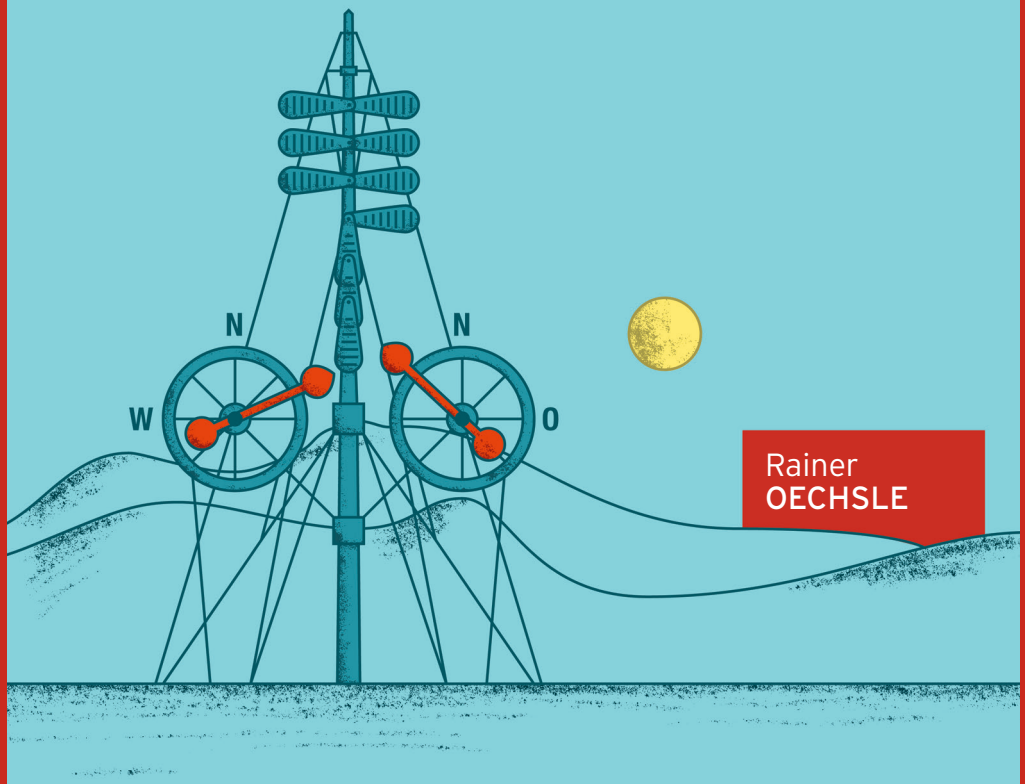
6., überarbeitete und erweiterte Auflage



Rainer
OECHSLE

Parallele und verteilte Anwendungen in JAVA

Alle Beispielprogramme zum Download auf plus.hanser-fachbuch.de

HANSER

Oechsle

Parallele und verteilte Anwendungen in Java

Lehrbücher zur Informatik

Begründet von
PROF. DR. MICHAEL LUTZ UND PROF. DR. CHRISTIAN MÄRTIN

weitergeführt von
PROF. DR. CHRISTIAN MÄRTIN
Hochschule Augsburg Fachbereich Informatik

Zu dieser Buchreihe

Die Werke dieser Reihe bieten einen gezielten Einstieg in grundlegende oder besonders gefragte Themenbereiche der Informatik und benachbarter Disziplinen. Alle Autoren verfügen über langjährige Erfahrung in Lehre und Forschung zu den jeweils behandelten Themengebieten und gewährleisten Praxisnähe und Aktualität.

Die Bände der Reihe können vorlesungsbegleitend oder zum Selbststudium eingesetzt werden. Sie lassen sich teilweise modular kombinieren. Wegen ihrer Kompaktheit sind sie gut geeignet, bestehende Lehrveranstaltungen zu ergänzen und zu aktualisieren.

Die meisten Werke stellen Ergänzungsmaterialien wie Lernprogramme, Software-Werkzeuge, Online-Kapitel, Beispielaufgaben mit Lösungen und weitere aktuelle Inhalte zur Verfügung.

Lieferbare Titel in dieser Reihe

- Rainer Oechsle, Parallele und verteilte Anwendungen in Java
- Wolfgang Riggert/Ralf Lübben, Rechnernetze
- Georg Stark, Robotik mit MATLAB
- Rolf Socher, Theoretische Grundlagen der Informatik

Rainer Oechsle

Parallele und verteilte Anwendungen in Java

6., aktualisierte Auflage

HANSER

Der Autor:

Prof. Dr. Rainer Oechsle, Hochschule Trier

Bibliografische Information der Deutschen Nationalbibliothek:
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de abrufbar.

Internet: www.hanser-fachbuch.de

Lektorat: Dipl.-Ing. Natalia Silakova-Herzberg
Herstellung: Frauke Schafft
Covergestaltung: Max Kostopoulos
Coverkonzept: Marc Müller-Bremer, www.rebranding.de, München
Titelbild: © Max Kostopoulos
Satz: Eberl & Kœsel Studio GmbH
Druck und Bindung: CPI books GmbH, Leck
Printed in Germany

Print-ISBN 978-3-446-46919-8
E-Book-ISBN 978-3-446-47348-5
E-Pub-ISBN 978-3-446-47504-5

Vorwort zur 6. Auflage

Dieses Buch handelt von der Entwicklung paralleler und verteilter Anwendungen in Java. Nach einem einleitenden Kapitel, in dem wichtige Begriffe wie Programme, Prozesse und Threads auch anhand einer Metapher aus dem täglichen Leben erläutert werden, lassen sich die folgenden acht Kapitel in drei Teile gruppieren:

- Entwicklung paralleler Anwendungen in Java (Kapitel 2 und 3): Das Buch beginnt mit einer relativ ausführlichen Einführung in das Gebiet der parallelen Programmierung. Die Sachverhalte sind für Neulinge oft anspruchsvoll, denn Programmcode, der bei rein sequenzieller Ausführung korrekt ist, kann im Fall einer parallelen Nutzung fehlerbehaftet sein. Der Einstieg in das Thema Parallelität wird im Zusammenhang mit Objektorientierung für viele noch problematischer. Denn zum einen muss man verstehen, dass es Thread-Objekte gibt, dass diese aber nicht identisch mit den parallelen Aktivitäten, den Threads selbst, sind. Zum anderen muss man begreifen lernen, dass es mehrere Objekte einer Klasse geben kann, dass aber ein einziges Objekt (quasi) gleichzeitig von mehreren Threads verwendet werden kann, d. h., dass dieselbe und unterschiedliche Methoden auf einem Objekt gleichzeitig parallel ausgeführt werden können. In diesem Buch wird in die Gedankenwelt der Parallelität mit zahlreichen Programmbeispielen behutsam eingeführt (Kapitel 2). Es werden dann aber auch die grundlegenden Ideen weiterer anspruchsvoller Konzepte aus der Java-Concurrent-Bibliothek wie das Fork-Join-Framework, sequenzielles und paralleles Data-Streaming sowie CompletableFutures behandelt, ohne auf alle Details dieser Konzepte einzugehen (Kapitel 3).
- Entwicklung von Anwendungen mit grafischer Benutzeroberfläche in Java (Kapitel 4): Grafische Benutzeroberflächen scheinen auf den ersten Blick nichts mit parallelen und verteilten Anwendungen zu tun zu haben. Bei näherem Hinsehen erkennt man aber durchaus Zusammenhänge. So wird zum Beispiel in diesem Buch ausführlich erläutert, welche negativen Effekte es bei naiver Programmierung für Anwendungen mit grafischer Benutzeroberfläche gibt, wenn eine länger dauernde Aktivität aufgrund einer Interaktion mit der grafischen Benutzeroberfläche gestartet wird. Insbesondere bei verteilten Anwendungen können länger dauernde Aktivitäten immer bei einer Kommunikation zwischen einem Client und einem Server über das Internet vorkommen. Die Probleme lassen sich mithilfe von Parallelität lösen. Da Client-Programme oft und Server-Programme manchmal eine grafische Benutzeroberfläche haben, spielt also das Thema Parallelität bei verteilten Anwendungen mit grafischer Benutzeroberfläche eine Rolle. Aber auch wichtige Strukturierungsprinzipien für lokale Programme mit grafischer Benutzeroberfläche wie das MVP-Architekturmuster (MVP: Model View Presenter) lassen sich auf verteilte Anwendungen übertragen.

- Entwicklung verteilter Anwendungen in Java (Kapitel 5 bis 9): Verteilte Anwendungen folgen häufig dem Client-Server-Prinzip. Auch hier besteht wieder ein enger Zusammenhang zur Parallelität, denn auf Server-Seite ist fast immer Parallelität notwendig, um mehrere Clients (quasi) gleichzeitig zu bedienen und somit die Bedienung eines Clients nicht beliebig lange durch die Bearbeitung eines länger dauernden Auftrags eines anderen Clients zu verzögern. Um die parallele Bearbeitung von Client-Aufträgen zu erreichen, müssen die Threads bei der Programmierung eines Servers auf Socket-Basis (Kapitel 5) selbst explizit erzeugt werden. Wenn RMI (Kapitel 6) oder Servlets und Java Server Faces (Kapitel 8) benutzt werden, dann werden Threads implizit (d. h. nicht im Programmcode der Anwendung) erzeugt. Dies muss man wissen und den Umgang damit beherrschen, wenn man korrekte Server-Programme schreiben will. Wenn die Kommunikation zwischen den Rechnern nicht mehr direkt, sondern über einen Vermittler (Broker) erfolgt, erreicht man eine losere Kopplung zwischen den kommunizierenden Parteien mit einigen Vorteilen (Kapitel 7). Viele verteilte Anwendungen nutzen heutzutage Cloud-Dienste. Es werden abschließend einige Cloud-Anwendungen unter Nutzung der Cloud-Dienste von Amazon präsentiert (Kapitel 9).

In vielen Lehrbüchern werden Parallelitätsaspekte bei Programmen mit grafischer Benutzeroberfläche oder bei Server-Programmen nicht genügend oder überhaupt nicht berücksichtigt. So habe ich einige Beispiele in Lehrbüchern gefunden, die das Thema Parallelität vollständig ignorieren. Folglich sind solche Programme mit fehlender Synchronisation oftmals nicht korrekt, was beim oberflächlichen Ausprobieren in der Regel (zum Glück oder leider?) nicht auffällt. In diesem Buch wird dagegen durchgängig für alle Anwendungen ein besonderes Augenmerk auf Parallelitätsaspekte gelegt.

