Tags einer XML-Datei ausgibt

Listing 12.1 zeigt ein einfaches Programm, das die Namen aller Elemente einer XML-Datei ausgibt. Um dies zu erreichen haben wir einfach die Methode `startElement()` eines `DefaultHandlers` überschrieben und diesen als Argument an die `parse()`-Methode eines `SAXParser`-Objektes übergeben.

12.1.2 DOM

DOM ist ein Objekt-Modell des *World Wide Web Konsortiums* (*W3C – http://www.w3c.org/*). Es dient zur hierarchischen Darstellung von Dokumenten in einer Baumstruktur. Zum Erstellen dieser Darstellung wird meist das gesamte Dokument in einem Rutsch analysiert. Dabei werden für alle Knoten des Baums entsprechende Objekte instanziiert. Anschließend wird die Wurzel des Baums in Form eines `org.w3c.dom.Document`-Objektes an den Klienten zurückgegeben (Abbildung 12.2). Das bedeutet, dass der Benutzer nach dem Aufruf der `parse()`-Methode den vollständig geparsten Baum zurückbekommt.

DOM-Repräsentationen eines XML-Dokuments halten meist den gesamten Dokument-Baum im Speicher. Sie ermöglichen so schnellen wahlfreien Zugriff auf einzelne Dokument-Elemente sowie leichtes Traversieren des Baumes.

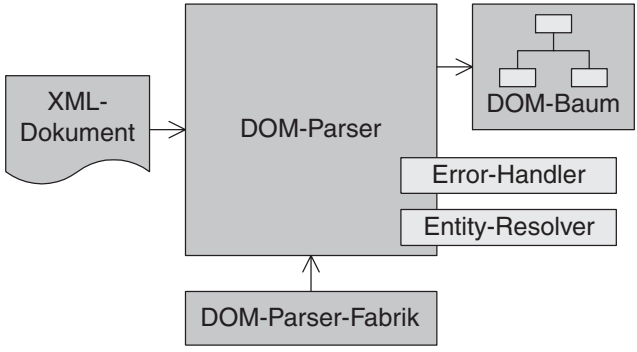

Abbildung 12.2: Verarbeitungskonzept von DOM

Dadurch, dass die Repräsentation im Speicher liegt, haben DOM-Implementierungen einen enormen Speicherverbrauch. Dies ist auch ihr größter Nachteil.

Zum Vergleich: Eine XML-Fassung von Goethes Faust II hat 9.755 Elemente und eine Größe von 551 KByte.[1] Wenn man sie mit dem in Sun JDK 1.4.0 enthaltenen Crimson-SAX-Parser durchliest, steigt der Heapspeicherverbrauch nicht über 1 Mbyte. Liest man dieselbe Datei dagegen mit dem Crimson-DOM-Parser, steigt der Heapspeicherverbrauch auf rund 5 Mbyte – beinahe das Zehnfache der Dokumentgröße.

1 Die benutzte Fassung basiert auf der Version von *http://www.kalliope.org/*.

Einige Implementierungen wie *Apache Xerces-J* (*http://xml.apache.org/*) versuchen dieses Problem zu lindern, indem sie Teile des Baumes erst beim Benutzen vollständig instanziieren (Deferred Node Expansion). Dadurch wird das `Document`-Objekt schneller vom Parser zurückgegeben und der Speicherverbrauch leicht gesenkt – sofern nicht jedes Element des Dokuments benutzt wird.

Listing 12.2 zeigt ein sehr einfaches Programm, das genau wie das SAX-Beispiel in Listing 12.1 ein XML-Dokument liest und jeweils die Namen der XML-Tags ausgibt. Dabei wird JAXP als Schnittstelle zu einem DOM-Parser benutzt.

```
package com.tagtraum.perf.xml;

import org.w3c.dom.Document;
import org.w3c.dom.NodeList;
import javax.xml.parsers.DocumentBuilder;
import javax.xml.parsers.DocumentBuilderFactory;
import java.io.File;

public class SimpleDOMDemo {

   public static void main(String[] args) throws Exception {
      DocumentBuilderFactory builderFactory
         = DocumentBuilderFactory.newInstance();
      DocumentBuilder documentBuilder
         = builderFactory.newDocumentBuilder();
      Document document = documentBuilder.parse(new File(args[0]));
      // Besorge Liste mit allen Elementen.
      // Achtung! getElementeByTagName() ist in einigen Parsern
      // sehr langsam!
      NodeList list = document.getElementsByTagName("*");
      for (int i = 0; i < list.getLength(); i++) {
         System.out.println(list.item(i).getNodeName());
      }
   }
}
```

Listing 12.2: DOM-basiertes Programm, das die Namen aller Elemente einer XML-Datei ausgibt

12.1.3 Pull-Parser

Pull-Parser sind Parser, die nur aktiv sind, wenn der Benutzer dies wünscht. Das heißt es wird nicht immer der komplette Dokument-Baum im Speicher aufgebaut oder ein Handler über alles in Kenntnis gesetzt, sondern immer nur das gelesen, was der Klient wünscht. Es lassen sich sogar ganze Baum-Knoten überspringen.

Somit liegt die Kontrolle nicht beim Parser, sondern beim Klienten, was zu einer effizienteren Benutzung des Parsers führen kann. Ein Beispiel für einen Pull-Parser ist *XPP* (*XML-*

Pull-Parser) des *Extreme! Labs* der Universität von Indiana (*http://www.extreme.indiana.edu/xgws/xsoap/xpp/*). XPP unterstützt keine Validierung, Entitäten, Kommentare oder Verarbeitungsanweisungen (Processing Instructions) und ist somit nicht universell einsetzbar. Er kann jedoch Dokument-Teile parsen und ist für J2ME geeignet.

XPP bietet dem Benutzer sowohl eine Baum- als auch eine Ereignissicht (Abbildung 12.3). Listing 12.3 und Listing 12.4 zeigen Beispielprogramme für beide Sichten.

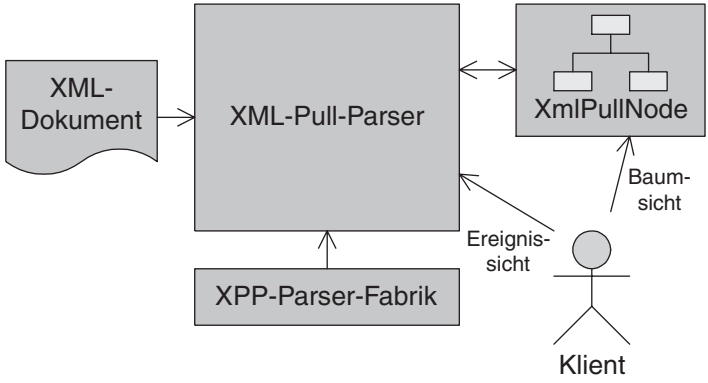

Abbildung 12.3: Verarbeitungskonzept von XPP

Für Faust II liegt der Heapspeicherverbrauch der Ereignissicht relativ konstant bei 1½ Mbyte. Bei der Baumsicht steigt der Speicherverbrauch während des Parsens auf knapp 5 Mbyte an. Würden wir Teile des Baumes beim Parsen überspringen oder bewusst nur einen Teil des Baumes parsen, wäre der Speicherverbrauch entsprechend geringer. Grundsätzlich können Sie während des Parsens leicht zwischen Baum- und Ereignissicht wechseln. Diese Wahl haben Sie bei anderen Parsern in der Regel nicht.

```
package com.tagtraum.perf.xml;

import org.gjt.xpp.XmlPullParser;
import org.gjt.xpp.XmlPullParserFactory;
import java.io.FileReader;

public class SimpleXPPEventDemo {
   public static void main(String[] args) throws Exception {
      XmlPullParserFactory parserFactory
         = XmlPullParserFactory.newInstance();
      XmlPullParser parser = parserFactory.newPullParser();
      parser.setInput(new FileReader(args[0]));
      // Lesen, bis das Ende des Dokumente erreicht ist.
```

```
        while (parser.next() != XmlPullParser.END_DOCUMENT) {
            if (parser.getEventType() == XmlPullParser.START_TAG) {
                // Gib alle Namen von Start-Tags aus.
                System.out.println(parser.getRawName());
            }
        }
    }
}
```

Listing 12.3: XPP-basiertes Programm, das die Namen aller Elemente einer XML-Datei ausgibt und dazu sequenziellen Zugriff benutzt

```
package com.tagtraum.perf.xml;

import org.gjt.xpp.XmlPullNode;
import org.gjt.xpp.XmlPullParser;
import org.gjt.xpp.XmlPullParserFactory;
import java.io.FileReader;

public class SimpleXPPTreeDemo {
    public static void main(String[] args) throws Exception {
        XmlPullParserFactory parserFactory
            = XmlPullParserFactory.newInstance();
        XmlPullParser parser = parserFactory.newPullParser();
        parser.setInput(new FileReader(args[0]));
        // Finde erstes Element.
        while (parser.next() != XmlPullParser.START_TAG) {}
        XmlPullNode node = parserFactory.newPullNode(parser);
        printElementName(node);
    }

    // Tarversiert rekursiv durch den Baum und druckt alle
    // Start-Element-Namen.
    private static void printElementName(XmlPullNode node)
            throws Exception {
        System.out.println(node.getRawName());
        Object object;
        while ((object = node.readNextChild()) != null) {
            if (object instanceof XmlPullNode) {
                printElementName((XmlPullNode) object);
            }
        }
    }
}
```

Listing 12.4: XPP-basiertes Programm, das die Namen aller Elemente einer XML-Datei ausgibt und dazu das Baummodell benutzt

12.2 Kleiner Modellvergleich

Es ist sehr schwierig, die verschiedenen Ansätze allgemeingültig und dennoch aussagekräftig quantitativ zu vergleichen. Insbesondere spielt die Beschaffenheit der zu verarbeitenden XML-Dokumente und die Art der Anwendung eine große Rolle.

Wir wollen dennoch einen kleinen Vergleich wagen, um uns eine grobe Vorstellung zu verschaffen. Zu diesem Zweck werden die oben abgedruckten Demo-Programme mit Faust II gefüttert und die Verarbeitungszeit gemessen. Um nicht die Ausgabegeschwindigkeit der Konsole zu messen, werden die Elementnamen jedoch nicht ausgegeben. Zudem stellt sich heraus, dass im Crimson-Parser des Sun JDK 1.4.0 die Methode `document.getElementsByTagName()` mangelhaft implementiert ist. Daher benutzen wir für den DOM-Test folgenden funktional gleichwertigen und in unserem Test etwa 600-mal schnelleren Code:

```
package com.tagtraum.perf.xml;

import org.w3c.dom.*;
import javax.xml.parsers.*;
import java.io.File;

public class DOMBenchDemo2 {
   public static void main(String[] args) throws Exception {
      DocumentBuilderFactory builderFactory
         = DocumentBuilderFactory.newInstance();
      builderFactory.setValidating(false);
      builderFactory.setNamespaceAware(false);
      DocumentBuilder documentBuilder
         = builderFactory.newDocumentBuilder();
      File file = new File(args[0]);
      parse(documentBuilder, file);
   }

   private static void parse(DocumentBuilder documentBuilder,
         File file) throws Exception {
      long start = System.currentTimeMillis();
      Document document = documentBuilder.parse(file);
      traverse(document.getDocumentElement());
      System.out.println(System.currentTimeMillis() - start);
   }
   private static void traverse(Element element) throws Exception {
      // Greife auf den Namen zu, ohne ihn jedoch auszugeben.
      element.getTagName();
      NodeList list = element.getChildNodes();
      for (int i=0, length = list.getLength(); i<length; i++) {
```

```
            if (list.item(i).getNodeType() == Node.ELEMENT_NODE) {
               traverse((Element)list.item(i));
            }
         }
      }
   }
```

Listing 12.5: Traversieren des DOM mittels rekursiver Aufrufe der Methode getChildNodes() kann um ein Vielfaches schneller sein als mit getElementsByTagName()

Abbildung 12.4 zeigt das Ergebnis unseres Tests. SAX ist der klare Sieger, gefolgt von XPP mit Ereignissicht. DOM und die XPP-Baumsicht sind in etwa gleichauf. Benutzt wurden jeweils die Standard-SAX- und DOM-Implementierungen aus Sun JDK 1.4.0 (Crimson) sowie XPP 2.1.7. Beim Testen fiel auf, dass die Server-VM in den ersten Parse-Durchgängen die meiste Zeit mit Kompilieren und Speicherbereinigung verbrachte. Erst in späteren Durchgängen wurden bessere Zeiten als mit der Client-VM erreicht. Wenn Sie realistische Tests durchführen wollen, parsen Sie also auf jeden Fall mehr als einmal.

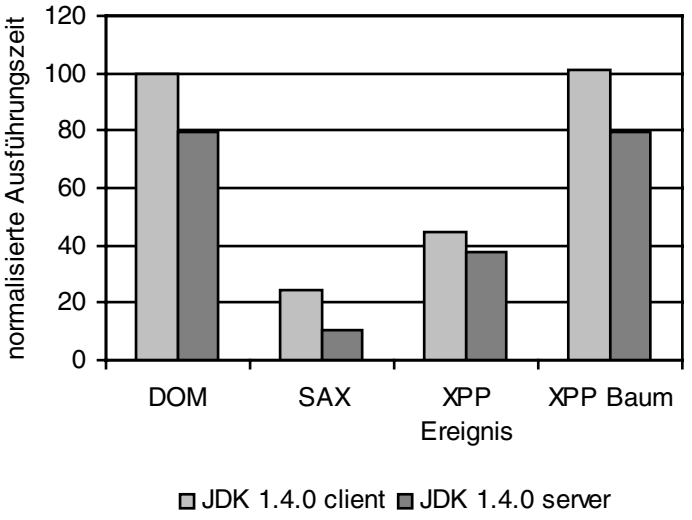

Abbildung 12.4: Dauer des Parsens und Besuchens jedes Elements in Faust II ohne Validierung oder Namensräume in Abhängigkeit von VM und Parse-Modell

Beim ersten Test waren weder Validierung noch Unterstützung für Namensräume eingeschaltet. Beide wirken sich jedoch auf die Verarbeitungsgeschwindigkeit aus. Daher wollen wir den Test nochmals mit diesen Optionen durchführen.

Abbildung 12.5 zeigt, dass sowohl Validierung als auch die Unterstützung von Namensräumen das Parsen verlangsamen. Insbesondere auf SAX wirkt sich die Kombination der beiden Optionen relativ gesehen sehr negativ aus.

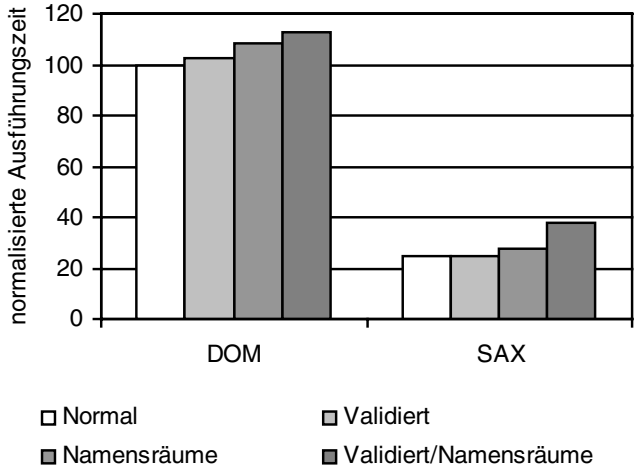

Abbildung 12.5: Dauer des Parsens und Besuchens jedes Elements in Faust II in Abhängigkeit von Parse-Modell sowie Validierung und Namensraumunterstützung

Als Ergebnis unseres kleinen Vergleichs lässt sich festhalten, dass zum Ausgeben aller Elementnamen in unserem Faust-II-Dokument von den getesteten Varianten der verwendete SAX-Parser ohne Validierung und Namensraumunterstützung am besten geeignet ist. Das ist jedoch auch schon alles. Lassen Sie sich nicht von Tests anderer oder gar der Anbieter irreführen. Letztlich zählen nur Ihre Anforderungen und Ihre Umgebung – und dazu gehört auch die VM. Xerces-J wird zu einem großen Teil von IBM-Mitarbeitern entwickelt, Crimson von Sun. Sie können sich vorstellen, für welche VM die beiden Parser jeweils optimiert wurden.

12.3 Den richtigen Parser wählen

Um den richtigen Parser zu finden, müssen Sie im Grunde zunächst eine Entscheidung für eine Parsergattung fällen. Die wichtigsten Fragen sind hierbei, wie Sie auf Dokumente zugreifen wollen und wie diese Dokumente beschaffen sind. Beispielsweise spielt der Speicherverbrauch für kleine Dokumente nicht so eine große Rolle wie für große Dokumente. Genauso ist wahlfreier Zugriff sehr aufwändig und für viele Anwendungen unnötig. Vielleicht benötigen Sie auch nur einen Parser, der lediglich eine Untermenge vom XML beherrscht, dafür aber sehr klein und schnell ist sowie auch unter J2ME läuft. Es ist extrem wichtig, dass Sie sich darüber klar werden, welche Dokumente Sie unter welchen Rahmenbedingungen auf welche Art verarbeiten wollen.

Erst wenn Sie sich über Ihre Bedürfnisse im Klaren sind, sollten Sie eine Parsergattung wählen. Nach der Entscheidung für die Gattung steht die Entscheidung für eine Implementierung an. Falls Sie sich für DOM oder SAX entscheiden, können Sie sich glücklich

schätzen. JAXP ermöglicht es Ihnen, die Implementierung auszutauschen ohne eine Zeile Code zu verändern. Die XPP-Implementierung lässt sich dank Abstrahierung durch Schnittstellen ähnlich einfach austauschen. Neben XPP existiert noch eine zweite Implementierung speziell für die J2ME-Umgebung namens *kXML2* (*http://www.kxml.org/*).

Ich kann Ihnen leider keine Empfehlung für eine Implementierung geben. Was bleibt ist der gut gemeinte Rat, dass nur Testen hilft – und zwar mit Dokumenten, die Sie später auch verwenden werden. Einige Parser scheinen geeigneter für große als für kleine Dokumente zu sein. Andere haben eine sehr schnelle SAX-Unterstützung oder eine sehr sparsame DOM-Unterstützung.

Welche Implementierung Sie wählen sollten, hängt von folgenden Faktoren ab:

- Parsergattung (DOM, SAX, XPP, ...)
- Unterstützte Schnittstellen (beispielsweise DOM Level 3, SAX 2, ...)
- Dokumentgröße
- Komplexität und Größe der DTD bzw. des XML-Schemas
- Zeit

Der Faktor Zeit ist daher so wichtig, weil wir uns noch immer in einem XML-Hype befinden. Viele Firmen und Organisationen stehen in einem harten Wettbewerb zueinander, so dass in einem halben Jahr einiges passieren kann. Wenn Sie sich also früh auf die schnelle, aber proprietäre Parser-Technologie eines Nischenanbieters festlegen, können Sie leicht den Anschluss an den Mainstream verlieren. Und der Mainstream ist eventuell in einem halben Jahr viel schneller als der Nischenanbieter, der inzwischen Konkurs angemeldet hat.

12.4 XML ausgeben

Nicht nur das Lesen, auch das Schreiben von XML ist eine zeitkritische Angelegenheit. Grundsätzlich gilt:

Der schnellste Weg XML zu produzieren, ist einfach in einen gepufferten Strom zu schreiben. Dies ist jedoch auch der fehleranfälligste.

Einer der sichersten Wege, XML zu produzieren, ist es, einen DOM-Baum aufzubauen und diesen anschließend in einen gepufferten Strom zu schreiben. Dies ist deshalb wenig fehleranfällig, weil Sie nicht vergessen können, ein Tag zu schließen. Der Preis dafür ist ein hoher Speicherverbrauch, da Sie den gesamten Baum im Speicher halten müssen, bevor Sie ihn ausgeben.

Falls Sie sich für diese Form der Ausgabe entscheiden sollten, ziehen Sie in Betracht zur Ausgabe des DOM das Xerces-J-Paket `org.apache.xml.serialize` zu benutzen. Es ist zwar Teil von Xerces-J, jedoch unabhängig von der Parser-Implementierung. Eine Alternative zu diesem Paket sind die Lade- und Speicher-Features von DOM Level 3 (*http://www.w3.org/TR/DOM-Level-3-ASLS/*). Da DOM nur durch Schnittstellen spezifiziert ist, kann es sein, dass die entsprechenden Klassen einer DOM-Implementierung besonders aufeinander abgestimmt und daher optimiert sind. Daher macht es durchaus Sinn, diese Features zu verwenden, sobald DOM Level 3 dem Entwurfsstadium entwachsen ist und allgemein unterstützt wird. Sun JDK 1.4.0 bietet zurzeit nur DOM Level 2. Jedoch verfügt Xerces-J 2.0.1 bereits über eine rudimentäre DOM-Level-3-Unterstützung.

Listing 12.6 zeigt beispielhaft, wie Sie ein Dokument mit einem `org.w3c.dom.ls.DOMWriter` in die Standardausgabe drucken können.

```
Document document = ...
DOMImplementation domImpl = DOMImplementationRegistry
   .getDOMImplementation("Core 2.0 LS-Save 3.0");
if (domImpl != null) {
   DOMImplementationLS implls = (DOMImplementationLS) domImpl;
   DOMWriter writer = implls.createDOMWriter();
   writer.writeNode(System.out, document);
}
else {
   System.out.println("Konnte keinen DOM-Level-3-Parser finden,"
      + " der Speichern unterstützt.");
}
```

Listing 12.6: Beispiel-Code für die Ausgabe eines DOM-Baums mit DOM Level 3[2]

Ein Kompromiss zwischen DOM und rohem Strom ist ein an den SAX-`ContentHandler` angelehnter `XMLWriter` (Listing 12.7). Während Sie XML ausgeben, kontrolliert dieser, ob die Tags in der richtigen Reihenfolge geschrieben werden. Dieses Vorgehen ist allerdings nur sinnvoll, wenn Sie das Dokument nicht anschließend im selben Prozess mit Stylesheets manipulieren wollen.

```
package com.tagtraum.perf.xml;

import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import java.io.*;
import java.util.*;

public class XMLWriter extends DefaultHandler {
```