Dieses Buch ist weder ein Handbuch mit allen Details, die man bei der Software-Entwicklung benötigt, noch ist es ein Überblicksbuch, in dem eine Fülle von Themen nur angerissen wird. Stattdessen versucht es seinem Charakter als Lehrbuch gerecht zu werden, indem es die Grundprinzipien zentraler Konzepte herausarbeitet. Der Fokus liegt auf den beiden eng miteinander verzahnten Themen Parallelität (Nebenläufigkeit) und Verteilung. Bei dem Thema parallele Programmierung ist es in der Praxis sicher ratsam, nach Möglichkeit die Klassen aus der Java-Concurrent-Klassenbibliothek zu verwenden, die in Kapitel 3 auch behandelt werden. Allerdings wird die Mehrzahl der Beispiele aus didaktischen Gründen ohne Nutzung dieser Bibliothek entwickelt. Ähnliches trifft auf Servlets und JSF (Java Server Faces) zu: In der Praxis wird man eher JSF nutzen, im Buch werden die meisten Beispiele mit Servlets entwickelt, um den Leserinnen und Lesern besser verständlich zu machen, was bei der Ausführung des Programms passiert. Auch in dieser Auflage behandle ich immer noch RMI sehr intensiv. Man mag der Meinung sein, dass RMI inzwischen veraltet ist, aber aus meiner Sicht ist RMI weiterhin eine sehr elegante und konsequent zu Ende gedachte Realisierung einer Client-Server-Kommunikation. Ich bin überzeugt davon, dass die intensive Beschäftigung mit RMI elementar wichtige Aspekte der Informatik wie zum Beispiel den Unterschied zwischen Call-by-value und Call-by-result gut verständlich macht. So ist die Beschäftigung mit den hier vorhandenen Inhalten nicht nur dazu da, um aktuell notwendige Kenntnisse und Fertigkeiten für die Berufswelt zu erlernen, sondern vor allem zum Erlernen grundlegender Informatikkonzepte. Aus diesem Grund ist die hier verwendete Programmiersprache Java auch nur ein Vehikel zur Darstellung unterschiedlicher Aspekte aus dem Bereich der Programmierung paralleler und verteilter Anwendungen. So wie dieselben Inhalte dieses Buchs statt in Deutsch in einer anderen Sprache wie Englisch oder

Französisch vermittelt werden könnten, so ließen sich viele der vorgestellten Konzepte auch durch Programmbeispiele in anderen Programmiersprachen wie etwa C++ oder C# illustrieren.

Bei der Auswahl des Lehrstoffs ist es immer auch schmerzlich, viele interessante und relevante Themen nicht tiefer oder gar nicht behandeln zu können aufgrund des begrenzten Umfangs des Buchs. So könnte das neue Kapitel 9 zum Thema Cloud Computing auch leicht so ausgedehnt werden, dass es ein ganzes Buch füllen würde. Zu den leider gar nicht behandelten Themen gehören beispielsweise die Bereiche Spring Boot und Kubernetes. Bei der Stoffauswahl muss man eben Kompromisse eingehen.

Für diese sechste Auflage wurden neben der Korrektur von Fehlern, die in der fünften Auflage bemerkt wurden, folgende wesentlichen Änderungen vorgenommen:

- Das Kapitel 7 zur indirekten Kommunikation wurde neu geschrieben.
- Außerdem ist Kapitel 9 zum Thema Cloud Computing neu dazugekommen.
- In Kapitel 5 wird im neu geschriebenen Abschnitt 5.7 nun auch verschlüsselte Kommunikation über SSL (Secure Socket Layer) bzw. TLS (Transport Layer Security) behandelt.
- Da Java EE (Enterprise Edition) von der Firma Oracle nicht mehr weiterentwickelt wird, sondern dies von der Eclipse Foundation übernommen wurde und nun den Namen Jakarta EE trägt, wurden Bezüge auf die Enterprise Edition von Java EE in Jakarta EE geändert.
- Der Abschnitt 6.8 (Laden von Klassen über das Netz) im RMI-Kapitel der fünften Auflage wurde ersatzlos gestrichen, da der dort benutzte Security Manager seit der Java-Version 17 „deprecated“ ist (d. h. nicht mehr benutzt werden sollte, da er in einer späteren Version nicht mehr vorhanden sein könnte).
- Ferner wurden mehrere kleinere Anpassungen vorgenommen wie z. B. die Ergänzung des Abschnitts 2.1.4 über parallele Abläufe oder die Ergänzung des Abschnitts 5.6.4 über horizontale Skalierung.

Die Beispielprogramme folgen gängigen Programmierkonventionen für Java bezüglich der Groß- und Kleinschreibung von Bezeichnern und dem Einrücken. Alle Bezeichner für Klassen, Schnittstellen, Methoden und Attribute sind einheitlich in Englisch geschrieben. Die Ausgaben, die von den Programmen erzeugt werden, sind jedoch alle in deutscher Sprache. In den abgedruckten Programmen wurden alle Package-Anweisungen entfernt (bis auf eine Ausnahme in Kapitel 9, Listing 9.6, weil bei diesem Beispiel auf den Package-Namen explizit Bezug genommen wird). Beachten Sie aber bitte, dass in der elektronischen Version, die Sie von einer der Webseiten zum Buch *www.hochschule-trier.de/puva6* (puva: **p**arallele **u**nd **v**erteilte **A**nwendungen) und *plus.hanser-fachbuch.de* beziehen können, die Klassen und Schnittstellen kapitelweise in unterschiedliche Packages gruppiert wurden (chapter2, chapter3 usw.). Alle Java-Programme wurden mit einem Java-Compiler der Version 14 übersetzt und ausprobiert.

Von den soeben bereits erwähnten Webseiten *www.hochschule-trier.de/puva6* und *plus.hanser-fachbuch.de* können Sie nicht nur alle Programme des Buchs in Form einer ZIP-Datei herunterladen. Auch nachträglich entdeckte Fehler werde ich mitsamt ihren Richtigstellungen und den Namen der Entdeckenden wie für die vorhergehende Auflage auf dieser Seite veröffentlichen. Ich habe zwar für diese Auflage alle entdeckten Fehler korrigiert, aber es ist nicht auszuschließen, dass bisher unentdeckte alte Fehler noch zutage treten werden,

und ich bin mir sehr sicher, dass ich bei der Überarbeitung der alten Texte und dem Schreiben der neuen Texte unabsichtlich neue Fehler eingebaut habe. Ich bin allen Leserinnen und Lesern dankbar für alle Arten von Fehlermeldungen, sowohl für die Meldung gravierender Fehler als auch einfacher Komma-, Tipp- oder Formatierungsfehler. Kommentare, Verbesserungsvorschläge und weitere Programmbeispiele, die Sie mir gerne senden können, werde ich ebenfalls auf dieser Webseite veröffentlichen, sofern sie mir für einen größeren Kreis interessant erscheinen.

Meinen Wunsch, geschlechtsneutrale Formulierungen zu verwenden, habe ich so umgesetzt, dass ich an manchen Stellen die männliche und weibliche Form angebe, an anderen Stellen nur die männliche oder nur die weibliche Form. Ich hoffe, dass sich dadurch Lesende beiderlei Geschlechts in gleicher Weise angesprochen fühlen.

Sollten Sie tiefer in die Thematik dieses Buches einsteigen wollen, dann empfehle ich Ihnen, das Modul „Fortgeschrittene Programmiertechniken (FOPT)“ im Rahmen des Informatik-Fernstudiums an der Hochschule Trier zu belegen. Hier können Sie zu den Themen dieses Buches Einsendeaufgaben bearbeiten, an zusätzlichen Tutorien (per Videokonferenz) teilnehmen sowie ein einwöchiges Präsenzpraktikum absolvieren. Nähere Informationen hierzu, insbesondere über die Voraussetzungen für die Belegung, über die Kosten sowie über die weiteren Module des Fernstudiums, finden Sie auf den Webseiten des Fachbereichs Informatik der Hochschule Trier (*www.hochschule-trier.de/informatik*).

Diese sechste Auflage hätte ohne die Hilfe der nachfolgend genannten Personen nicht bzw. nicht in dieser Form erscheinen können. Ich bedanke mich daher sehr gerne

- bei den für dieses Buch verantwortlichen Mitarbeiterinnen des Hanser-Verlags, Natalia Silakova und Christina Kubiak, für das Ergreifen der Initiative zur sechsten Auflage dieses Buchs, für die Möglichkeit der Erweiterung des Umfangs des Buchs um ca. 20 %, für die Umsetzung meines Vorschlags eines Semaphors als Titelbild sowie für die jederzeit schnelle Klärung aller meiner Fragen,
- bei Daniel Aggintus, Andreas Daum, Fabian Gibert, Robin Grell, Yanik Kaypinger, Marc Kutscher, Frank Seeger, Gunnar Sperveslage, Timo Vollmert und Thomas Zehrer für ihre Hinweise auf entdeckte Fehler und ihre Verbesserungsvorschläge, die alle auf der Webseite *www.hochschule-trier.de/puva5* veröffentlicht und in dieser sechsten Auflage berücksichtigt wurden,
- bei Stefan Bodenschatz, der mit mir zusammen an der Hochschule Trier eine Lehrveranstaltung zum Thema Cloud Computing aufgebaut und mehrfach durchgeführt hat, für seine zahlreichen Verbesserungsvorschläge zu einer früheren Fassung von Kapitel 9,
- und schließlich bei meiner Frau Ingrid für die gewährte Zeit zur Überarbeitung des Buchs.

Über positive und negative Bemerkungen zu diesem Buch, Hinweise auf Fehler und Verbesserungsvorschläge würde ich mich auch dieses Mal wieder freuen. Senden Sie Ihre Kommentare bitte in Form einer elektronischen Post an *oechsle@hochschule-trier.de*.

Konz-Oberemmel, im Februar 2022

Rainer Oechsle

Inhaltsverzeichnis

1 Einleitung

Computer-Nutzer dürften mit großer Wahrscheinlichkeit sowohl mit parallelen Abläufen auf ihrem eigenen Rechner als auch verteilten Anwendungen vertraut sein. So ist jeder Benutzer eines PC heutzutage gewohnt, dass z. B. gleichzeitig eine größere Video-Datei kopiert, ein Musikstück aus einer MP3-Datei abgespielt, ein Java-Programm übersetzt und ein Dokument in einem Editor oder Textverarbeitungsprogramm bearbeitet werden kann. Aufgrund der Tatsache, dass die Mehrzahl der genutzten Computer an das Internet angeschlossen ist und fast alle Menschen ein Handy nutzen, sind heute auch nahezu alle den Umgang mit verteilten Anwendungen wie der elektronischen Post, Messenger-Diensten oder dem World Wide Web gewohnt.

Dieses Buch handelt allerdings nicht von der Benutzung, sondern von der Entwicklung paralleler und verteilter Anwendungen mit Java. In diesem ersten einleitenden Kapitel werden zunächst einige wichtige Begriffe wie Parallelität, Nebenläufigkeit, Verteilung, Prozesse und Threads geklärt.

1.1 Parallelität, Nebenläufigkeit und Verteilung

Wenn mehrere Vorgänge gleichzeitig auf einem Rechner ablaufen, so sprechen wir von Parallelität oder Nebenläufigkeit (engl. concurrency). Diese Vorgänge können dabei echt gleichzeitig oder nur scheinbar gleichzeitig ablaufen: Wenn ein Rechner mehrere Prozessoren bzw. einen Mehrkernprozessor (Multicore-Prozessor) besitzt, dann ist echte Gleichzeitigkeit möglich. Man spricht in diesem Fall auch von echter Parallelität. Besitzt der Rechner aber nur einen einzigen Prozessor mit einem einzigen Kern, so wird die Gleichzeitigkeit der Abläufe nur vorgetäuscht, indem in sehr hoher Frequenz von einem Vorgang auf den nächsten umgeschaltet wird. Man spricht in diesem Fall von Pseudoparallelität oder Nebenläufigkeit. Die Begriffe Parallelität und Nebenläufigkeit werden in der Literatur nicht einheitlich verwendet: Einige Autoren verwenden den Begriff Nebenläufigkeit als Oberbegriff für echte Parallelität und Pseudoparallelität, für andere Autoren sind Nebenläufigkeit und Pseudoparallelität Synonyme. In diesem Buch wird der Einfachheit halber nicht zwischen Nebenläufigkeit und Parallelität unterschieden; mit beiden Begriffen sollen sowohl die echte als auch die Pseudoparallelität gemeint sein.