2 Gemäß Entwurf vom 9. April 2002.

```
   private PrintWriter out;
   private String encoding;
   private List stack;
   private boolean canonical;

   public XMLWriter(OutputStream out, String encoding,
         boolean canonical) throws UnsupportedEncodingException {
      this.out = new PrintWriter(new BufferedWriter(
         new OutputStreamWriter(out, encoding)));
      stack = new ArrayList();
      ...
   }

   ...

   public void startElement(String uri, String local, String raw,
         Attributes attrs) throws SAXException {
      stack.add(raw);
      insert(stack.size());
      out.print('<');
      out.print(raw);
      if (attrs != null) {
         // schreibe Attribute
         ...
      }
      out.print('>');
   }

   public void endElement(String uri, String local, String raw)
         throws SAXException {
      String expectedTag = (String) stack.remove(stack.size() - 1);
      if (!raw.equals(expectedTag))
         throw new SAXException("Expected </" + expectedTag
            + "> instead of </" + raw + ">.");
      out.print("</");
      out.print(raw);
      out.print('>');
   }

   // Schließt alle Tags, die noch offen sind.
   public void endAllPendingElements() {
      while (!stack.isEmpty()) {
         out.print("</");
         out.print(stack.remove(stack.size() - 1));
         out.print('>');
      }
   }
   ...

}
```

Listing 12.7: Ausschnitt aus einem XMLWriter, der überprüft, ob Elemente in der richtigen Reihenfolge ausgegeben werden

12.5 DOM-Bäume traversieren

Wie oben bereits erwähnt, ist die Methode `document.getElementsByTagName()` des Crimson-Parsers aus Sun JDK 1.4.0 nicht gerade ein Ausbund an Spritzigkeit. Grundsätzlich ist diese Methode jedoch bei großen Objekt-Bäumen ohnehin nicht zu empfehlen, da sie eine neue Liste mit allen passenden Knoten erstellt. Dies ist sehr speicherintensiv. Wesentlich eleganter ist der Zugriff über einen `org.w3c.dom.traversal.NodeIterator` der W3C-Schnittstelle *DOM Level 2 Traversal and Range* (*http://www.w3.org/TR/2000/REC-DOM-Level-2-Traversal-Range-20001113/*). Leider wird auch dieses API nicht von Sun JDK 1.4.0 unterstützt. Zu den oben bereits vorgestellten zwei Methoden alle Tagnamen auszugeben (`getElementsByTagName("*")` und rekursives Traversieren) gesellt sich also noch eine dritte hinzu. Listing 12.8 zeigt ein Beispiel.

```
package com.tagtraum.perf.xml;

import org.w3c.dom.*;
import org.w3c.dom.traversal.*;
import javax.xml.parsers.*;
import java.io.File;

public class DOMNodeIteratorDemo {

   public static void main(String[] args) throws Exception {
      DocumentBuilderFactory builderFactory
         = DocumentBuilderFactory.newInstance();
      DocumentBuilder documentBuilder
         = builderFactory.newDocumentBuilder();
      Document document = documentBuilder.parse(new File(args[0]));
      // Folgendes funktioniert nur, wenn die DOM-Implementierung
      // DOM Level 2 Traversal and Range unterstützt.
      DocumentTraversal traversable = (DocumentTraversal)document;
      NodeIterator iterator = traversable
         .createNodeIterator(document, NodeFilter.SHOW_ELEMENT,
          null, true);
      Element element;
      while ((element = (Element) iterator.nextNode()) != null) {
         System.out.println(element.getTagName());
      }
   }
}
```

Listing 12.8: `NodeIterator`, *der die Namen aller Tags einer XML-Datei ausgibt*

Um die drei Arten, alle Tags eines DOM-Baums zu besuchen, zu vergleichen, lesen wir jeweils Faust II ein und messen anschließend die Zeit, die zum Traversieren benötigt wird, sowie den dadurch verursachten zusätzlichen Speicherverbrauch. Da Crimson in der aktuellen Version keine `NodeIteratoren` unterstützt, führen wir diesen Test mit Apache Xerces-J 2.0.1 und der Sun JDK 1.4.0 Client-VM durch.

Xerces-J 2.x verfügt über die Fähigkeit, Knoten erst bei Bedarf endgültig zu initialisieren (Deferred Node Expansion). Daher führen wir je einen Lauf mit und einen ohne dieses Feature durch. Beim Lauf mit Deferred Node Expansion erfolgt ein Teil des Parsens erst zur Zeit der Traversion, während ohne Deferred Node Expansion das Dokument vollständig vor der Traversion geparsed wird. Deshalb erwarten wir beim Lauf mit Deferred Node Expansion eine wesentlich längere Iterationszeit und einen größeren Zuwachs des Speicherverbrauchs. Dies trifft auch so ein.

Tabelle 12.1 zeigt, dass die selbst geschrieben `traverse()`-Methode am besten abschneidet, da sie zwar genauso schnell ist wie der `NodeIterator`, aber einen geringeren Speicherverbrauch verursacht. Es fällt zudem auf, dass der Speicherverbrauchszuwachs der `NodeIterator`-Variante mit Deferred Node Expansion nur halb so groß ist wie der der anderen Varianten mit Deferred Node Expansion. Anscheinend kann der Xerces-`NodeIterator` also vom Wissen über Implementierungsdetails des Xerces-Parsers profitieren und so den Speicherverbrauch minimieren. Daher gilt:

Es kann sich lohnen, einen `NodeIterator` *zu benutzen, sofern dieser vom Parser angeboten wird.*

Traversionsart	**Normalisierte Ausführungszeit**	**Zuwachs im Speicherverbrauch**
`NodeIterator`	100%	16 Kbyte
`getElementsByTagName("*")`	220%	110 Kbyte
`traverse()`	100%	13 Kbyte
`NodeIterator` mit Deferred Node Expansion	1442%	917 Kbyte
`getElementsByTagName("*")` mit Deferred Node Expansion	1522%	1.994 Kbyte
`traverse()` mit Deferred Node Expansion	1242%	1.841 Kbyte

Tabelle 12.1: Vergleich der Performance verschiedener Methoden zum Besuchen aller Tags

Tatsächlich gibt es noch einen vierten Weg über alle Tags zu iterieren – nämlich mit Hilfe eines `org.w3c.dom.traversal.TreeWalkers`. Während der `NodeIterator` eine sequenzielle Sicht auf den Baum liefert, offeriert der `TreeWalker` eine hierarchische Sicht. Das heißt man kann nicht nur vor und zurück, sondern auch hoch und runter navigieren. Gegenüber dem `NodeIterator` bietet ein `TreeWalker` für unser Beispiel jedoch keinerlei Vorteile.

12.6 XML komprimieren

XML ist mit jedem simplen Editor lesbar und performante Parser sind für fast alle Plattformen zu haben – Fakten, die begeistern. Wenn es um Speicherverbrauch und effiziente Netzwerk-Nutzung geht, ist XML jedoch nicht gerade für seine Sparsamkeit bekannt. Da lässt die Freude schon mal ein wenig nach.

Zum Glück lassen sich Speicherhunger und Bandbreitenschwund mit Daten-Kompression bekämpfen. Und gerade wegen seiner exorbitanten Redundanz kann man XML ganz hervorragend komprimieren. Für die Lagerung in langsamen Speichern wie Festplatten bieten sich System-Werkzeuge wie *gzip*, *zip* und *bzip* an. Zumindest *gzip* und *zip* werden auch von Java exzellent unterstützt. Die entsprechenden Klassen befinden sich im Paket `java.util.zip` und sind sehr performant. Das Speicherproblem lässt sich also leicht lösen.

Was bleibt, ist das Netzwerkproblem. Wenn Sie beide Enden der Kommunikation kontrollieren oder ein Protokoll verwenden, das verschiedene Kompressionsalgorithmen unterstützt, können Sie auch hier *gzip* benutzen. Eines dieser Protokolle ist HTTP.

12.6.1 HTTP

Wenn Webbrowser eine Anforderung an einen Webserver schicken, signalisieren sie meistens, dass sie in der Lage sind, gzip-kodierte Dokumente zu verarbeiten. Dies geschieht mit dem `Accept-Encoding`-Header. Hier ein Beispiel für eine Anforderung von Microsoft Internet Explorer 5.5:

```
GET /test.html HTTP/1.1
Connection: Keep-Alive
Accept-Language: de
Host: 127.0.0.1:8080
Accept-Encoding: gzip, deflate
Accept: */*
User-Agent: Mozilla/4.0 (compatible; MSIE 5.5; Windows NT 5.0)
```

Und hier eine Anforderung von Netscape 6.2.1:

```
GET /test.html HTTP/1.1
Connection: keep-alive
Accept-Language: de-DE
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept: text/xml, application/xml, application/xhtml+xml,
   text/html;q=0.9, image/png, image/jpeg, image/gif;q=0.2,
   text/plain;q=0.8, text/css, */*;q=0.1
Host: 127.0.0.1:8080
Accept-Charset: ISO-8859-1, utf-8;q=0.66, *;q=0.66
Keep-Alive: 300
User-Agent: Mozilla/5.0 (Windows; U; Windows NT 5.0; de-DE;
   rv:0.9.4) Gecko/20011128 Netscape6/6.2.1
Cache-Control: max-age=0
```

Beide Browser akzeptieren Antworten, die mit gzip oder deflate komprimiert wurden. Darüber hinaus akzeptiert Netscape auch noch *compress*. Wenn Sie also ein Servlet schreiben, das große XML-Dokumente produziert, können Sie vorgehen wie in Listing 12.9.

```
package com.tagtraum.perf.servlet;

import javax.servlet.*;
import javax.servlet.http.*;
import java.io.*;
import java.util.zip.GZIPOutputStream;

public abstract class CompressorServlet extends HttpServlet {

   public void doGet(HttpServletRequest request,
         HttpServletResponse response) throws IOException,
         ServletException {
      OutputStream out = getOutputStream(request, response);
      try {
         writeToStream(request, response, out);
      }
      finally {
         // Sicherstellen, dass der GZIP-Stream korrekt beendet
         // wird.
         out.close();
      }
   }

   // Gibt einen OutputStream zurück. Wenn der Client gzip
   // akzeptiert, handelt es sich um einen GZIPOutputStream.
   // Im Header der Antwort wird zudem die verwendete Kodierung
   // gesetzt.
   private OutputStream getOutputStream(HttpServletRequest request,
         HttpServletResponse response) throws IOException {
      OutputStream out;
      if (acceptsGzip(request)) {
         response.setHeader("Content-Encoding", "gzip");
         out = new GZIPOutputStream(response.getOutputStream());
      } else {
         out = response.getOutputStream();
      }
      return out;
   }

   // Gibt an, ob der Client gzip akzeptiert.
   private boolean acceptsGzip(HttpServletRequest request) {
      String acceptEncoding = request.getHeader("Accept-Encoding");
      return acceptEncoding != null
         && acceptEncoding.indexOf("gzip") != -1;
   }

   // Wird von Subklassen anstelle von doGet() implementiert.
   // Da HttpServletResponse keine setOutputStream()-Methode hat,
   // übergeben wir den Strom als separates Argument.
```

```
    public abstract void writeToStream(HttpServletRequest request,
          HttpServletResponse response, OutputStream out)
          throws IOException, ServletException;
}
```

Listing 12.9: Einfaches Servlet, das Kompression mit gzip unterstützt

Sie müssen lediglich noch die `writeToStream()`-Methode implementieren und in ihr das XML-Dokument in den zur Verfügung gestellten `OutputStream` schreiben. Alles Weitere erledigen der Browser und HTTP.

Etwas komplizierter wird es, wenn Sie aus dem Servlet heraus eine andere Ressource per `include()` einbinden wollen. Sie müssten unseren speziellen Ausgabestrom an die andere Ressource übergeben. Dies ist mit Servlet API 2.2 jedoch nur schwierig möglich.