Wenn das gleichzeitige Ablaufen von Vorgängen auf mehreren Rechnern betrachtet wird, wobei die Rechner über ein Rechnernetz gekoppelt sind und darüber miteinander kommunizieren, spricht man von Verteilung (verteilte Systeme, verteilte Anwendungen).

Wir unterscheiden also, ob die Vorgänge auf einem Rechner oder auf mehreren Rechnern gleichzeitig ablaufen; im ersten Fall sprechen wir von Parallelität, im anderen Fall von Verteilung. Die Mehrzahl der Leserinnen und Leser dürfte vermutlich mit dieser Unterscheidung zufrieden sein. In manchen Fällen ist es aber gar nicht so einfach, zu entscheiden, ob ein gegebenes System einen einzigen Rechner oder eine Vielzahl von Rechnern darstellt. Betrachten Sie z. B. ein System zur Steuerung von Maschinen, wobei dieses System in einem Schaltschrank untergebracht ist, in dem sich mehrere Einschübe mit Prozessoren befinden. Handelt es sich hier um einen oder um mehrere kommunizierende Rechner? Zur Klärung dieser Frage wollen wir uns hier an die allgemein übliche Unterscheidung zwischen eng und lose gekoppelten Systemen halten: Ein eng gekoppeltes System ist ein Rechnersystem bestehend aus mehreren gekoppelten Prozessoren, wobei diese auf einen gemeinsamen Speicher (Hauptspeicher) zugreifen können. Ein lose gekoppeltes System (auch verteiltes System genannt) besteht aus mehreren gekoppelten Prozessoren ohne gemeinsamen Speicher (Hauptspeicher), die über ein Kommunikationssystem Nachrichten austauschen. Ein eng gekoppeltes System sehen wir als einen einzigen Rechner, während wir ein lose gekoppeltes System als einen Verbund mehrerer Rechner betrachten.

Parallelität und Verteilung schließen sich nicht gegenseitig aus, sondern hängen im Gegenteil eng miteinander zusammen: In einem verteilten System laufen auf jedem einzelnen Rechner mehrere Vorgänge parallel (echt parallel oder pseudoparallel) ab. Wie auch in diesem Buch noch ausführlich diskutiert wird, arbeitet ein Server im Rahmen eines Client-Server-Szenarios häufig parallel, um mehrere Clients gleichzeitig zu bedienen. Außerdem können verteilte Anwendungen, die für den Ablauf auf unterschiedlichen Rechnern vorgesehen sind, im Spezialfall auf einem einzigen Rechner parallel ausgeführt werden.

Sowohl Parallelität als auch Verteilung werden durch Hard- und Software realisiert. Bei der Software spielt das Betriebssystem eine entscheidende Rolle. Das Betriebssystem verteilt u. a. die auf einem Rechner gleichzeitig möglichen Abläufe auf die vorhandenen Prozessoren bzw. die vorhandenen Kerne des Rechners. Auf diese Art vervielfacht also das Betriebssystem die Anzahl der vorhandenen Prozessoren bzw. der vorhandenen Kerne virtuell. Diese Virtualisierung ist eines der wichtigen Prinzipien von Betriebssystemen, die auch für andere Ressourcen realisiert wird. So wird z. B. durch das Konzept des virtuellen Speichers ein größerer Hauptspeicher vorgegaukelt als tatsächlich vorhanden. Erreicht wird dies, indem immer die gerade benötigten Daten vom Hintergrundspeicher (Platte) in den Hauptspeicher transferiert werden.

1.2 Programme, Prozesse und Threads

Im Zusammenhang mit Parallelität bzw. Nebenläufigkeit und Verteilung muss zwischen den Begriffen Programm, Prozess und Thread (Ausführungsfaden) unterschieden werden. Da es einen engen Zusammenhang zu den Themen Betriebssysteme, Rechner und verteilte Systeme gibt, sollen alle diese Begriffe anhand einer Metapher verdeutlicht werden:

- Ein Programm entspricht einem Rezept in einem Kochbuch. Es ist statisch. Es hat keine Wirkung, solange es nicht ausgeführt wird. Dass man von einem Rezept nicht satt wird, ist hinlänglich bekannt.
- Einen Prozess kann man sich vorstellen als eine Küche und einen Thread als einen Koch. Ein Koch kann nur in einer Küche existieren, aber nie außerhalb davon. Umgekehrt muss sich in einer Küche immer mindestens ein Koch befinden. Alle Köche gehen streng nach Rezepten vor, wobei unterschiedliche Köche nach demselben oder nach unterschiedlichen Rezepten kochen können. Jede Küche hat ihre eigenen Pfannen, Schüsseln, Herde, Waschbecken, Messer, Gewürze, Lebensmittel usw. Köche in unterschiedlichen Küchen können sich gegenseitig nicht in die Quere kommen, wohl aber die Köche in einer Küche. Diese müssen den Zugriff auf die Materialien und Geräte der Küche koordinieren.
- Ein Rechner ist in dieser Metapher ein Haus, in dem sich mehrere Küchen befinden.
- Ein Betriebssystem lässt sich mit einem Hausmeister eines Hauses vergleichen, der dafür sorgt, dass alles funktioniert (z. B. dass immer Strom für den Herd da ist). Der Hausmeister übernimmt u. a. auch die Rolle eines Boten zwischen den Küchen, um Gegenstände oder Informationen zwischen den Küchen auszutauschen. Auch kann er eine Küche durch einen Anbau vergrößern, wenn eine Küche zu klein geworden ist.

Ein verteiltes System besteht entsprechend aus mehreren solcher Häuser mit Küchen, wobei die Hausmeister der einzelnen Häuser z. B. über Telefon oder über hin- und herlaufende Boten untereinander kommunizieren können. Somit können Köche, die in unterschiedlichen Häusern arbeiten, Gegenstände oder Informationen austauschen, indem sie ihre jeweiligen Hausmeister damit beauftragen.

Diese Begriffe und ihre Beziehung sind in Bild 1.1 zusammenfassend dargestellt.

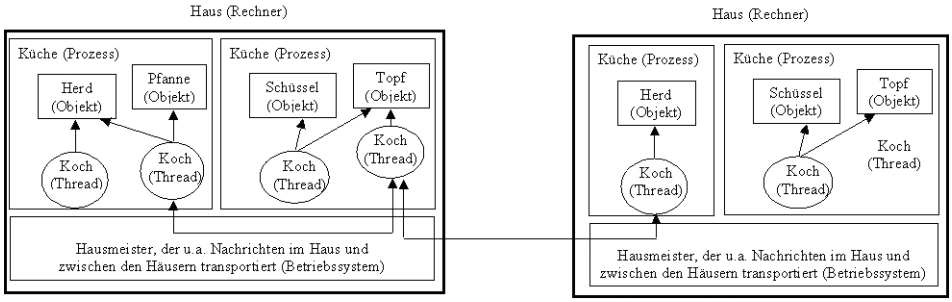


Bild 1.1 Häuser, Küchen, Köche und Hausmeister als Metapher für Rechner, Prozesse, Threads und Betriebssysteme

Am Beispiel der Programmiersprache Java und der Ausführung von Java-Programmen lässt sich diese Metapher nun auf die Welt der Informatik übertragen:

- Ein Programm (Kochrezept) ist in einer Datei abgelegt: als Quelltext in einer oder mehreren Java-Dateien und als übersetztes Programm (Byte-Code) in einer oder mehreren Class-Dateien.
- Zum Ausführen eines Programms mithilfe des Kommandos java wird eine JVM (Java Virtual Machine) gestartet. Bei jedem Erteilen des Java-Kommandos wird ein neuer Prozess

(Küche) erzeugt. Ein Prozess stellt im Wesentlichen einen Adressraum für den Programmcode und die Daten dar. Der Programmcode, der sich in einer oder mehreren Dateien befindet, wird in den Adressraum des Prozesses geladen. Es ist möglich, mehrere JVMs zu starten, sodass die entsprechenden Prozesse alle gleichzeitig existieren, wobei jeder Prozess seinen eigenen Adressraum besitzt.

- Jeder Prozess und damit auch jede JVM hat als Aktivitätsträger mindestens einen Thread (Koch). Neben den sogenannten Hintergrund-Threads, die z. B. für die Speicherbereinigung (Garbage Collection) zuständig sind, gibt es einen Thread, der die Main-Methode der im Java-Kommando angegebenen Klasse ausführt. Dieser Thread kann durch Aufruf entsprechender Methoden weitere Threads starten. Die Threads innerhalb desselben Prozesses können auf dieselben Objekte (Gegenstände in einer Küche wie Herd, Pfannen, Töpfe, Schüsseln usw.) lesend und schreibend zugreifen, nicht aber auf die Objekte, die sich in anderen Prozessen befinden.
- Das Betriebssystem (Hausmeister) verwaltet die Adressräume der Prozesse und teilt den Threads abwechselnd die vorhandenen Prozessoren bzw. den vorhandenen Kernen zu. Das Betriebssystem garantiert gemeinsam mit der Hardware, dass ein Prozess keinen Zugriff auf den Adressraum eines anderen Prozesses auf demselben Rechner besitzt. Damit sind die Prozesse eines Rechners voneinander isoliert. Zwei Prozesse auf unterschiedlichen Rechnern sind ebenfalls voneinander isoliert, da ein verteiltes System laut Definition ein lose gekoppeltes System ist, das keinen gemeinsamen Speicher hat.

Die Isolierung der Prozessadressräume kann durch Inanspruchnahme von Leistungen des Betriebssystems über Systemaufrufe in kontrollierter Weise durchbrochen werden. Damit können die Prozesse miteinander interagieren. Betriebssysteme bieten Dienste zur Synchronisation und Kommunikation zwischen Prozessen sowie zur gemeinsamen Nutzung von speziellen Speicherbereichen an. Ferner stellen Betriebssysteme Funktionen bereit, um über ein Rechnernetz Daten an Prozesse anderer Rechner zu senden oder eingetroffene Nachrichten entgegenzunehmen. Durch Systemaufrufe kann das Betriebssystem auch beauftragt werden, den Adressraum eines Prozesses zu vergrößern.

In diesem Buch geht es um zwei wesentliche Aspekte:

- Parallelität innerhalb eines Prozesses: Die Leserinnen und Leser sollen das Konzept der Parallelität innerhalb eines Prozesses aus Sicht einer Programmiererin bzw. eines Programmierers mit Java-Threads beherrschen lernen. Sie sollen erkennen, welche Probleme entstehen, wenn mehrere Threads auf dieselben Objekte zugreifen und wie diese Probleme gelöst werden können.
- Verteilung: Darüber hinaus zeigt das Buch, wie verteilte Anwendungen mit Java entwickelt werden. Wir unterscheiden dabei eigenständige Client-Server-Anwendungen und webbasierte Anwendungen. Bei eigenständigen Client-Server-Anwendungen entwickeln wir sowohl die Client- als auch die Server-Programme selbst. Client und Server kommunizieren dabei über die Socket-Schnittstelle, über RMI (Remote Method Invocation) oder indirekt über einen Vermittler. Bei webbasierten Anwendungen benutzen wir als Client einen Browser. Die Server-Seite besteht aus einem Web-Server, der durch selbst entwickelte Programme erweitert werden kann. Sowohl bei den eigenständigen Client-Server-Anwendungen als auch bei den webbasierten Anwendungen spielt die Parallelität insbesondere auf Server-Seite eine wichtige Rolle. Immer wichtiger wird die Nutzung von Cloud-Diensten durch verteilte Anwendungen. Auch dieses Thema wird behandelt.