Die Lösung für dieses Problem bringt Servlet API 2.3. Sie heißt `javax.servlet.Filter`.

Listing 12.10 zeigt einen entsprechenden Filter für *gzip*. Zunächst wird festgestellt, ob der Client *gzip* akzeptiert. Ist dies der Fall, wird ein `HttpServletResponseWrapper` instanziiert, dessen `getOutputStream()`-Methode überschrieben ist. Statt des normalen `ServletOutputStream` wird ein `ServletOutputStream` zurückgegeben, der in einen `GZIPOutputStream` mündet, Gleiches gilt für den Writer. Nachdem das nächste Element der Filterkette aufgerufen wurde, wird der Writer geflushed und der Strom geschlossen, um sicherzustellen, dass der gzip-Strom korrekt beendet wird.

```
package com.tagtraum.perf.servlet;

import javax.servlet.*;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.http.HttpServletResponseWrapper;
import java.io.IOException;
import java.io.OutputStream;
import java.io.PrintWriter;
import java.io.OutputStreamWriter;
import java.util.zip.GZIPOutputStream;

public class GZIPFilter implements Filter {
   public void doFilter(ServletRequest request,
         final ServletResponse response, FilterChain chain)
         throws IOException, ServletException {
      HttpServletRequest req = (HttpServletRequest) request;
      HttpServletResponse res = (HttpServletResponse) response;
      if (acceptsGzip(req)) {
         res = new HttpServletResponseWrapper(res) {
            private ServletOutputStream wrappedOut;
            private PrintWriter wrappedWriter;
```

```
            public ServletOutputStream getOutputStream()
                  throws IOException {
               if (wrappedOut == null) {
                  setHeader("Content-Encoding", "gzip");
                  final OutputStream out
                     = new GZIPOutputStream(
                     response.getOutputStream());
                  wrappedOut = new ServletOutputStream() {
                     public void write(int b) throws IOException {
                        out.write(b);
                     }

                     public void write(byte b[], int off,
                           int len) throws IOException {
                        out.write(b, off, len);
                     }

                     public void close() throws IOException {
                        out.close();
                     }
                  };
               }
               return wrappedOut;
            }

            public PrintWriter getWriter() throws IOException {
               if (wrappedWriter == null) {
                  wrappedWriter = new PrintWriter(
                     new OutputStreamWriter(getOutputStream(),
                     getCharacterEncoding()));
               }
               return wrappedWriter;
            }

         };
         try {
            chain.doFilter(req, res);
         } finally {
            // Sicherstellen, dass der Strom korrekt geschlossen
            // wird.
            res.getWriter().flush();
            res.getOutputStream().close();
         }
      }
      else {
         chain.doFilter(req, res);
      }
   }

   private boolean acceptsGzip(HttpServletRequest request) {
      String acceptEncoding = request.getHeader("Accept-Encoding");
```

```
        return acceptEncoding != null
            && acceptEncoding.indexOf("gzip") != -1;
    }

    public void init(FilterConfig config) throws ServletException {
    }

    public void destroy() {
    }
}
```

Listing 12.10: Gzip-Kompressionsfilter für Servlets

Wie oben bereits erwähnt, unterstützen die meisten Browser automatisch *gzip* als Kompressionsformat. `java.net.HttpURLConnection` bietet diese Unterstützung von Haus aus nicht. Sie können jedoch leicht nachhelfen:

```
URL url = ...
HttpURLConnection connection
   = (HttpURLConnection)url.openConnection();
connection.addRequestProperty("Accept-Encoding", "gzip");
InputStream in = connection.getInputStream();
String contentEncoding
   = connection.getHeaderField("Content-Encoding");
if (contentEncoding != null && contentEncoding.equals("gzip")) {
   in = new GZIPInputStream(in);
}
// benutze 'in'
...
```

Somit steht großen XML-Dokumenten nichts mehr im Wege.

12.6.2 Binärformate

Wie wir gesehen haben, löst Kompression mit Standardalgorithmen das Bandbreiten-Problem für den allgemeinen Fall recht effizient. Wir wollen uns noch einen Spezialfall anschauen.

Angenommen, Sie haben eine Client-Server-Anwendung, bei der XML-Dokumente vom Server an den Client gesendet werden müssen. Verhältnismäßig verfüge der Server über sehr viel mehr Rechenleistung und Speicher als der Client. Zudem sei die Bandbreite zum Client begrenzt und die XML-Dokumente müssen dynamisch auf dem Server erstellt werden.

Das beschriebene Szenario trifft auf viele Anwendungen für Kleingeräte wie PDA und Handys zu, die mit einem Server kommunizieren. Das binäre XML-Format WBXML des WAP-Forums (*http://www.wapforum.org/*) adressiert genau diese Problemstellung und erreicht zwei Ziele:

- kompakte Darstellung von XML-Dokumenten
- leichtes und schnelles Parsen

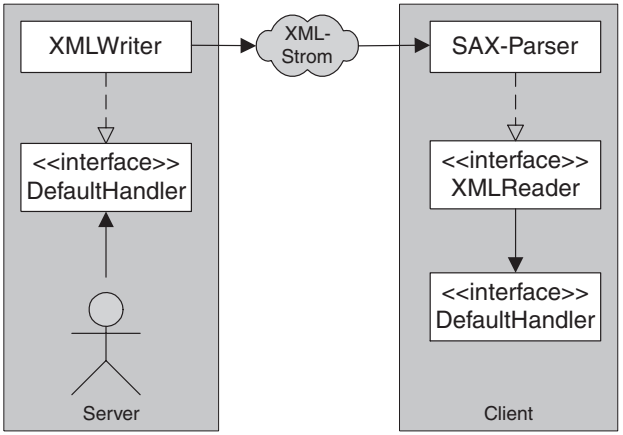

Abbildung 12.6: Typisches Szenario zum Übertragen von XML von einem Rechner zum anderen

Leider gibt es jedoch kaum freie Software, die WBXML für gewöhnliche J2SE-Anwendungen verwendet. Insbesondere konnte ich keine Bibliothek finden, die WBXML mit DOM- oder SAX-Schnittstelle umsetzt.

Wir wollen dennoch ausprobieren, welchen Nutzen wir aus einer Binärdarstellung ziehen können. Angenommen, der Server produziert XML, indem er den oben beschriebenen `XMLWriter` benutzt. Er ruft also die Methoden der SAX-Schnittstelle `ContentHandler` auf. `XMLWriter` konvertiert die Methodenaufrufe in Zeichen und schreibt diese in einen Ausgabestrom. Der Client wiederum liest diesen Strom mit einem SAX-Parser.

Wenn es uns gelingt, statt des `XMLWriters` eine andere Klasse zu benutzen, die nicht XML erzeugt, sondern eine Binärdarstellung, können wir auf der Clientseite diese Binärdarstellung parsen und wiederum die Methoden eines SAX-`ContentHandler` aufrufen. Der Unterschied liegt lediglich in der Darstellung des Dokuments, während es vom Server zum Client transportiert wird. Die Vorteile sind die gleichen wie bei WBXML: kompaktere Darstellung und leichtes Parsen. Zudem können wir Software, die bereits auf die SAX-Schnittstellen zugeschnitten ist, wieder verwenden.

Das Binärformat ist schnell definiert. Jede Methode der `ContentHandler`-Schnittstelle bekommt eine Zahl zugewiesen, Tag- und Attributnamen sowie Entitäten und Namensräume werden in einer Tabelle hinterlegt und alle anderen Zeichen werden einfach als UTF-Strings geschrieben. Zusätzlich definieren wir noch einige Steuerungszeichen, die es uns erlauben, das Dokument in Blöcke mit je einer eigenen Stringtabelle zu unterteilen, das Ende einer Attributliste zu markieren sowie zwischen mehreren

Codeseiten der Stringtabelle zu wechseln. Eine Codeseite soll jeweils auf 14 verschiedene Einträge (ein halbes Byte abzüglich zweier global eindeutiger Steuerungstokens) der Stringtabelle verweisen können.

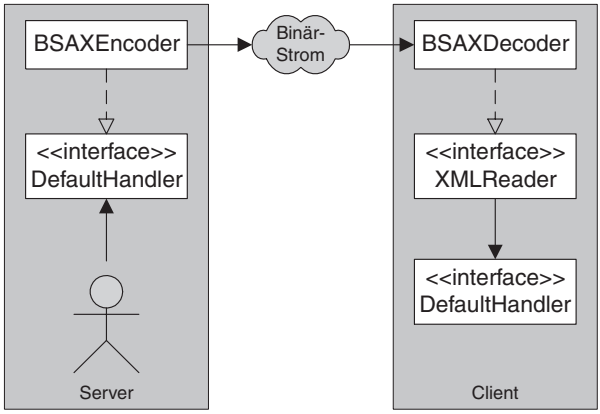

Abbildung 12.7: XML-Datenübertragung mit BSAX

Listing 12.11 zeigt die wichtigsten Methoden aus der Kodierklasse `BSAXEncoder`, die die Klasse `DefaultHandler` erweitert. Um Platz zu sparen werden die Daten mit einem (hier nicht weiter beschriebenen) `XtendedDataOutputStream` geschrieben, der in der Lage ist, halbe Bytes zu schreiben.

```
package com.tagtraum.perf.xml;

import org.xml.sax.*;
import org.xml.sax.helpers.DefaultHandler;
import java.io.*;
import java.util.*;
import com.tagtraum.perf.io.XtendedDataOutputStream;

public class BSAXEncoder extends DefaultHandler {

   // Global eindeutige Tokens
   static final int SWITCH_CODE_PAGE = 0;
   static final int END = 1;

   // Steuerungstoken
   static final int NEW_BLOCK = 2;

   // Methodentoken
   static final int START_DOCUMENT = 3;
   static final int END_DOCUMENT = 4;
   static final int START_ELEMENT = 5;
```

```
static final int END_ELEMENT = 6;
static final int CHARACTERS = 7;
static final int IGNORABLE_WHITESPACE = 8;
static final int PROCESSING_INSTRUCTION = 9;
static final int SKIPPED_ENTITY = 10;
static final int START_PREFIXMAPPING = 11;
static final int END_PREFIXMAPPING = 12;
static final int NOTATION_DECLARATION = 13;
static final int UNPARSED_ENTITY_DECLARATION = 14;

static final int RESERVED_CODES = 2;
static final int CODES_PER_CODEPAGE = 16;
static final int DEFINABLE_CODES_PER_CODEPAGE =
   CODES_PER_CODEPAGE-RESERVED_CODES;

private static int DEFAULT_BLOCKSIZE = 8*1024;
private static ResourceBundle localStrings
   = ResourceBundle.getBundle(
       "com.tagtraum.perf.xml.localStrings");

private XtendedDataOutputStream out;
private XtendedDataOutputStream bufferedDataStream;
private ByteArrayOutputStream buffer;
private int currentCodePage;
private Map symbolTable;
private List symbolList;
private Map oldSymbolTable;
private List oldSymbolList;
private int code = RESERVED_CODES;
private int codePage;
private int blockSize;
private boolean firstFlush;
private StringBuffer characters;
private boolean ignoreIgnorableWhitespace;

public BSAXEncoder(OutputStream out, int blockSize) {
   this.out = out instanceof BufferedOutputStream
      ? new XtendedDataOutputStream(out)
      : new XtendedDataOutputStream(
         new BufferedOutputStream(out));
   this.blockSize = blockSize;
   buffer = new ByteArrayOutputStream(blockSize + 256);
   bufferedDataStream = new XtendedDataOutputStream(buffer);
   symbolTable = new HashMap();
   symbolList = new ArrayList();
   oldSymbolTable = new HashMap();
   oldSymbolList = new ArrayList();
   firstFlush = true;
   characters = new StringBuffer(1024);
}
```

```
public boolean isIgnoreIgnorableWhitespace() {
   return ignoreIgnorableWhitespace;
}

public void setIgnoreIgnorableWhitespace(
      boolean ignoreIgnorableWhitespace) {
   this.ignoreIgnorableWhitespace = ignoreIgnorableWhitespace;
}

private void writeVersion() throws IOException {
   // version 1.0
   out.write(1); // major
   out.write(0); // minor
}

private void checkBufferSize() throws IOException {
   if (buffer.size() > blockSize) {
      flush();
   }
}

private void flush() throws IOException {
   if (firstFlush) {
      writeVersion();
      firstFlush = false;
   }

   bufferedDataStream.flush();
   writeSymbolTable();
   out.write(buffer.toByteArray());
   out.flush();
   buffer.reset();
}

public void startElement(String uri, String localName,
                         String qName, Attributes attributes)
      throws SAXException {
   // Schreibe alle Zeichen, die sich zuvor angesammelt haben.
   flushCharacters();
   try {
      // Schreibe Methodentoken
      writeCode(START_ELEMENT);
      // Schreibe vollständigen Namen des Elements
      writeSymbol(getSymbol(qName));
      // Schreibe Attribute, falls vorhanden
      if (attributes != null) {
         for (int i = 0, len =attributes.getLength(); i < len;
               i++) {
            // Schreibe Attributnamen und -typ
            writeSymbol(getSymbol(attributes.getQName(i)));
            writeSymbol(getSymbol(attributes.getType(i)));
            // Schreibe Wert des Attributs als UTF-String
```

```
                bufferedDataStream.writeUTF(attributes.getValue(i));
            }
         }
         // Markiere Ende der Attributliste
         writeCode(END);
         checkBufferSize();
      } catch (IOException ioe) {
         throw new SAXException(localStrings.getString(
            "exception_during_io"), ioe);
      } catch (RuntimeException re) {
         throw new SAXException(localStrings.getString(
            "unexpected_exception_during_io"), re);
      }
   }

   public void endElement(String uri, String localName,
         String qName) throws SAXException {
      flushCharacters();
      try {
         writeCode(END_ELEMENT);
         checkBufferSize();
      } catch (IOException ioe) {
         throw new SAXException(localStrings.getString(
            "exception_during_io"), ioe);
      } catch (RuntimeException re) {
         throw new SAXException(localStrings.getString(
            "unexpected_exception_during_io"), re);
      }
   }

   // Aufeinander folgende characters()-Aufrufe werden zu einem
   // Aufruf zusammengefasst.
   public void characters(char ch[], int start, int length)
         throws SAXException {
      if (length > 0) {
         characters.append(ch, start, length);
      }
   }

   public void flushCharacters() throws SAXException {
      if (characters.length() > 0) {
         try {
            writeCode(CHARACTERS);
            bufferedDataStream.writeUTF(characters.toString());
            checkBufferSize();
            // Sicherstellen, dass der StringBuffer nicht zu
            // groß wird
            if (characters.length() > 8*1024)
               characters = new StringBuffer(1024);
            else characters.setLength(0);
         } catch (IOException ioe) {
```

```
            throw new SAXException(localStrings.getString(
                "exception_during_io"), ioe);
         } catch (RuntimeException re) {
            throw new SAXException(localStrings.getString(
                "unexpected_exception_during_io"), re);
         }
      }
   }

   public void ignorableWhitespace(char ch[], int start,
         int length) throws SAXException {
      flushCharacters();
      // ignorableWhitespace wird nur geschrieben, wenn dies auch
      // erwünscht ist.
      if (!ignoreIgnorableWhitespace && length > 0) {
         try {
            writeCode(IGNORABLE_WHITESPACE);
            bufferedDataStream.writeUTF(new String(ch,
                start, length));
            checkBufferSize();
         } catch (IOException ioe) {
            throw new SAXException(localStrings.getString(
                "exception_during_io"), ioe);
         } catch (RuntimeException re) {
            throw new SAXException(localStrings.getString(
                "unexpected_exception_during_io"), re);
         }
      }
   }

   // andere ContentHandler-Methoden nach demselben Schema
   …

   public void fatalError(SAXParseException e)
         throws SAXException {
      throw e;
   }

   // Gibt ein Symbol für einen String zurück.
   private Symbol getSymbol(String string) {
      Symbol s = (Symbol) symbolTable.get(string);
      if (s == null) s = (Symbol) oldSymbolTable.get(string);
      if (s == null) {
         if (code == CODES_PER_CODEPAGE) {
            codePage++;
            code = RESERVED_CODES;
         }
         s = new Symbol(string, code, codePage);
         symbolTable.put(string, s);
         symbolList.add(string);
         code++;
      }
```

```
      return s;
   }

   private void writeCode(int code) throws IOException {
      bufferedDataStream.writeHalfByte(code);
   }

   // Schreibt ein Symbol-Code und falls nötig zuvor das
   // Switchcodepage-Token gefolgt von der Codepage-Nummer des
   // zu schreibenden Codes.
   private void writeSymbol(Symbol s) throws IOException {
      if (currentCodePage != s.getCodePage()) {
         bufferedDataStream.writeHalfByte(SWITCH_CODE_PAGE);
         bufferedDataStream.write(s.getCodePage());
         currentCodePage = s.getCodePage();
      }
      bufferedDataStream.writeHalfByte(s.getCode());
   }

   // Schreibt die Symboltabelle.
   private void writeSymbolTable() throws IOException {
      ByteArrayOutputStream bout = new ByteArrayOutputStream();
      XtendedDataOutputStream symbolTableStream
         = new XtendedDataOutputStream(bout);
      symbolTableStream.writeHalfByte(NEW_BLOCK);
      symbolTableStream.writeInt(symbolTable.size());
      for (int i = 0, c = symbolList.size(); i < c; i++) {
         symbolTableStream.writeUTF((String) symbolList.get(i));
      }
      symbolTableStream.flush();
      out.write(bout.toByteArray());

      symbolList.clear();
      oldSymbolTable.putAll(symbolTable);
      symbolTable.clear();
   }

   // Klasse zum schnellen Zugriff auf Code und Codepage eines
   // Symbols.
   private static class Symbol {
      private int code;
      private int codePage;
      private String value;

      public Symbol(String value, int code, int codePage) {
         this.value = value;
         this.code = code;
         this.codePage = codePage;
      }
```

```
        public int getCode() {
           return code;
        }

        public int getCodePage() {
           return codePage;
        }

        public String getValue() {
           return value;
        }
     }
  }
```

Listing 12.11: Ausschnitt aus der Klasse BSAXEncoder

Das Gegenstück zum `BSAXEncoder`, der `BSAXDecoder`, implementiert das `XMLReader`-Interface. Somit können die beiden Klassen zur Kommunikation verwendet werden, ohne dass XML-Produzent oder Konsument etwas davon mitbekommen. Aus Platzgründen möchte ich darauf verzichten, hier den gesamten Quellcode wiederzugeben. Sie finden ihn jedoch auf der zum Buch gehörenden CD-ROM bzw. auf der Website.

Wie Tabelle 12.2 zeigt, ist es sehr schwierig, *gzip* zu schlagen, wenn es um Kompression geht. BSAX zeigt die besten Ergebnisse, wenn wenige Tags oft benutzt werden und viel Whitespace aus dem Original-Dokument entfernt werden kann. Bei kurzen Nachrichten, solchen mit sich kaum wiederholenden Tags oder Dokumenten mit einem hohen Text-Anteil ist die Kompressionsrate eher mäßig.

Zur effizienten Kompression ist gzip in der Regel einem Binärformat vorzuziehen.

Eine Ausnahme hiervon kann lediglich sein, wenn beiden Parteien die Symboltabelle bekannt ist. Dies ist bei BSAX jedoch nicht der Fall. Zudem können Sie nicht davon ausgehen, dass *gzip* in einer J2ME-Umgebung unterstützt wird. Eventuell können Sie sich jedoch mit *zip* behelfen.

Dokument	BSAX	BSAX ohne Whitespace	gzip
Sehr kurze Soap-Nachricht (281 Byte)	83%	83%	68%
Soap-Nachricht (3,6 Kbyte)	92%	92%	49%
Web-Archiv-Deployment-Deskriptor (7,1 Kbyte)	57%	57%	17%
Faust II (550 Kbyte)	81%	68%	24%
Adressliste ohne Whitespace (1,2 Mbyte)	39%	39%	24%

Tabelle 12.2: Prozentanteil der ursprünglichen Größe, auf die die komprimierten Dateien reduziert wurden. Kleiner ist also besser.

Wenn es nur um Kompression geht, ist BSAX also keine Wunderwaffe. Anders sieht das beim Parsen aus (Tabelle 12.3). Gegenüber dem SAX-Parser aus dem Sun JDK 1.4.0 ist BSAX für fast alle getesteten Dokumente schneller. Lediglich für Faust II ist die BSAX-Parsezeit ein wenig schlechter als mit SAX.

Dokument	**BSAX**	**BSAX ohne Whitespace**	**SAX**	**gzip + SAX**
Sehr kurze Soap-Nachricht (281 Byte)	100%	95%	1376%	1560%
Soap-Nachricht (3,6 Kbyte)	100%	100%	172%	209%
Web-Archiv-Deployment-Deskriptor (7,1 Kbyte)	100%	102%	164%	191%
Faust II (550 Kbyte)	100%	72%	72%	90%
Adressliste ohne Whitespace (1,2 Mbyte)	100%	100%	156%	187%

Tabelle 12.3: Normalisierte Ausführungszeit des Parsens verschiedener Dokumente. Kleiner ist schneller.

Zusammenfassend bedeutet dies:

Um ein effizientes Parsen auf der Clientseite zu garantieren, kann es sich lohnen, zur Datenübertragung ein Binärformat zu benutzen.

13 Applikationen starten

Noch bevor eine einzige Zeile Ihrer Applikation ausgeführt wird, vergeht während des Starts der Java VM so einige Zeit. Die VM und unbedingt notwendige Klassen werden in den Speicher geladen und initialisiert. Erst dann wird die Hauptklasse Ihrer Applikation geladen und ausgeführt.

IBM JDK 1.3.0	Sun JDK 1.3.1	Sun JDK 1.4.0
345	209	273

Tabelle 13.1: Anzahl von Klassen, die geladen werden müssen, bevor die auszuführende Klasse geladen wird. Gemessen mit Windows-Versionen der VMs.

Je nach VM sind zwischen 200 und 350 Klassen das unbedingte Minimum (Tabelle 13.1). Hinzu kommen Ihre eigenen Klassen und alle Klassen, die Ihre Klassen benutzen, sowie jene, die von den bereits geladenen Klassen benutzt werden. Wenn Ihre Applikation schnell starten muss, lohnt es sich daher, sich ein wenig mehr mit dem Laden und Initialisieren von Klassen auseinander zu setzen.