Wir betrachten hier nicht gesondert die Parallelität, Interaktion und Synchronisation von Threads unterschiedlicher Prozesse desselben Rechners. Dies liegt vor allem daran, dass es hierzu keine speziellen Java-Klassen gibt. Dies bedeutet aber keine Einschränkung, denn alle in diesem Buch vorgestellten Kommunikationskonzepte zwischen dem Client- und Server-Prozess einer verteilten Anwendung können auch angewendet werden, wenn sich Client und Server auf demselben Rechner befinden. Das heißt: Bezüglich der Kommunikation zwischen Threads unterschiedlicher Prozesse unterscheiden wir nicht, ob sich die Prozesse auf demselben oder auf unterschiedlichen Rechnern befinden.

Die Synchronisations- und Kommunikationskonzepte, die anhand von Java-Threads innerhalb eines Prozesses vorgestellt werden, gibt es in ähnlicher Weise auch für das Zusammenspiel von Threads unterschiedlicher Prozesse auf einem Rechner. Wie schon erwähnt, gibt es zwar hierfür keine spezielle Java-Schnittstelle, aber die erlernten Konzepte wie Semaphore, Message Queues und Pipes bilden eine gute Grundlage für das Verständnis der Dienste, die ein Betriebssystem wie Linux zur Synchronisation und Kommunikation zwischen unterschiedlichen Prozessen anbietet.

2 Grundlegende Synchronisationskonzepte in Java

In diesem Kapitel geht es um die grundlegenden Synchronisationskonzepte in Java. Diese bestehen im Wesentlichen aus dem Schlüsselwort synchronized sowie den Methoden wait, notify und notifyAll der Klasse Object. Es wird erläutert, welche Wirkung synchronized, wait, notify und notifyAll haben und wie sie eingesetzt werden sollen. Außerdem spielt die Klasse Thread eine zentrale Rolle. Diese Klasse wird benötigt, um Threads zu erzeugen und zu starten.

2.1 Erzeugung und Start von Java-Threads

Wie schon im einleitenden Kapitel erläutert wurde, wird beim Start eines Java-Programms (z. B. mittels des Kommandos java) ein Prozess erzeugt, der u. a. einen Thread enthält, der die Main-Methode der angegebenen Klasse ausführt. Der Programmcode weiterer vom Anwendungsprogrammierer definierter Threads muss sich in Methoden namens *run* befinden:

```
public void run ()
{
    // Code, der in eigenem Thread ausgeführt wird
}
```

Es gibt zwei Möglichkeiten, in welcher Art von Klasse diese Run-Methode definiert wird.

2.1.1 Ableiten der Klasse Thread

Die erste Möglichkeit besteht darin, aus der Klasse *Thread*, die bereits eine leere Run-Methode besitzt, eine neue Klasse abzuleiten und darin die Run-Methode zu überschreiben. Die Klasse Thread ist (wie String) eine Klasse des Package java.lang und kann deshalb ohne Import-Anweisung in jedem Java-Programm verwendet werden. Hat man eine derartige Klasse definiert, so muss noch ein Objekt dieser Klasse erzeugt und dieses Objekt (das ja ein Thread ist, da es von Thread abgeleitet wurde) mit der *Start-Methode* gestartet werden. Das Programm in Listing 2.1 zeigt dies anhand eines Beispiels.

Listing 2.1

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Hallo Welt");
    }
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

An diesem ersten Programmbeispiel mag auf den ersten Blick verwirrend sein, dass in der Klasse MyThread zwar eine Run-Methode definiert wird, dass aber in der Main-Methode eine Methode namens *start* auf das Objekt der Klasse MyThread angewendet wird. Die Methode start ist in der Klasse Thread definiert und wird somit auf die Klasse MyThread vererbt.

```
public class Thread
{
    public void start () {...}
    ...
}
```

Natürlich könnte man statt start auch die Methode run auf das erzeugte Objekt anwenden. Der Benutzer würde keinen Unterschied zwischen den beiden Programmen feststellen können, denn in beiden Fällen wird „Hallo Welt“ ausgegeben. Allerdings ist der Ablauf in beiden Fällen deutlich verschieden: In der Metapher der Küchen und Köche passiert bei dem oben angegebenen Programm Folgendes: Der bereits vorhandene Koch, der nach dem Rezept der Main-Methode kocht, erzeugt einen neuen Koch und erweckt diesen mithilfe der Start-Methode zum Leben. Dieser neue Koch geht nach dem Rezept der entsprechenden Run-Methode vor und gibt „Hallo Welt“ aus. Würde dagegen der Aufruf der Start-Methode durch einen Aufruf der Run-Methode in obigem Programm ersetzt, so wäre dies ein gewöhnlicher Methodenaufruf, wie Sie das aus der bisherigen sequenziellen Programmierung bereits kennen. Die Ausgabe „Hallo Welt“ erfolgt also in diesem Fall durch den Thread, der die Main-Methode ausführt, und nicht durch einen neuen Thread. In der Metapher der Küchen und Köche könnte man einen Methodenaufruf so sehen wie einen Hinweis in einem Kochbuch, in dem in einem Rezept die Anweisung „Hefeteig zubereiten“ (s. Seite 456) steht. Derselbe Koch, der diese Anweisung liest, würde dann auf die Seite 456 blättern, die dort stehenden Anweisungen befolgen und anschließend zum ursprünglichen Rezept zurückkehren.

Dieses kleine, nur wenige Zeilen umfassende Beispielprogramm enthält noch ein weiteres Verständnisproblem für viele Neulinge: Warum muss ein Thread-Objekt (genauer: ein Objekt der aus Thread abgeleiteten Klasse MyThread) mit new erzeugt und warum muss dieses dann noch zusätzlich mit der Start-Methode gestartet werden? Diese Verständnisschwierigkeit kann beseitigt werden, indem man sich klar macht, dass es einen Unterschied zwischen einem Thread-Objekt und dem eigentlichen Thread im Sinne einer selbst-

ständig ablaufenden Aktivität gibt. In unserer Küchen-Köche-Metapher entspricht das Thread-Objekt dem Körper eines Kochs. Ein solcher Körper wird mit new erzeugt. Man kann bei diesem Objekt wie bei anderen Objekten üblich Attribute lesen und verändern, also z. B. Name, Personalnummer und Schuhgröße des Kochs. Dieses Objekt ist aber leblos wie andere Objekte bei der sequenziellen Programmierung auch. Erst durch Aufruf der Start-Methode wird dem Koch der Odem eingehaucht; er beginnt zu atmen und eigenständig gemäß seines Run-Rezepts zu handeln. Dieses Leben des Kochs ist als Objekt im Programm nicht repräsentiert, sondern lediglich der Körper des Kochs. Das Leben des Kochs ist beim Ablauf des Programms durch die vorhandene Aktivität zu erkennen.

Wie im richtigen Leben kann auf ein Thread-Objekt nur ein einziges Mal die Start-Methode angewendet werden. Wenn mehrere gleichartige Threads gestartet werden sollen, dann müssen entsprechend viele Thread-Objekte erzeugt werden (s. Abschnitt 2.1.3).

Ist das Run-Rezept eines Kochs abgehandelt (d. h. ist die Run-Methode zu Ende), so stirbt dieser Koch wieder (der Thread ist als Aktivität nicht mehr vorhanden). Damit muss aber der Körper des Kochs nicht auch verschwinden, sondern dieser kann weiter existieren (falls es noch Referenzen auf das entsprechende Thread-Objekt gibt, ist dieses Objekt noch vorhanden; die verbleibenden Threads können weitere Methoden auf dieses Objekt anwenden).

2.1.2 Implementieren der Schnittstelle Runnable

Falls sich im Rahmen eines größeren Programms die Run-Methode in einer Klasse befinden soll, die bereits aus einer anderen Klasse abgeleitet ist, so kann diese Klasse nicht auch zusätzlich aus Thread abgeleitet werden, da es in Java keine Mehrfachvererbung für Klassen gibt. Als Ersatz für die Mehrfachvererbung existieren in Java Schnittstellen (Interfaces). Es gibt eine Schnittstelle namens *Runnable* (wie die Klasse Thread im Package java.lang), die nur die schon oben vorgestellte Run-Methode enthält.

```
public interface Runnable
{
    public void run();
}
```

Will man nun die Run-Methode in einer nicht aus Thread abgeleiteten Klasse definieren, so sollte diese Klasse stattdessen die Schnittstelle Runnable implementieren. Wenn ein Objekt einer solchen Klasse, die diese Schnittstelle implementiert, dem Thread-Konstruktor als Parameter übergeben wird, dann wird die Run-Methode dieses Objekts nach dem Starten des Threads ausgeführt. Das Programm in Listing 2.2 zeigt diese Vorgehensweise anhand eines Beispiels.

Listing 2.2

```
public class SomethingToRun implements Runnable
{
    public void run()
    {
        System.out.println("Hallo Welt");
    }
```

```
    public static void main(String[] args)
    {
        SomethingToRun runner = new SomethingToRun();
        Thread t = new Thread(runner);
        t.start();
    }
}
```

Voraussetzung für die korrekte Übersetzung beider Beispielprogramme ist, dass die Klasse Thread u. a. folgende Konstruktoren besitzen muss:

```
public class Thread
{
    public Thread() {...}
    public Thread(Runnable r) {...}
   ...
}
```

Der zweite Konstruktor ist offenbar für das zweite Beispiel nötig. Die Nutzung des ersten Konstruktors im ersten Beispiel ist weniger offensichtlich. Da in der Klasse MyThread kein Konstruktor definiert wurde, ist automatisch der folgende Standardkonstruktor vorhanden:

```
public MyThread()
{
    super();
}
```

Der Super-Aufruf bezieht sich auf den parameterlosen Konstruktor der Basisklasse Thread. Einen solchen muss es geben, damit das Programm übersetzbar ist.

Auch für das zweite Beispiel gilt die Unterscheidung zwischen dem Thread-Objekt und dem eigentlichen Thread. Deshalb muss auch hier nach der Erzeugung des Thread-Objekts der eigentliche Thread noch gestartet werden.

Auch wenn wie oben beschrieben ein Thread nur einmal gestartet werden kann, kann hier dennoch dasselbe Runnable-Objekt mehrmals als Parameter an Thread-Konstruktoren übergeben werden. Es wird ja jedes Mal ein neues Thread-Objekt erzeugt, das dann nur einmal gestartet wird. Unter Umständen kann dies aber zu Synchronisationsproblemen führen (s. Abschnitt 2.2 und Abschnitt 2.3).