13.1 Klassen laden und initialisieren

Vor allem stellt sich hier die Frage nach dem Wann. Gewöhnlich werden Klassen und Schnittstellen in folgenden Situationen geladen:

- Wenn der `ClassLoader` einer Klasse symbolische Referenzen auf andere Klassen oder Schnittstellen auflöst
- Wenn eine Klasse benutzt wird und daher automatisch die Klasse sowie alle ihre Superklassen und Schnittstellen geladen werden
- Wenn eine Klasse explizit vom Nutzer durch die `loadClass()`-Methode eines `ClassLoaders` oder die entsprechenden `Class.forName()`-Methoden geladen wird

Gewöhnlich lösen VMs symbolische Referenzen auf andere Klassen und Schnittstellen erst spät auf (*Lazy Resolution*). Das heißt, wenn eine Klasse `A` eine Klasse `B` benutzt, sind die Chancen gut, dass die symbolische Referenz von `A` auf `B` erst aufgelöst wird, wenn `B` tatsächlich von `A` benutzt wird. Erst das Auflösen der symbolischen Referenz auf `B` führt zum Laden von `B`.

Nur wenn `B` eine Superklasse von `A` ist, muss `B` früher geladen werden, da bereits beim Laden von `A` alle symbolischen Referenzen auf Superklassen aufgelöst werden müssen.

Wenn eine Klasse geladen ist, existiert eine binäre Repräsentation der Klasse im Speicher. Dessen Korrektheit wird verifiziert und statische Felder werden mit ihren Standardwerten bzw. solchen Werten initialisiert, die sich nach der Übersetzung nicht mehr ändern können (Übersetzungszeit-Konstanten). Solche Werte sind beispielsweise Strings, die einer als `final` deklarierten Variable zugewiesen werden [vgl. Gosling00, §15.28]. Das bedeutet nicht, dass die Klasse initialisiert ist. Es ist außerdem noch keine Zeile Code ausgeführt worden. Eine Klasse ist erst dann initialisiert, wenn

- alle statischen Initialisierungsblöcke ausgeführt wurden
- alle statischen Variablen initialisiert wurden

Gemäß Java-Sprachspezifikation werden Klassen nur dann initialisiert, wenn eine der folgenden Bedingungen zutrifft [vgl. Gosling00, §12.4]:

- Eine Instanz der Klasse wird erzeugt.
- Eine Klassenmethode der Klasse wird aufgerufen.
- Es wird einem Klassenattribut der Klasse ein Wert zugewiesen.
- Ein Klassenattribut der Klasse, dessen Referenz keine Übersetzungszeit-Konstante ist, wird benutzt.
- Bestimmte Methoden aus dem `java.lang.reflect`-Paket werden aufgerufen.

So führt beispielsweise folgender Code dazu, dass die Klasse `java.util.HashMap` geladen und initialisiert wird, da eine Klasse instanziiert wurde:

```
// HashMap wird instanziiert
HashMap map = new HashMap();
```

Folgender Code führt jedoch nicht zum Laden der Klasse, da lediglich eine leere Referenz angelegt wird:

```
// Die Klasse HashMap wird weder instanziiert noch werden
// irgendwelche statischen Methoden oder Attribute benutzt.
HashMap map =null;
```

Ebenso führt das Ausführen der Klasse `One` nicht dazu, dass `Two` geladen und der statische Initialisierungs-Block (Zeile 9-11) ausgeführt wird. Dies ist so, weil `aMessage` eine Übersetzungszeit-Konstante ist. Ihr Wert kann sich nach der Übersetzung nicht mehr ändern.

```
01 public class One {
02    public static void main(String[] args) {
03       // Gibt 'A Message' aus
04       System.out.println(Two.aMessage);
05    }
06 }
07
08 public class Two {
09    static {
10       System.out.println("Dieser Text wird nicht ausgegeben!");
11    }
12    public static final String aMessage = "Eine Nachricht";
13 }
```

Die einzige Ausgabe des obigen Codes ist also:

```
Eine Nachricht
```

13.2 Verzögertes Klassenladen

Um also eine Klasse erst zu laden und zu initialisieren, wenn sie benutzt wird, können Sie beispielsweise folgenden Code benutzen:

```
private AClass aClass;
public AClass getInstance() {
   if (aClass == null) {
      aClass = new AClass();
   }
   return aClass;
}
```

Sie dürfen jedoch vorher nicht auf Klassenmethoden oder statische, veränderbare Attribute der Klasse zugreifen. Denken Sie zudem daran, die Methode `getInstance()` ordnungsgemäß zu synchronisieren (*Kapitel 9.1.4 Double-Check-Idiom*), wenn sie von mehreren Threads benutzt wird.

Besonders sinnvoll ist diese Technik für Applikationen, bei denen von vornherein klar ist, dass bestimmte Teile erst später oder eventuell gar nicht benutzt werden.

13.3 Frühes Klassenladen

Unter Umständen macht es Ihnen wenig aus, wenn Ihre Applikation beim Start ein wenig mehr Zeit zum Klassenladen benötigt, solange danach alles reibungslos klappt. Ist dies der Fall, können Sie das Laden von Klassen am besten kontrollieren, indem Sie es selbst erledigen. Geeignet sind dazu die beiden `java.lang.Class.forName()`-Methoden.

So sorgt folgender Code dafür, dass die Klasse `AClass` geladen und initialisiert wird, wenn `BClass` ausgeführt wird.

```
public AClass {
   static {
      System.out.println("AClass wurde initialisiert.");
   }
}

public BClass {
   public static void main(String[] args)
         throws ClassNotFoundException{
      Class.forName("AClass");
   }
}
```

Die Ausgabe lautet entsprechend:

```
AClass wurde initialisiert.
```

Wenn Sie jedoch folgende, leicht modifizierte Version der Klasse `BClass` ausführen, bleibt die Ausgabe leer.

```
public BClass {
   public static void main(String[] args) {
      Class.forName("AClass",
         false, BClass.class.getClassLoader());
   }
}
```

Entscheidend hierfür ist der zweite Parameter der `forName()`-Methode. Ist dieser `false`, so wird die verlangte Klasse zwar geladen, nicht jedoch initialisiert. Die Initialisierung erfolgt automatisch, sobald dies nötig ist.

Manchmal kann es sich lohnen, Teile einer Applikation von einem separaten Thread im Hintergrund laden zu lassen, nachdem der Benutzer bereits angefangen hat, mit der Applikation zu arbeiten. Dabei können diese Teile vollständig erzeugt oder lediglich die benötigten Klassen geladen und initialisiert werden, um dann später ein schnelles Erzeugen der Objekte zu begünstigen. Dies ist insbesondere dann interes-

sant, wenn die Klassen nicht in einem Jar- oder Zip-Archiv gespeichert sind und nur über ein langsames Netzwerk geladen werden können, da die Klassen dann einzeln geladen werden müssen. *Grundsätzlich sollten Sie jedoch Ihre Klassen wenn möglich immer in Jars verpacken!*

Listing 13.1 zeigt einen solchen Hintergrund-Klassenlader, der sich nach dem Erzeugen mit `start()` starten lässt. Wenn Sie sich sicher sein wollen, dass die geladenen Klassen nicht wieder von der Speicherbereinigung aus dem Speicher entfernt werden, sollten Sie den VM Kommandozeilen-Parameter `-Xnoclassgc` verwenden. Dadurch wird die Speicherbereinigung für Klassen-Objekte abgeschaltet. Dies kann jedoch zu Speicherproblemen bei Programmen führen, die Klassen erzeugen, laden, benutzen und wieder wegwerfen, wie dies beispielsweise JSP-Engines tun.

```
package com.tagtraum.perf.classloading;

public class BackgroundClassLoader extends Thread {
   private volatile static int count;
   private String[] classnames;
   private boolean initialize;
   private ClassLoader classLoader;

   public BackgroundClassLoader(String[] classnames,
         boolean initialize, ClassLoader classLoader) {
      super("BackgroundClassLoader - " + count++);
      if (classnames == null) throw new NullPointerException();
      this.classnames = classnames;
      // Wenn kein ClassLoader übergeben wurde, nehmen wir den
      // eigenen, sonst den übergebenen.
      this.classLoader = classLoader == null
         ? this.getClass().getClassLoader() : classLoader;
      this.initialize = initialize;
      // Wenn die Applikation nicht mehr läuft, sollten dieser
      // Thread auch terminieren. Daher markieren wir ihn als
      // Daemon.
      setDaemon(true);
      // Natürlich wollen wir nur Klassen laden, wenn wirklich
      // nichts anderes zu tun ist. Der Effekt dieser Zeile ist
      // jedoch stark vom Betriebssystem abhängig! Evtl. muss
      // die Priorität manuell hochgesetzt werden, damit der Thread
      // überhaupt zum Zuge kommt.
      setPriority(Thread.MIN_PRIORITY);
   }

   public void run() {
      for (int i = 0; i < classnames.length; i++) {
         try {
            Class.forName(classnames[i], initialize, classLoader);
         } catch (Exception e) {
```

```
                System.err.println(getName()
                    + ": Failed to load class " + classnames[i]);
            }
        }
    }
}
```

Listing 13.1: BackgroundClassLoader lädt Klassen in einem Hintergrund-Thread.

13.4 Geschwätziges Klassenladen

Natürlich macht das frühe Klassenladen nur Sinn, wenn Sie wissen, welche Klassen von Ihrer Applikation überhaupt benötigt werden. Um rauszubekommen, welche Klassen von der VM geladen werden, können Sie die VM-Option `-verbose:class` setzen.

Beispiel:

```
java -verbose:class -classpath <classpath> <main class>
```

Dies führt dazu, dass die VM ausgibt, welche Klasse geladen wurde. Leider wird dabei kein Zeitstempel ausgegeben. Wenn Sie versuchen, das Klassenladen zu beeinflussen, ist dies dennoch Ihr bestes Verifikations-Werkzeug.

Die Ausgabe von Sun JDK 1.4.0 sieht folgendermaßen aus:

```
[Opened C:\j2sdk1.4.0\jre\lib\rt.jar]
[Opened C:\j2sdk1.4.0\jre\lib\sunrsasign.jar]
[Opened C:\j2sdk1.4.0\jre\lib\jsse.jar]
[Opened C:\j2sdk1.4.0\jre\lib\jce.jar]
[Opened C:\j2sdk1.4.0\jre\lib\charsets.jar]
[Loaded java.lang.Object from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.io.Serializable from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.Comparable from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.CharSequence from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.String from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.Class from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.Cloneable from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.ClassLoader from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.System from C:\j2sdk1.4.0\jre\lib\rt.jar]
[Loaded java.lang.Throwable from C:\j2sdk1.4.0\jre\lib\rt.jar]
...
```

Falls Sie nur daran interessiert sind, wann einige bestimmte Klassen initialisiert werden, können Sie diesen Klassen auch einen Klasseninitialisierer spendieren, der kundtut, wann die Klasse initialisiert wurde:

```
public class AClass {
   static {
      System.out.println(new java.util.Date() + ": "
         + AClass.class.getName());
   }
   ...
}
```

13.5 Klassenarchive

Grundsätzlich können Sie Ihre Klassen entweder einfach in einem Verzeichnis oder in Jar- bzw. Zip-Archiven speichern. Welche Option für Ihre Applikation besser ist, hängt im Wesentlichen davon ab, wie schnell Ihre Applikation auf die Klassen zugreifen kann.

Angenommen Ihre Klassen liegen auf der lokalen Festplatte, dann ist die Zugriffsgeschwindigkeit sehr hoch. Liegen die Klassen jedoch auf einem schlecht angebundenen Webserver, ist sie niedrig. Zudem spielt das Zugriffsprotokoll eine Rolle. Wird beispielsweise bei einem Applet als Codebasis ein Verzeichnis und nicht eine Jar-Datei gewählt, so muss im schlechtesten Fall[1] für jede Klassendatei eine neue TCP/IP-Verbindung geöffnet und ein neuer HTTP-Request an den Webserver gesandt werden. Sie können sich ausmalen, was dies bedeutet, wenn Sie nach allen Regeln der Kunst Hunderte von übersichtlichen, kleine Klassen programmiert haben.

Meist ist es daher günstiger, Jar-Dateien zu verwenden. Hier stellt sich noch die Frage, ob Sie komprimierte oder unkomprimierte Jars verwenden. Standardmäßig erstellt das `jar`-Programm komprimierte Dateien. Sie können dies jedoch mit dem Parameter `-0` (Null) abstellen. Zwar wird dann das entstehende Archiv größer, dafür muss aber während des Klassenladens nicht mehr dekomprimiert werden, was zu geringen Zeitvorteilen führen kann.

Im Grunde müssen Sie zwischen Geschwindigkeit der Datenübertragung und Geschwindigkeit der Dekompression abwägen. Bei langsamer Datenübertragung lohnen sich komprimierte Jar-Dateien (Tabelle 13.2). Die meist lokal gespeicherten JDK-Klassen befinden sich dagegen in einer unkomprimierten Jar-Datei namens *rt.jar*. Im Zweifelsfall ist eine komprimierte Jar-Datei vorzuziehen, da das Netzwerk meist der Flaschenhals ist und nicht die CPU.

1 Dies trifft ein, wenn entweder der HTTP-Server oder der -Client lediglich HTTP/1.0 ohne Keep-Alive sprechen. Falls Server und Client Keep-Alive beherrschen, muss trotzdem immer noch für jede Klasse ein separater Request gesandt werden.

Speicherort der Klassen	Dateiformat
Lokale Festplatte	Unkomprimierte Jar-Datei
Schnelles Netzlaufwerk	Unkomprimierte Jar-Datei
Langsames Netzlaufwerk	Komprimierte Jar-Datei
Schneller Intranet Webserver	Unkomprimierte Jar-Datei
Langsamer Webserver	Komprimierte Jar-Datei

Tabelle 13.2: Wie Sie Ihre Applikation in Abhängigkeit vom Wo speichern sollten

Selbst für das lokale Dateisystem ist auf jeden Fall davon abzuraten, die Klassen unarchiviert abzulegen, da dann auf jede Klasse einzeln zugegriffen werden muss. Die Ladezeit kann sich so leicht verdoppeln.

Abzuraten ist auch von Zip-Archiven. Gründe hierfür sind:

- Es gibt verschiedene Zip-Formate, aber nur ein Jar-Format.
- Zip-Archive werden in Webapplikationen oft nicht erkannt, wenn sie unter */WEB-INF/lib* abgelegt werden.[2]
- Zip-Dateien lassen sich nicht signieren und verfügen in der Regel nicht über ein Manifest.

Natürlich gibt es auch zu diesen Regeln Ausnahmen. Wenn Sie beispielsweise eine sehr große Applikation als Applet ausliefern wollen, kann es sein, dass es zu lange dauert, erst darauf zu warten, dass das Jar-Archiv vollständig geladen ist, bevor irgendetwas passiert. In diesem Fall kann es günstiger sein, wenn Sie Ihre Klassen in einem Verzeichnis und nicht in einer Jar-Datei ablegen, da Sie dann, während Ihr Programm schon läuft, später benötigte Klassen nachladen können. Bevor Sie sich zu solch einem Schritt entschließen, sollten Sie jedoch auf jeden Fall Tests durchführen.

Bedenken Sie zudem, dass Browser und auch das Java-Plugin Jar-Archive und Klassen cachen. Bei einer Jar-Datei muss nur einmal überprüft werden, ob die gecachte Version noch aktuell ist, bei einzelnen Klassen muss jede einzelne Klasse überprüft werden.

13.6 Start-Fenster für große Applikationen

Wenn Sie feststellen, dass Ihre GUI-Applikation sehr lange zum Starten benötigt, können Sie die Startzeit mit den oben beschriebenen Techniken zum späten Klassenladen vielleicht etwas verkürzen. Es gibt jedoch einen Punkt, an dem nichts mehr hilft.

2 Dies führt zu haarsträubenden Problemen insbesondere mit JDBC-Treibern von IBM und Oracle, die teilweise immer noch in Zip-Archiven ausgeliefert werden.

Spätestens, wenn Sie diesen Punkt erreicht haben und die Startzeit immer noch zu lang ist, sollten Sie darüber nachdenken, wie Sie dem Nutzer die Zeit verkürzen können. Eine sehr einfache Technik ist es, als Erstes einen separaten Thread zu starten, der kontinuierlich etwas im Konsole-Fenster Ihrer Applikation ausgibt. Leider hat jedoch nicht jede Java-Applikation solch ein Fenster, weshalb diese Option für viele Anwendungen ausscheidet. Stattdessen können Sie ein Startbild (Splashscreen) anzeigen. Das ist ein rahmenloses Fenster, in dem nur eine Grafik angezeigt wird.

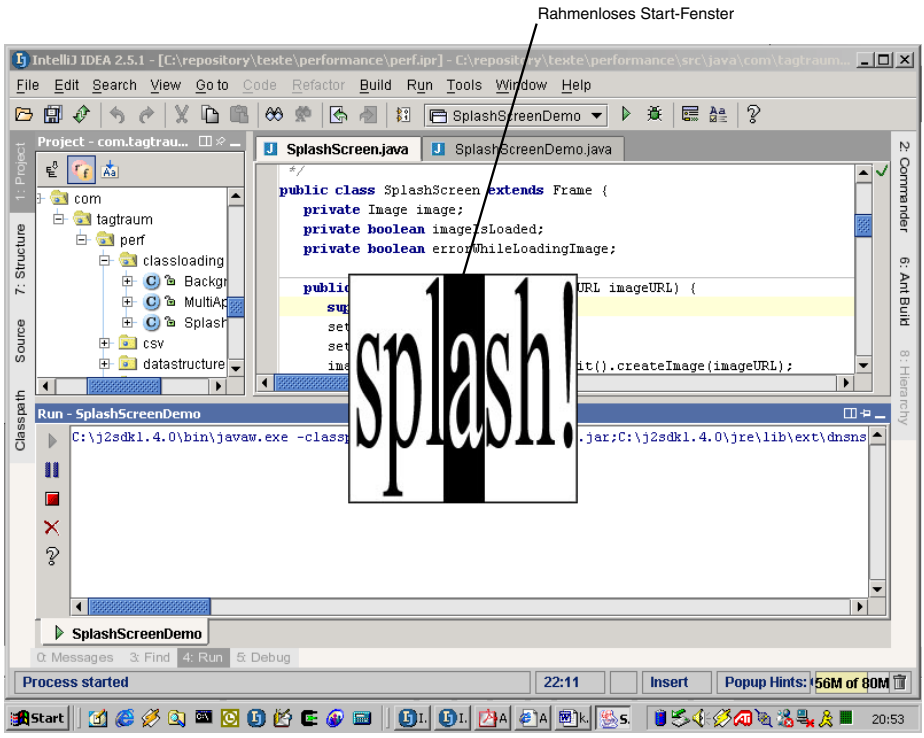

Abbildung 13.1: Rahmenloses Start-Fenster

Um möglichst schnell etwas auf den Bildschirm zu zaubern, beschränkt sich der Code zu Abbildung 13.1 auf das absolute Minimum. Es wird lediglich festgestellt, wie groß das darzustellende Bild ist und wo es platziert werden soll. Sonst nichts.

```
package com.tagtraum.perf.classloading;

import javax.swing.*;
import java.awt.*;
import java.awt.image.ImageObserver;
import java.net.URL;
```

```
public class SplashScreen extends Frame {
   private Image image;
   private boolean imageIsLoaded;
   private boolean errorWhileLoadingImage;

   public SplashScreen(String title, URL imageURL) {
      super(title);
      setCursor(Cursor.WAIT_CURSOR);
      setUndecorated(true);
      image = Toolkit.getDefaultToolkit().createImage(imageURL);
      // Zunächst müssen wir diesen Frame als ImageObserver für das
      // Image registrieren.
      image.getHeight(this);
      // Dann warten wir, bis Breite und Höhe bekannt sind.
      waitForWidthAndHeight();
      // Falls wir das Image nicht korrekt laden konnten, zeigen
      // wir eine Fehlermeldung an.
      if (errorWhileLoadingImage) {
         JOptionPane.showMessageDialog(this,
            "Failed to load image from " + imageURL, "Error",
            JOptionPane.ERROR_MESSAGE);
      } else {
         // Höhe und Breite sind nun bekannt und wir können den
         // Frame entsprechend skalieren und in der Mitte
         // des Bildschirms positionieren.
         int imageWidth = image.getWidth(this);
         int imageHeight = image.getHeight(this);
         DisplayMode dm = GraphicsEnvironment
            .getLocalGraphicsEnvironment()
            .getDefaultScreenDevice().getDisplayMode();
         setBounds((dm.getWidth() - imageWidth) / 2,
            (dm.getHeight() - imageHeight) / 2, imageWidth,
            imageHeight);
         setVisible(true);
      }
   }

   public void update(Graphics g) {
      // Vermeidet Flackern
      paint(g);
   }

   public void paint(Graphics g) {
      // Zeichnet das Bild
      g.drawImage(image, 0, 0, this);
   }

   // Wird vom Image aufgerufen, wenn neue Informationen über das
   // Bild bekannt werden.
   public synchronized boolean imageUpdate(Image img,
         int infoflags, int x, int y, int width, int height) {
```

```
        // Falls Abort- oder Error-Flags gesetzt sind, setze den
        // Fehlerzustand.
        errorWhileLoadingImage = errorWhileLoadingImage
           || (infoflags & (ImageObserver.ABORT
           | ImageObserver.ERROR)) != 0;
        // Falls ein Fehler vorlag oder Höhe und Breite bekannt
        // sind, setze imageIsLoaded auf true
        imageIsLoaded = errorWhileLoadingImage || imageIsLoaded ||
           ((infoflags & ImageObserver.WIDTH) != 0
           && (infoflags & ImageObserver.HEIGHT) != 0);
        // Benachrichtige wartende Threads, dass Höhe und Breite nun
        // bekannt sind.
        if (imageIsLoaded) notifyAll();
        return super.imageUpdate(img, infoflags, x, y, width, height)
           && (imageIsLoaded);
     }

     // Wartet darauf, dass Höhe und Breite des Bildes bekannt sind.
     public synchronized void waitForWidthAndHeight() {
        try {
           while (!imageIsLoaded) {
              wait();
           }
        } catch (InterruptedException ie) {
           // Ignorieren
        }
     }
  }
```

Listing 13.2: SplashScreen zeigt einen rahmenlosen Frame, der lediglich ein Bild enthält.