Seit Java 8 gibt es sogenannte *Lambda-Ausdrücke*. Die Definition der Klasse SomethingToRun in Listing 2.2, welche die Schnittstelle Runnable implementiert, sowie das Erzeugen eines Objekts dieser Klasse kann mit einem Lambda-Ausdruck durch eine einzige Anweisung ersetzt werden:

```
Runnable runner = () -> System.out.println("Hallo Welt");
```

Wenn man das Runnable-Objekt, das man dem Konstruktor von Thread übergibt, in keine lokale Variable speichern möchte, dann kann man die Thread-Erzeugung noch kürzer auch so schreiben:

```
Thread t = new Thread(() -> System.out.println("Hallo Welt"));
```

Und wenn man auf die lokale Thread-Variable t auch noch verzichten möchte, dann schrumpft der Inhalt der Main-Methode auf diese eine Zeile zusammen:

```
new Thread(() -> System.out.println("Hallo Welt")).start();
```

Lambda-Ausdrücke können im Programmcode immer dort angegeben werden, wo ein Objekt vom Typ einer sogenannten *funktionalen Schnittstelle* erwartet wird. Eine funktionale Schnittstelle (Functional Interface) ist eine Schnittstelle mit einer einzigen Methode (genauer müsste man sagen: eine Schnittstelle mit einer einzigen *abstrakten* Methode, denn seit Java 8 können Schnittstellen auch nicht-abstrakte Methoden, sogenannte Default-Methoden, besitzen, für die in der Schnittstelle eine Implementierung angegeben ist). Die Schnittstelle Runnable ist ganz offensichtlich eine funktionale Schnittstelle. Also kann auf der rechten Seite einer Zuweisung an eine Runnable-Variable (Runnable runner = ...) oder als Parameterwert eines Thread-Konstruktors mit Runnable-Parameter ein Lambda-Ausdruck eingesetzt werden.

Allgemein hat ein Lambda-Ausdruck folgende Form:

```
Parameterliste -> Code
```

Der Name der implementierten Methode der Schnittstelle muss (und darf auch) bei einem Lambda-Ausdruck nicht angegeben werden; der Typ des Lambda-Ausdrucks ist nämlich eine funktionale Schnittstelle mit einer einzigen abstrakten Methode, und genau diese Methode wird implementiert. Für die Parameter müssen Bezeichner und optional der jeweilige Typ angegeben werden. Sie können im Code-Teil verwendet werden. Betrachten wir dazu z. B. folgende funktionale Schnittstelle I1:

```
public interface I1
{
    public void m(String s, boolean b);
}
```

Nun könnte man beispielsweise schreiben:

```
I1 i11 = (String s, boolean b) -> System.out.println(s + ", " + b);
```

Oder auch kürzer durch Weglassen der Parametertypen, die sich wie der Methodenname eindeutig aus der funktionalen Schnittstelle I1 herleiten lassen:

```
I1 i12 = (s, b) -> System.out.println(s + ", " + b);
```

Mischformen (also ein Parameter mit Typangabe und ein anderer Parameter ohne Typangabe im selben Lambda-Ausdruck) sind nicht möglich.

Da die Methode run der Schnittstelle Runnable parameterlos ist, musste im obigen Thread-Beispiel der Lambda-Ausdruck mit einer leeren Klammer beginnen. Wenn die zu implementierende Methode genau einen Parameter besitzt, dann können im Lambda-Ausdruck die Klammern um den Parameter weggelassen werden, wenn man auch auf die Angabe des Typs verzichtet.

Der Codeteil bestand in den bisherigen Beispielen immer aus genau einer Anweisung. Im Allgemeinen können es mehrere Anweisungen sein, die dann aber als Java-Block in geschweiften Klammern zusammengefasst sein müssen:

```
I1 i13 = (s, b) -> {System.out.println(s); System.out.println(b);};
```

Nun muss auch jede Java-Anweisung wie allgemein üblich durch ein Semikolon abgeschlossen werden. Ich betrachte es als schlechten Stil, wenn in einem Lambda-Ausdruck sehr viel Code enthalten ist. Im Idealfall besteht nach meiner Auffassung der Code-Teil nur aus einer einzigen Anweisung, wenn auch Java-Code beliebiger Länge erlaubt ist, der z. B. wiederum Lambda-Ausdrücke enthalten darf.

Wenn die Methode der funktionalen Schnittstelle nicht void als Rückgabetyp hat, dann kann der Codeteil auch lediglich aus einem Ausdruck für den zurückgegebenen Wert bestehen. Wir verwenden zur Erläuterung die funktionale Schnittstelle I2 mit einer Methode, dessen Rückgabetyp int ist:

```
public interface I2
{
    public int op(int arg1, int arg2);
}
```

Jetzt könnten wir zum Beispiel schreiben:

```
I2 i21 = (i, j) -> {return i+j;};
```

Oder kürzer nur durch Angabe eines Ausdrucks für den zurückgegebenen Wert als Code:

```
I2 i22 = (i, j) -> i+j;
```

Damit soll es mit den Erläuterungen zu Lambda-Ausdrücken genug sein. Wir wenden uns jetzt wieder unserem eigentlichen Thema, den Threads, zu.

2.1.3 Einige Beispiele

Um das bisher Gelernte zum Thema Threads etwas zu vertiefen, betrachten wir das Beispielprogramm aus Listing 2.3:

Listing 2.3

```
public class Loop1 extends Thread
{
    private String myName;

    public Loop1(String name)
    {
        myName = name;
    }

    public void run()
    {
        for(int i = 1; i <= 100; i++)
        {
            System.out.println(myName + " (" + i + ")");
        }
    }

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

```
        Loop1 t1 = new Loop1("Thread 1");
        Loop1 t2 = new Loop1("Thread 2");
        Loop1 t3 = new Loop1("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

In diesem Beispiel werden drei zusätzliche Threads gestartet. Die dazugehörigen Thread-Objekte gehören alle derselben Klasse an, sodass die Threads alle dieselbe Run-Methode ausführen. Bei der Ausgabe innerhalb der For-Schleife der Run-Methode wird auf das Attribut name des dazugehörigen Thread-Objekts und auf die lokale Variable i zugegriffen. Für alle Threads gibt es jeweils eigene Exemplare sowohl von name als auch von i. Für das Attribut name ist dies deshalb so, weil jeder Thread zu genau einem Thread-Objekt gehört und die Run-Methode jeweils auf das Attribut des dazugehörigen Thread-Objekts zugreift. Da in jedem Thread ein Aufruf der Methode run stattfindet, gibt es entsprechend auch für jeden Methodenaufruf gesonderte Exemplare der lokalen Variablen i wie bei rein sequenziellen Programmen auch.

Nach dem Übersetzen dieses Programms ergibt sich bei der Ausführung des Programms auf meinem Rechner folgende Ausgabe (... steht für Zeilen, die aus Gründen des Platzsparens ausgelassen wurden):

```
Thread 1 (1)
Thread 1 (2)
...
Thread 1 (45)
Thread 1 (46)
Thread 2 (1)
Thread 3 (1)
Thread 2 (2)
Thread 3 (2)
Thread 2 (3)
Thread 3 (3)
Thread 2 (4)
Thread 1 (47)
Thread 2 (5)
Thread 1 (48)
Thread 2 (6)
Thread 1 (49)
Thread 3 (4)
Thread 1 (50)
Thread 3 (5)
Thread 1 (51)
Thread 3 (6)
Thread 1 (52)
Thread 3 (7)
Thread 2 (7)
Thread 3 (8)
Thread 2 (8)
Thread 3 (9)
Thread 2 (9)
Thread 1 (53)
```

```
Thread 2 (10)
Thread 1 (54)
Thread 2 (11)
Thread 1 (55)
Thread 2 (12)
Thread 1 (56)
Thread 3 (10)
Thread 2 (13)
Thread 1 (57)
Thread 3 (11)
Thread 2 (14)
...
```

Die Threads laufen scheinbar oder echt gleichzeitig. Welcher Fall vorliegt, kann anhand der Ausgabe nicht eindeutig unterschieden werden. Falls der Ablauf nur scheinbar gleichzeitig ist, dann wird dies durch das automatische Umschalten zwischen den verschiedenen Threads realisiert, sodass sich der Eindruck eines parallelen Ablaufs ergibt. Dabei ist die Anzahl der Ausgaben, die von einem Thread direkt hintereinander erscheinen, bevor eine Ausgabe von einem anderen Thread kommt, nicht immer gleich groß. Wie Sie anhand der Ausgabe sehen können, gibt es zu Beginn 46 Mal Ausgaben des ersten Threads. Danach erscheint die erste Ausgabe des zweiten Threads. Diesem Thread ist aber nur eine einzige Ausgabe am Stück gegönnt. Danach ist der dritte Thread an der Reihe, und zwar auch nur mit einer einzigen Runde. Dieses Verhalten ist schematisch in Bild 2.1 illustriert. Dabei ist die horizontale Achse die Zeitachse. Es werden unsere drei Threads sowie drei beliebige weitere Threads aus einem oder mehreren anderen Prozessen gezeigt. Damit Sie eine Vorstellung von der Anzahl der Prozesse und Threads auf einem Rechner bekommen, gebe ich hier wieder, was mir der Task Manager auf meinem Windows-System momentan anzeigt: Dort gibt es mehr als 200 Prozesse und mehr als 2000 Threads. Die gezeigten sechs Threads sind also eine drastische Vereinfachung. Wir gehen außerdem von nur zwei Prozessoren aus. Man sieht, welche Threads zu einem bestimmten Zeitpunkt von welchen Prozessoren ausgeführt werden. Auch ist zu erkennen, dass die Zeitintervalle, in denen die Threads ununterbrochen laufen können, unterschiedlich lang sein können.

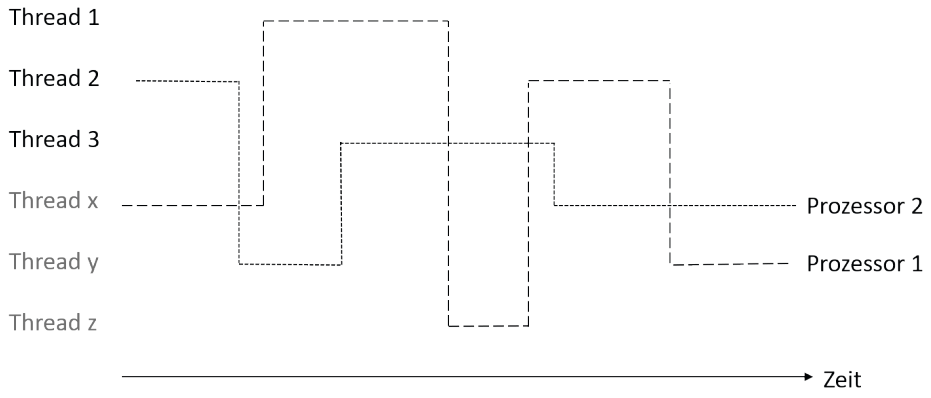


Bild 2.1 Ausführungsintervalle von drei Threads

Wenn Sie Bild 2.1 genauer betrachten, werden Sie erkennen, dass es Zeitpunkte gibt, an denen zwei unserer Threads echt parallel laufen (Thread 1 und Thread 3 und kurzzeitig auch Thread 2 und Thread 3), an denen nur einer unserer Threads oder auch keiner unserer Threads läuft (am Ende). Außerdem können Sie sehen, dass ein Thread nicht immer auf demselben Prozessor laufen muss. So wird Thread 2 zu Beginn der Betrachtung von Prozessor 2 ausgeführt und später von Prozessor 1.