Wenn Sie eine Klasse wie `SplashScreen` (Listing 13.2) benutzen wollen, stellen Sie sicher, dass Sie als Erstes diese Klasse instanziieren und anzeigen. Und zwar sollte dies in der `main()`-Methode Ihrer Hauptklasse passieren, bevor irgendetwas anderes passiert. Ist das geschehen, ist der Benutzer erst mal beruhigt und harrt der Dinge, die da kommen.

```
public BigSlowApplicationStarter {
   public static void main(String[] args) {
      Frame splashScreen = new SplashScreen("BigSlowApplication",
         SplashScreen.class.getResource("/splashscreen.jpg"));
      BigSlowApplication bsa = new BigSlowApplication();
      bsa.setVisible(true);
      splashScreen.dispose();
   }
}
```

Listing 13.3: Beispiel für die Benutzung der Klasse SplashScreen

13.7 Mehrere Applikationen in einer VM starten

Dank der Classloader-Technologie ist es recht einfach, in Java mehr als ein Programm von einer VM ausführen zu lassen. Die Vorteile sind offensichtlich:

- Weniger Speicherverbrauch
- Kürzere Startzeiten

Leider gibt es auch Nachteile:

- Applikationen lassen sich nicht mehr individuell mit VM-Parametern optimieren und parametrisieren (-D-Optionen)
- Stürzt die VM ab, stürzen viele Applikationen ab und nicht nur eine
- Da der Heap potenziell sehr groß wird, können Speicherbereinigungspausen lang und lästig werden

Gerade auf Maschinen mit geringem Hauptspeicher und begrenzter Prozessorleistung kann es sich jedoch lohnen, mehrere Applikationen in einer VM laufen zu lassen.

Abbildung 13.2 zeigt einen einfachen Applikations-Starter mit zwei laufenden Programmen. Zum Starten eines Programms muss man lediglich die Hauptklasse, den Klassenpfad sowie optional Start-Argumente angeben und START drücken.

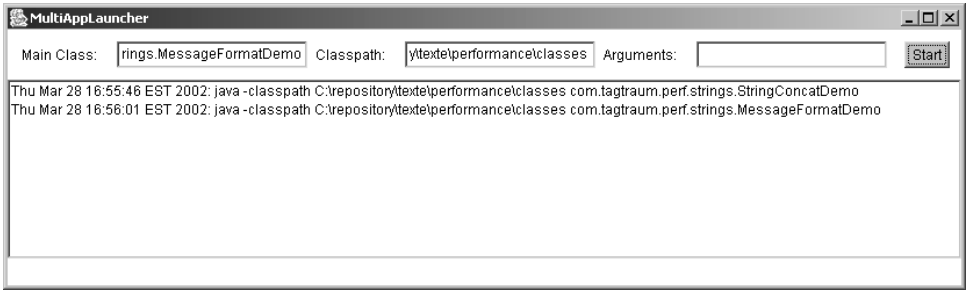

Abbildung 13.2: Java-Applikations-Starter

Mit den angegebenen Daten wird ein neuer `ClassLoader` instanziiert. Mit diesem `ClassLoader` wird dann die Hauptklasse geladen und die `main()`-Methode mit den angegebenen Argumenten aufgerufen. Das Aufrufen der `main()`-Methode geschieht dabei in einem neuen `AppRunner`-Thread (Listing 13.4). Auf diese Weise können wir beliebig viele Applikation starten, solange der Speicher reicht!

Denkbar ist anstelle eines grafischen Applikations-Starters natürlich auch einer ohne grafische Benutzeroberfläche [vgl. Wilson00, S.79].

```
package com.tagtraum.perf.classloading;

import java.awt.*;
import java.awt.event.*;
import java.io.File;
import java.lang.reflect.Method;
import java.net.*;
import java.util.Date;
import java.util.StringTokenizer;

public class MultiAppLauncher extends Frame implements ActionListener {

   private Label statusBar;
   private java.awt.List appList;
   private TextField classPathField;
   private TextField mainClassField;
   private TextField argumentsField;

   public MultiAppLauncher() {
      super("MultiAppLauncher");
      // GUI aufbauen
      addWindowListener(new WindowAdapter() {
         public void windowClosing(WindowEvent e) {
            dispose();
            // System.exit(0) ist nicht mehr nötig in JDK 1.4.0
         }
      });
      classPathField = new TextField(20);
      mainClassField = new TextField(20);
      argumentsField = new TextField(20);
      Label classPathLabel = new Label("Classpath:");
      Label mainClassLabel = new Label("Main Class:");
      Label argumentsLabel = new Label("Arguments:");
      Button startButton = new Button("Start");
      startButton.addActionListener(this);
      appList = new List(10);
      statusBar = new Label();

      Panel dataPanel = new Panel();
      ((FlowLayout) dataPanel.getLayout())
         .setAlignment(FlowLayout.LEFT);
      dataPanel.add(mainClassLabel);
      dataPanel.add(mainClassField);
      dataPanel.add(classPathLabel);
      dataPanel.add(classPathField);
      dataPanel.add(argumentsLabel);
      dataPanel.add(argumentsField);

      Panel startButtonPanel = new Panel();
      ((FlowLayout) dataPanel.getLayout())
```

```
            .setAlignment(FlowLayout.RIGHT);
        startButtonPanel.add(startButton);

        Panel northPanel = new Panel();
        northPanel.add(dataPanel);
        northPanel.add(startButtonPanel);

        add(northPanel, BorderLayout.NORTH);
        add(appList, BorderLayout.CENTER);
        add(statusBar, BorderLayout.SOUTH);
        setSize(400, 400);
        pack();
        setVisible(true);
    }

    public void actionPerformed(ActionEvent e) {
        try {
            AppRunner appRunner = new AppRunner(
              classPathField.getText(), mainClassField.getText(),
              argumentsField.getText());
            // Zeige Thread in der Liste an.
            appList.add(appRunner.getName());
            // Starte Applikation im eignen Thread.
            appRunner.start();
        } catch (Exception exception) {
            statusBar.setText(exception.toString());
        }
        repaint();
    }

    public static void main(String[] args) {
        new MultiAppLauncher();
    }

    // Spezialisierter Thread, der eine Applikation ausführt.
    private class AppRunner extends Thread {

        private final Class[] PARAMETERTYPES_FOR_MAIN
            = new Class[]{String[].class};

        private String[] args;
        private ClassLoader classLoader;
        private Class mainClass;
        private Method mainMethod;
        private String classPath;

        public AppRunner(String classPath, String mainClass,
              String args) throws ClassNotFoundException,
              MalformedURLException, NoSuchMethodException {
            super(new Date().toString() + ": java -classpath "
```

```
      + classPath + " " + mainClass + " " + args);
   this.args = createArgsArray(args);
   this.classPath = classPath;
   this.classLoader = createClassLoader(classPath);
   this.mainClass = this.classLoader.loadClass(mainClass);
   this.mainMethod = this.mainClass.getMethod("main",
      PARAMETERTYPES_FOR_MAIN);
}

// Führt die main()-Methode der Hauptklasse in einem eigenen
// Thread auf.
public void run() {
   try {
      mainMethod.invoke(null, new Object[]{args});
      statusBar.setText(toString() + " exited.");
   } catch (Exception e) {
      System.err.println(this);
      e.printStackTrace();
      statusBar.setText(e.toString());
   }
   // Entferne diesen Thread aus der Liste, wenn er
   // beendet wurde.
   appList.remove(this.getName());
   repaint();
}

// Erstellt einen neuen ClassLoader für den angegebenen
// Klassenpfad.
private ClassLoader createClassLoader(String classPath)
      throws MalformedURLException {
   StringTokenizer st = new StringTokenizer(classPath,
      File.pathSeparator);
   URL[] urls = new URL[st.countTokens()];
   for (int i = 0; st.hasMoreTokens(); i++) {
      urls[i] = new File(st.nextToken()).toURL();
   }
   // Wir müssen den Bootclassloader als Elter-Loader
   // benutzen, damit sich verschiedene Applikationen nicht
   // in die Quere kommen. String muss vom Bootclassloader
   // geladen worden sein. Das nutzen wir aus!
   return new URLClassLoader(
      urls, String.class.getClassLoader());
}

// Erstellt einen String-Array als Argument für die
// main()-Methode.
private String[] createArgsArray(String args) {
   StringTokenizer st = new StringTokenizer(args, " \t\n\r");
   String[] argsArray = new String[st.countTokens()];
   for (int i = 0; st.hasMoreTokens(); i++) {
```

```
            argsArray[i] = st.nextToken();
        }
        return argsArray;
    }
  }
}
```

Listing 13.4: Klasse MultiAppLauncher

Letzte Worte

Es hat mir sehr viel Freude bereitet, dieses Buch zu schreiben, und ich hoffe, auch Sie mussten beim Lesen gelegentlich mal schmunzeln. Vor allem aber hoffe ich, dass dieses Buch für Sie nützlich war, dass es Ihnen half und helfen wird, performanteren Code zu schreiben und bessere Designs zu entwickeln.

Sowohl die verschiedenen Java VMs als auch die Java-Klassenbibliotheken werden in den nächsten Jahren immer schneller werden. Das heißt jedoch nicht, dass es weniger wichtig wird, guten, schnellen Code zu schreiben, da erfahrungsgemäß die Basistechnologien in gleichem Maße anspruchsvoller werden. Heute arbeiten wir mit EJB – und morgen? Das Performance-Problem bleibt bestehen. Daher wird es sich immer lohnen, sich mit verschiedenen Virtuellen Maschinen sowie Performance verbessernden Programmiertechniken auseinander zusetzen.

Sie müssen sich nur Ihre Neugierde bewahren.

Literatur

[Beck00] Beck, Kent: Extreme programming explained: embrace change. Addison-Wesley 2000.

[Bentley00] Bentley, Jon: Programming Pearls. 2. Auflage. Addison-Wesley 2000.

[Bloch02] Bloch, Joshua: Effektiv Java programmieren. München: Addison-Wesley 2002.

[Bulka00] Bulka, Dov: Java Performance and Scalability Band 1. Server-Side Programming Techniques. Addison-Wesley 2000.

[Gamma96] Gamma, Erich, Richard Helm, Ralph Johnson, John Vlissides: Entwurfsmuster: Elemente wiederverwendbarer objektorientierter Software. Bonn: Addison-Wesley 1996.

[Gosling00] Gosling, James, Bill Joy, Guy Steele, Gilad Bracha: The Java™ Language Specification. 2. Auflage. Addison-Wesley 2000.

[Jones96] Jones, Richard, Rafael Lins: Garbage Collection: Algorithms for Automatic Dynamic Memory Management. New York: Wiley & Sons 1996.

[Lea99] Lea, Doug: Concurrent Programming in Java™: Design Principles and Patterns. 2. Auflage. Addison-Wesley 1999.

[Lindholm99] Lindholm, Tim, Frank Yellin: The Java™ Virtual Machine Specification. 2. Auflage. Addison-Wesley 1999.

[Prechelt00] Prechelt, Lutz: An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program.
http://www.ipd.uka.de/~prechelt/Biblio/jccpprtTR.pdf

[Pugh01] Pugh, William (Hrsg.): »The Double-Checked Locking is Broken«-Declaration. University of Maryland. März 2001.
http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html

[Raymond 96] Raymond, Eric S. (Hrsg.): The New Hacker's Dictionary. Cambridge: MIT-Press 1996.

[Shirazi00] Shirazi, Jack: Java Performance Tuning. Cambridge: O'Reilly 2000.

[Weiss99] Weiss, Mark Allen: Data Structures & Algorithms in Java™. Addison-Wesley 1999.

[Wilson00] Wilson, Steve, Jeff Kesselman: Java™ Platform Performance: Strategies and Tactics. Addison-Wesley 2000.

Index

D

E

T

Copyright

Zusatzdaten

Möglicherweise liegt dem gedruckten Buch eine CD-ROM mit Zusatzdaten bei. Die Zurverfügungstellung dieser Daten auf der Website ist eine freiwillige Leistung des Verlags. Der Rechtsweg ist ausgeschlossen.