Der Ablauf und entsprechend auch die Ausgabe dieses Programms müssen nicht bei jeder Ausführung gleich sein. Bei wiederholter Ausführung des Programms können sich unterschiedliche Ausgaben ergeben. Auch kann die Ausgabe vom eingesetzten Betriebssystem (z. B. Windows oder Linux) und der Hardware (z. B. Anzahl der Prozessoren bzw. Anzahl der Prozessorkerne) abhängen. Falls Sie dieses Beispiel ausprobieren und Sie finden Ihre Ausgabe zu langweilig (erst alle 100 Ausgaben des ersten Threads, dann alle des zweiten und schließlich alle des dritten), dann sollten Sie die Anzahl der Schleifendurchläufe so lange erhöhen (z. B. von 100 auf 1000), bis Sie eine Vermischung der Ausgaben der unterschiedlichen Threads sehen können.

Doch nun zurück zu unserem Beispielprogramm von Listing 2.3. Das Attribut name ist nicht nötig, da die Klasse Thread bereits ein solches String-Attribut für den Namen des Threads besitzt. Der Wert des Namen-Attributs kann im Konstruktor als Argument angegeben werden und später durch die Methode *setName* verändert werden. Mithilfe der Methode *getName* kann der Name gelesen werden. Wird der Name eines Threads nicht explizit gesetzt, so wird ein Standardname gewählt. Neben den schon bekannten Konstruktoren der Klasse Thread ohne Argument und mit einem Runnable-Argument gibt es Konstruktoren mit einem zusätzlichen Namensargument. Damit kennen wir nun vier Konstruktoren der Klasse Thread. Zusätzlich werden die neuen Methoden setName und getName gezeigt:

```
public class Thread
{
    public Thread() {...}
    public Thread(Runnable r) {...}
    public Thread(String name) {...}
    public Thread(Runnable r, String name) {...}
    ...
    public final void setName(String name) {...}
    public final String getName() {...}
    ...
}
```

Im folgenden Beispiel (Listing 2.4) wird das Namensattribut der Klasse Thread statt eines eigenen Attributs verwendet. Beachten Sie, dass in der Ausgabe von System.out.println der Name nun mit getName beschafft werden muss.

Listing 2.4

```
public class Loop2 extends Thread
{
    public Loop2(String name)
    {
        super(name);
    }

    public void run()
```

```
    {
        for(int i = 1; i <= 100; i++)
        {
            System.out.println(getName() + " (" + i + ")");
        }
    }

    public static void main(String[] args)
    {
        Loop2 t1 = new Loop2("Thread 1");
        Loop2 t2 = new Loop2("Thread 2");
        Loop2 t3 = new Loop2("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Das nächste Beispiel (Listing 2.5) variiert das vorige Beispiel nochmals. Die For-Schleife der Run-Methode, die jetzt nur noch zehnmal durchlaufen wird, enthält einen zusätzlichen Sleep-Aufruf.

Listing 2.5

```
public class Loop3 extends Thread
{
    public Loop3(String name)
    {
        super(name);
    }

    public void run()
    {
        for(int i = 1; i <= 10; i++)
        {
            System.out.println(getName() + " (" + i + ")");
            try
            {
                sleep(100);
            }
            catch(InterruptedException e)
            {
            }
        }
    }

    public static void main(String[] args)
    {
        Loop3 t1 = new Loop3("Thread 1");
        Loop3 t2 = new Loop3("Thread 2");
        Loop3 t3 = new Loop3("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

Die Methode *sleep* ist eine Static-Methode der Klasse Thread und kann deshalb ohne Angaben eines Objekts oder einer Klasse in der Run-Methode der Klasse Loop3, die ja aus Thread abgeleitet ist, aufgerufen werden:

```
public class Thread
{
    public static void sleep(long millis) throws InterruptedException {...}
    public static void sleep(long millis, int nanos)
           throws InterruptedException {...}
    ...
}
```

Der Aufruf von sleep bewirkt, dass der aufrufende Thread die als Argument angegebene Zahl von Millisekunden „schläft". In einer überladenen Variante der Methode sleep kann diese Zeit feiner in Milli- und Nanosekunden angegeben werden, wobei die Angabe der Nanosekunden von den meisten Implementierungen ignoriert wird. Das „Schlafen" wird dabei so realisiert, dass es keine Rechenzeit beansprucht. Das heißt, die Sleep-Methode ist nicht so realisiert, dass in einer Schleife immer wieder abgefragt wird, ob die angegebene Zeit vergangen ist, sondern der Thread wird für die angegebene Zeit bei der Thread-Umschaltung vom Betriebssystem nicht mehr berücksichtigt und verbraucht in dieser Phase keine Rechenzeit.

Die Sleep-Methoden können eine Ausnahme vom Typ *InterruptedException* werfen (wodurch eine solche Ausnahme ausgelöst werden kann, wird in Abschnitt 2.4 besprochen). Aus diesem Grund muss im Allgemeinen der Aufruf von sleep in einem Try-Catch-Block erfolgen oder die Methode, die diesen Aufruf enthält, muss so gekennzeichnet sein, dass sie selbst Ausnahmen dieser Art werfen kann (z. B. so wie sleep auch mit „throws InterruptedException"). In obigem Beispielprogramm ist nur die erste Alternative möglich (Try-Catch-Block). Als Argument von sleep wurde 100 angegeben. Der aufrufende Thread schläft somit 100 Millisekunden oder 0,1 Sekunden. Dies ist zwar für uns Menschen eine kurze Zeitspanne, die uns für eine echte Erholung nicht reichen würde. Für den Rechner ist dies aber eine sehr lange Zeit. Die Ausführung der Ausgabeanweisung System.out.println dauert wesentlich weniger lang. Wenn also ein Thread sleep aufruft, wird auf einen anderen Thread umgeschaltet. Dieser kann dann seine Ausgabe machen und beginnt ebenfalls zu schlafen. Da der erste Thread zu diesem Zeitpunkt gerade erst angefangen hat zu schlafen und deshalb immer noch schläft, kann jetzt nur noch auf den dritten Thread geschaltet werden. Dieser führt ebenfalls seine Ausgabe durch und ruft sleep auf. Dies läuft alles so schnell ab, dass von der Schlafenszeit des ersten Threads noch fast nichts vergangen ist. Somit schlafen alle Threads für einen gewissen Zeitraum. Danach wacht der erste Thread auf. Es wird auf ihn umgeschaltet. Er führt seine Ausgabe durch und schläft wieder. Inzwischen ist aber der zweite Thread aufgewacht. Sie können sicher selber den weiteren Ablauf gedanklich fortsetzen. Die Ausgabe dieses Programms ist deshalb auch so wie erwartet: Die Ausgaben der drei Threads wechseln sich regelmäßig ab:

```
Thread 1 (1)
Thread 2 (1)
Thread 3 (1)
Thread 1 (2)
Thread 2 (2)
Thread 3 (2)
...
```

```
Thread 1 (9)
Thread 2 (9)
Thread 3 (9)
Thread 1 (10)
Thread 2 (10)
Thread 3 (10)
```

Allerdings sollte man sich darauf nicht verlassen. Bei mehrfacher Ausführung des Programms kann man z. B. auch einmal folgende Ausgabe sehen:

```
Thread 1 (1)
Thread 2 (1)
Thread 3 (1)
Thread 1 (2)
Thread 2 (2)
Thread 3 (2)
...
Thread 1 (9)
Thread 3 (9)
Thread 2 (9)


Thread 1 (10)
Thread 3 (10)
Thread 2 (10)
```

Zunächst erfolgt acht Mal die Ausgabe in der Reihenfolge 1-2-3. Ab dem neunten Mal ist die Reihenfolge 1-3-2. Eine solche Reihenfolgeänderung kann erfolgen, weil wegen der sehr kurzen Ausführungszeit der Ausgabeanweisung alle Threads fast gleichzeitig einschlafen. Auf vielen Rechnersystemen ist nun aber die zeitliche Auflösung nicht sehr fein. So kann das Aufwachen nur zu bestimmten Zeitpunkten erfolgen (z. B. in einem 10 Millisekunden-Raster). Aus dem täglichen Leben ist Ihnen eine solche Sachlage vertraut: So können Sie einen Wecker nur so einstellen, dass er Sie zur vollen Minute weckt, also z. B. um 7:30 oder 7:31 Uhr, aber eben nicht z. B. 16 Sekunden nach 7:30 Uhr. Für unser Beispielprogramm bedeutet dies, dass alle Threads eventuell genau zur selben Zeit geweckt werden. Die Reihenfolge, in der auf die Threads umgeschaltet wird, ist aber nicht fest vorgegeben. Aus diesem Grund kann es wie gesehen zu einer Änderung der Reihenfolge kommen.

Aus diesem Beispiel sollte eine wichtige Lehre gezogen werden: Wenn man eine bestimmte Ausführungsreihenfolge zwischen Threads erzwingen möchte, dann ist man sehr schlecht beraten, wenn man dies mit Sleep-Methoden realisiert. Die Wahl der Schlafenszeit ist hierbei nämlich außerordentlich kritisch. Wenn eine Zeit gewählt wird, die beim Testen in allen Fällen funktioniert hat, so kann es unter gewissen Bedingungen (z. B. nach der Installation eines neuen Betriebssystems oder der Portierung des Programms auf einen anderen Rechner oder wenn der ausführende Rechner durch andere Prozesse stärker belastet ist als zuvor) dazu kommen, dass die gewünschte Reihenfolge nicht mehr eingehalten wird. Wählt man auf der anderen Seite eine sehr große Zeit, die in jedem Fall ausreicht, so ist dies in den meisten Fällen ineffizient, weil ein Thread dann viel zu lange schläft.

Wenn Sie an der Lösung dieses Problems interessiert sind, so lesen Sie weiter. Ein großer Teil dieses Buchs beschäftigt sich u. a. mit der Problematik, gewünschte Reihenfolgen zwischen Threads zu erzwingen.

2.1.4 Parallele Abläufe

Grundsätzlich gibt es bei der Ausführung eines parallelen Programms immer mehrere mögliche Ablaufreihenfolgen. Dies sei an einem Beispiel erläutert: Der Thread 1 führt die Aktionen A1 und danach A2 aus, der Thread 2 B1 und danach B2. Dann sind die folgenden Abläufe möglich: A1-A2-B1-B2, A1-B1-A2-B2, B1-A1-A2-B2, A1-B1-B2-A2, B1-A1-B2-A2 und B1-B2-A1-A2. Bei zwei Threads, von denen jeder nur zwei Aktionen durchführt, gibt es also bereits sechs Möglichkeiten. Es sind natürlich nicht alle Permutationen von A1, A2, B1 und B2 möglich (das wären sogar 4! = 24), da A1 immer vor A2 und B1 immer vor B2 kommen muss. Ein Beispiel mit nur zwei Aktionen und nur zwei Threads ist sehr einfach. Ich habe einmal für einige Beispiele die Anzahl der möglichen parallelen Abläufe ausgerechnet: Bei zwei Threads, die jeweils fünf Aktionen ausführen, ergeben sich schon 252 mögliche Ausführungsreihenfolgen. Sind es zehn Aktionen pro Thread bei weiterhin zwei Threads, so steigt diese Zahl auf 184 756. Sind drei Threads im Spiel, so ergeben die Berechnungen wesentlich größere Zahlen. Bei nur zwei Aktionen pro Thread ergeben sich schon 90 Möglichkeiten. Für drei Threads, die jeweils acht Aktionen durchführen, habe ich einen Wert von mehr als 9 Milliarden errechnet, und für drei Threads mit je zehn Aktionen komme ich auf ca. 5,5 Billionen. Wenn wir von dem Programm aus Listing 2.3 als Aktionen nur die jeweils 100 Ausgaben pro Thread betrachten, ergibt sich eine astronomisch große Zahl für die unterschiedlichen Ausgaben, die man bei einer Programmausführung zu sehen bekommen könnte. Sicher sind davon nicht alle gleich wahrscheinlich, aber alle sind theoretisch möglich. Programme sind nur dann korrekt, wenn sie sich bei allen möglichen Ausführungsreihenfolgen richtig verhalten. Nun haben wir als Aktionen unserer Threads nur die 100 Ausgaben pro Thread betrachtet. Tatsächlich führen die Threads viel mehr Aktionen aus, also z. B. auch den Vergleich von i mit 100 und die Erhöhung von i. Und selbstverständlich ist Ihnen klar, wie klein die Methode run ist und wie viel mehr Aktionen wohl in Abläufen in der Praxis üblich sind.

Die im Allgemeinen extrem große Zahl möglicher Abläufe machen das Nachdenken über Parallelität für uns Menschen so schwer. In manchen Fällen ist es relativ gleichgültig, in welcher Reihenfolge die einzelnen Aktionen durchgeführt werden. Dies gilt für unsere bisher betrachteten Beispielprogramme. In anderen Fällen, zu denen wir in Kürze kommen, kann es aber passieren, dass gewisse Reihenfolgen ein erwünschtes Verhalten zeigen, andere Reihenfolgen aber einen Fehler verursachen. Da es in der Regel sehr viele Reihenfolgen gibt und wir Menschen uns eher an denen orientieren, die ein erwünschtes Verhalten zeigen, sind Fehler in parallelen Programmen für uns Menschen oft nur sehr schwer oder gar nicht zu erkennen. Wenn die vielen möglichen Abläufe zu keinen Problemen führen, muss man sich um die große Anzahl an Ausführungsmöglichkeiten keine Gedanken machen. In vielen Fällen ist dies aber nicht so. Durch Synchronisation sollen die Ausführungsreihenfolgen, die zu Problemen führen können, ausgeschlossen werden. Darum geht es im weiteren Verlauf dieses Kapitels.

Bei der Betrachtung möglicher Reihenfolgen von Aktionen wird übrigens echte Parallelität nie berücksichtigt, wie Sie gesehen haben und was Sie vielleicht wundern mag. Wenn zwei Aktionen A und B parallel laufen können, dann entspricht dies in der Betrachtung einfach den beiden möglichen Abläufen A-B und B-A. Wenn wir uns im restlichen Teil dieses Buchs Abläufe anschauen, dann immer so, dass wir irgendeine Reihenfolge von hintereinander ausgeführten Aktionen betrachten. Deshalb auch gehen die Überlegungen immer davon

aus, dass es nur einen einzigen Prozessor gibt, der eine gewisse Zeit lang Aktionen eines Threads ausführt, dass dann auf einen anderen Thread umgeschaltet wird und folglich Aktionen dieses anderen Threads abgearbeitet werden.

2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte

In den bisherigen Beispielen arbeiten die einzelnen Threads weitgehend unabhängig voneinander, da jeder Thread seine eigenen Attribute und lokalen Variablen besitzt. In vielen Anwendungen werden Threads jedoch eingesetzt, um in kooperativer Weise an einer gemeinsamen Aufgabe zu arbeiten. In solchen Fällen ist immer auch der Zugriff auf gemeinsame Daten (in der Regel in einem Objekt gekapselt) von mehreren Threads aus nötig.

Im folgenden Beispiel wird in stark vereinfachter Form ein Bankbetrieb programmiert. Eine Bank verwaltet mehrere Konten (Account). Für jede Angestellte der Bank (Clerk) wird ein Thread realisiert. Diese Threads führen Buchungen auf den Konten durch. Dabei soll von jedem Thread aus der Zugriff auf jedes Konto möglich sein.

Die von mehreren Threads aus gemeinsam genutzten Objekte können nicht als Argumente der Run-Methode übergeben werden, da diese Methode keine Argumente besitzt und auch nie vom Anwendungsprogramm explizit aufgerufen wird, sodass solche Argumente übergeben werden könnten. Es bieten sich mehrere Alternativen an. So können z. B. Referenzen auf die von mehreren Threads benutzten Objekte in den Klassen, in denen sich die Run-Methoden befinden, als Attribute geführt werden. Die Werte dieser Attribute müssen dann „von außen“ mithilfe bestimmter Methoden gesetzt oder bereits als Argumente im Konstruktor übergeben werden. In fast allen Beispielen dieses Buches werden Referenzen auf gemeinsam benutzte Objekte als Argumente von Konstruktoren übergeben. Dies kann auch im folgenden Beispiel (Listing 2.6) gesehen werden.

Listing 2.6

```
class Account //Konto
{
    private float balance; //Kontostand

    public void setBalance(float balance)
    {
        this.balance = balance;
    }

    public float getBalance()
    {
        return balance;
    }
}

class Bank
{
```

```
    private final static int NUMBER_OF_ACCOUNTS = 100;
    private Account[] account;

    public Bank()
    {
        account = new Account[NUMBER_OF_ACCOUNTS];
        for(int i = 0; i < account.length; i++)
        {
            account[i] = new Account();
        }
    }

    public void transferMoney(int accountNumber, float amount)
    {
        float oldBalance = account[accountNumber].getBalance();
        float newBalance = oldBalance + amount;
        account[accountNumber].setBalance(newBalance);
    }
}

class Clerk extends Thread
{
    private Bank bank;

    public Clerk(String name, Bank bank)
    {
        super(name);
        this.bank = bank;
        start();
    }

    public void run()
    {
        for(int i = 0; i < 10000; i++)
        {
            /* Kontonummer einlesen; simuliert durch Wahl einer Zufallszahl
               zwischen 0 und 99 */
            int accountNumber = (int)(Math.random()*100);

            /* Überweisungsbetrag einlesen; simuliert durch Wahl einer
               Zufallszahl zwischen -500 und +499 */
            float amount = (int)(Math.random()*1000) - 500;

            bank.transferMoney(accountNumber, amount);
          }
    }
}

public class Banking
{
    public static void main(String[] args)
    {
        Bank myBank = new Bank();
        new Clerk("Andrea Müller", myBank);
        new Clerk("Petra Schmitt", myBank);
    }
}
```

Die Klasse Account repräsentiert ein Konto mit einem Attribut für den aktuellen Kontostand und Methoden zum Abfragen und Setzen des Kontostands. Die Klasse Bank hat als Attribut ein Feld von Referenzen auf Konto-Objekte. Dieses Feld sowie die Konto-Objekte selbst werden im Konstruktor der Klasse Bank erzeugt. Mit der Methode transferMoney kann auf einem bestimmten Konto der Bank ein bestimmter Betrag gebucht werden. Dabei erhöht sich der Kontostand um diesen Betrag, falls der Betrag positiv ist, andernfalls erniedrigt sich der Kontostand entsprechend. Die Klasse Clerk repräsentiert eine Bankangestellte und ist aus Thread abgeleitet. Da es möglich sein soll, dass mehrere Angestellte für dieselbe Bank arbeiten, wird eine Referenz auf die Bank neben dem Namen der Angestellten, der der Name des Threads wird, im Konstruktor übergeben. In der Run-Methode werden sehr viele Buchungen auf unterschiedliche Konten der Bank, für die die Angestellte arbeitet, ausgeführt. In der Klasse Banking befindet sich die Main-Methode. Darin werden eine Bank sowie zwei Bankangestellte, die beide für diese Bank arbeiten, erzeugt. Beachten Sie eine kleine Variation gegenüber unseren vorigen Beispielen: Im Konstruktor der Klasse Clerk befindet sich der Aufruf der Thread-Methode start. Somit muss also in der Main-Methode nur das entsprechende Clerk-Objekt erzeugt werden. Der Thread läuft dann automatisch los; in main muss keine Start-Methode mehr explizit aufgerufen werden. Manche lehnen ein solches Programmkonstrukt ab. Ich persönlich halte es für vertretbar, falls der Aufruf von start die letzte Anweisung im Konstruktor ist und somit die Initialisierung des Objekts bereits vollständig erfolgt ist.

Als Alternative für die gemeinsame Nutzung eines Objekts durch mehrere Threads wäre es auch möglich gewesen, das Attribut Bank in der Klasse Clerk statisch zu machen, es einmalig zu initialisieren und auf die Übergabe des Bank-Objekts im Konstruktor von Clerk zu verzichten. Diese Variante hat aber den Nachteil, dass damit immer alle Bankangestellten eines Prozesses mit derselben Bank arbeiten müssen. In unserer Lösung oben könnte auch ein Main-Programm geschrieben werden, in dem mehrere Bankangestellte erzeugt werden, die in unterschiedlichen Banken arbeiten. Deshalb wird die Static-Variante in diesem Buch nicht verwendet.

So weit, so gut. Allerdings steckt in diesem scheinbar einfachen Programm ein großes und grundsätzliches Problem, das immer dann vorkommt, wenn mehrere Threads auf gemeinsamen Objekten arbeiten und deren Zustände lesen und verändern. Betrachten wir dazu folgende Situation: Die Angestellte Andrea Müller möchte 100 € vom Konto 47 abbuchen. Es wird also die Methode transferMoney mit den Argumenten 47 und −100 aufgerufen. In der Methode transferMoney wird nun in der ersten Anweisung der aktuelle Kontostand des Kontos 47 in die lokale Variable oldBalance gespeichert. Nehmen wir an, dieser Kontostand sei 0. Nehmen wir nun weiter an, dass anschließend Aktionen des anderen Threads ausgeführt werden (dies ist ja eine gültige mögliche Ausführungsreihenfolge). Petra Schmitt führt nun mehrere Buchungen durch, u. a. sollen 200 € dem Konto 47 gutgeschrieben werden. Es wird also vom Thread mit dem Namen „Petra Schmitt“ die TransferMoney-Methode der Klasse Bank mit den Argumenten 47 und 200 ausgeführt. Dort wird nun zunächst der aktuelle Kontostand, der natürlich immer noch 0 ist, gelesen. Anschließend wird der neue Stand zu 200 berechnet und entsprechend gesetzt. Petra Schmitt führt nun noch weitere Buchungen auf anderen Konten durch. Irgendwann wird die Ausführung des Threads von Andrea Müller fortgesetzt. In dem eigenen Exemplar der lokalen Variablen oldBalance befindet sich immer noch der Wert 0. Der neue Kontostand wird entsprechend zu −100 berechnet und für das Konto 47 so gesetzt. Die Gutschrift von 200 €, die Petra Schmitt ausge-

führt hat, ist verloren gegangen; der aktuelle Kontostand beträgt −100 € statt +100 €. Diese Tatsache dürfte den Kontoinhaber nicht gerade erfreuen.

Man könnte nun einwenden, dass der soeben beschriebene Ablauf sehr unwahrscheinlich ist. Dies ist in der Tat so. Allerdings sollte dies kein Grund zur Beruhigung sein. Im Gegenteil: Wenn das angegebene Programm über Wochen und Monate läuft, steigt die Wahrscheinlichkeit an, dass der geschilderte Ablauf doch vorkommt. Eine verlorene Buchung wird zunächst keinen Fehler ergeben und nicht sofort auffallen, sondern vermutlich erst später. Man wird dann vielleicht sogar Petra Schmitt verdächtigen, die Buchung nicht durchgeführt zu haben. Eventuell gehen im Laufe des Jahres weitere Buchungen verloren. Schließlich denkt man auch daran, dass der Fehler in der Software zu suchen sein könnte. Solche Fehler, die nur sehr selten unter ganz bestimmten Bedingungen vorkommen, sind besonders schwer zu finden und damit auch schwer zu beheben.

Es kann auch vorkommen, dass derartige Probleme erst auftreten, nachdem eine neue Version des Betriebssystems installiert oder das Programm auf eine andere Hardware portiert wurde. Das Programmierteam der Bankensoftware argumentiert dann, dass der Fehler auf keinen Fall in ihrer Software zu suchen sei, denn diese lief ja auf dem alten System seit längerer Zeit fehlerfrei. Wenn sich dann nach einer gewissen Zeit, die der Bank durch die vorhandenen Probleme finanzielle Verluste eingebracht haben, herausstellt, dass der Fehler doch in der Bankensoftware steckt, dann wird dies die Geschäftsbeziehungen zwischen der Bank und dem Programmierteam enorm belasten. Sie sollten also im Zusammenhang mit paralleler Programmierung besonders sensibel für derartige Probleme sein.

Wir betrachten im folgenden zwei Versuche, das obige Problem zu lösen. Diese Ansätze lösen das Problem allerdings nicht. Erst in Abschnitt 2.3 werden Sie eine korrekte Lösung des geschilderten Problems kennenlernen.

2.2.1 Erster Lösungsversuch

Das Problem der verlorenen Buchungen kommt offensichtlich daher, dass eine Buchung aus mehreren Arbeitsschritten (Java-Anweisungen) besteht. Man könnte also denken, dass das Problem dann gelöst ist, wenn eine Buchung durch eine einzige Java-Anweisung realisiert wird. Im folgenden Programm (Listing 2.7) sind nur die Änderungen gegenüber dem obigen Programm dargestellt. Die Klasse Account enthält eine Methode transferMoney statt der Methoden zum Abfragen und Setzen des Kontostands. Diese Methode besteht aus einer einzigen Java-Anweisung. Die TransferMoney-Methode der Klasse Bank ruft dann einfach die TransferMoney-Methode der Klasse Account auf.

Listing 2.7

```
class Account
{
    private float balance;

    public void transferMoney(float amount)
    {
        balance += amount;
    }
}
```

```
class Bank
{
    private Account[] account;

    public Bank()
    {
       ... // wie bisher
    }

    public void transferMoney(int accountNumber, float amount)
    {
        account[accountNumber].transferMoney(amount);
    }
}

... //wie bisher
```

Dies ist keine Lösung unseres Problems, denn Java-Programme werden nicht im Quellcode ausgeführt, sondern erst in Java-Bytecode übersetzt und dieser Bytecode wird ausgeführt. Eine Anweisung wie

```
balance += amount;
```

wird dabei in mehrere primitive Anweisungen für die JVM (Java Virtual Machine) übersetzt. Schematisch sieht dies dann so aus:

```
lade den Inhalt von balance aus dem Hauptspeicher in ein Register;
addiere auf dieses Register den Inhalt von amount;
schreibe den Inhalt des Registers auf balance zurück;
```

Nun hat man wieder dasselbe Problem wie im ursprünglichen Bankenprogramm. Die Buchung besteht aus mehreren Teilschritten. Wenn nach Ausführung der ersten Anweisung eine Umschaltung erfolgt, kann dieselbe Situation, die zum Verlust einer Buchung führt, auch hier eintreten. Der Registerinhalt wird übrigens beim Umschalten auf einen anderen Thread abgespeichert und beim Zurückschalten auf diesen Thread wiederhergestellt. Das Register hat hier eine ähnliche Rolle wie zuvor die lokale Variable newBalance.

Ein weiterer Grund, der gegen diese Lösung spricht, ist die fehlende Allgemeingültigkeit. Wenn es uns auch hier gelungen ist, die kritische Operation in einer Java-Anweisung auszudrücken, so ist dies im Allgemeinen nicht möglich.

2.2.2 Zweiter Lösungsversuch

Ein weiterer Ansatz, das Problem der verlorenen Buchungen zu lösen, besteht in der Überlegung, dass zu einem Zeitpunkt nur eine Angestellte eine Buchung durchführen darf. Erst wenn eine Buchung vollständig abgeschlossen ist, darf eine andere Angestellte eine Buchung vornehmen. Dies ist in der Tat eine korrekte Lösung des Problems. Die Frage ist, wie man das realisieren kann. Im Folgenden wird dies mithilfe eines Sperrattributs der Klasse Bank versucht.

Die Klasse Account ist wieder so realisiert wie im ursprünglichen Programm; sie hat also Methoden zum Abfragen und Setzen des Kontostands. Alle anderen Klassen außer der

Klasse Bank sind ebenfalls unverändert. Die Klasse Bank wird wie folgt verändert (Änderungen sind in Listing 2.8 fett gedruckt):

Listing 2.8

```
class Bank
{
    private Account[] account;
    private boolean locked;

    public Bank()
    {
        account = new Account[100];
        for(int i = 0; i < account.length; i++)
        {
            account[i] = new Account();
        }
        locked = false;
    }

    public void transferMoney(int accountNumber, float amount)
    {
        while(locked);
        locked = true;

        float oldBalance = account[accountNumber].getBalance();
        float newBalance = oldBalance + amount;
        account[accountNumber].setBalance(newBalance);

        locked = false;
    }
}
```

Die drei bisherigen Anweisungen der TransferMoney-Methode können nur ausgeführt werden, falls die Bank im Moment nicht gesperrt ist (d. h. falls kein anderer Thread im Moment gerade dabei ist, eine Buchung durchzuführen). Ist die Bank gesperrt, so wartet man so lange, bis sie wieder frei ist. Danach sperrt man selber die Bank, um anzuzeigen, dass gerade eine Buchung läuft und dass kein anderer Thread eine Buchung beginnen darf. Nachdem die Buchung vollständig durchgeführt wurde, wird die Sperre wieder freigegeben. Dasselbe Verhalten spielt sich beispielsweise auch beim Besuch einer Toilette ab.

Diese auf den ersten Blick richtig aussehende Lösung hat allerdings eine ganze Reihe von Problemen:

1. Die Parallelität wird unnötig stark eingeschränkt. Es ist nämlich nicht kritisch, wenn Buchungen (quasi-)gleichzeitig auf unterschiedlichen Konten durchgeführt werden. Probleme können sich nur ergeben, wenn zwei Bankangestellte auf demselben Konto arbeiten. Sperren sollte es also für jedes Konto und nicht für die gesamte Bank geben.
2. Die Effizienz der Lösung ist äußerst schlecht. Wenn auf einen Thread umgeschaltet wird, der eine Buchung nicht durchführen kann, weil die Bank gerade gesperrt ist, so wird die gesamte kostbare Rechenzeit dafür verwendet, die Schleife while(locked), sehr oft auszuführen. Für die Geschäfte der Bank bringt dies keinen Fortschritt. Man bezeichnet diese Art des Wartens als *aktives Warten (busy waiting)* oder *Polling*. Die Leserinnen und Leser

dieses Buches sollen unter anderem lernen, aktives Warten in der Regel zu vermeiden und nur in begründeten Ausnahmefällen zu verwenden.

3. Das größte Problem dieser Lösung besteht jedoch darin, dass sie falsch ist! Das Warten auf das Freiwerden der Sperre und das Setzen der Sperre ist wiederum eine Aktion, die in mehrere Teilschritte zerfällt. Auch hier kann es wiederum zu einem Umschalten zu einem ungünstigen Zeitpunkt kommen.

Um dies zu verstehen, betrachten wir die Anweisungen

```
while(locked);
locked = true;
```

und ihre Übersetzung, die man schematisch so darstellen kann:

```
lade locked in ein Register;
falls dieses Register == true, springe zurück zur vorigen Anweisung;
lade true in locked;
```

Nehmen wir wieder an, dass eine Umschaltung nach der ersten Anweisung stattfindet, wobei der Registerwert gerettet wird. Nehmen wir weiter an, dass die Sperre zu diesem Zeitpunkt frei ist (das Register enthält also den Wert false). Der nächste Thread kann nun ebenfalls die TransferMoney-Methode einmal oder mehrmals ausführen. Wenn auf den ersten Thread zurückgeschaltet wird, während sich der zweite mitten in der Ausführung der TransferMoney-Methode befindet, dann kann der erste nun ebenfalls die Buchung durchführen, denn der alte Registerwert, der wieder geladen wird, enthält false. Also findet der Rücksprung in der zweiten Anweisung nicht statt und der erste Thread kann mit der Buchung beginnen. Dass dieser Ablauf tatsächlich eintritt und dann noch zusätzlich eine Buchung verloren geht, ist noch unwahrscheinlicher als zuvor. Aber wie schon zuvor diskutiert wurde, sollte dies kein Grund zur Beruhigung sein. Im Gegenteil: Es macht die Angelegenheit schlimmer.

2.3 synchronized und volatile

2.3.1 Synchronized-Methoden

Zur Lösung der soeben beschriebenen Probleme Nr. 2 und 3 verwenden wir die Möglichkeit, in Java Methoden als *synchronized* zu kennzeichnen. Im Folgenden (Listing 2.9) ist die Klasse Bank nochmals gezeigt. Das eingefügte Schlüsselwort synchronized, das zur Hervorhebung fett gedruckt ist, ist die einzige Änderung, die an dem ursprünglichen Bankprogramm vorgenommen werden muss.

Listing 2.9

```
class Bank
{
    private Account[] account;
```

```
    public Bank()
    {
        ... // wie im ursprünglichen Bankprogramm
    }

    public synchronized void transferMoney(int accountNumber, float amount)
    {
        float oldBalance = account[accountNumber].getBalance();
        float newBalance = oldBalance + amount;
        account[accountNumber].setBalance(newBalance);
    }
}
```

In Java besitzt jedes Objekt eine *Sperre*. Diese ist in der Klasse Object realisiert. Da jede Klasse direkt oder indirekt von Object abgeleitet ist, gilt dies für alle Klassen und damit für alle Objekte. Wird eine Methode, die nicht static ist, mit synchronized gekennzeichnet, so muss die Sperre des betreffenden Objekts zuerst gesetzt werden, bevor die Methode ausgeführt werden kann. Ist die Sperre von einem anderen Thread bereits gesetzt, so wird der aufrufende Thread blockiert. Dieses Blockieren geschieht aber nicht durch aktives Warten, sondern in ähnlicher Form wie beim Aufruf der Methode sleep: Der Thread wird im Folgenden nicht mehr berücksichtigt, wenn es darum geht, auf welchen Thread umgeschaltet wird. Das Warten auf das Freiwerden der Sperre benötigt also keine kostbare Rechenzeit. Wenn eine mit synchronized gekennzeichnete Methode verlassen wird, so wird die Sperre für das entsprechende Objekt wieder freigegeben. Falls ein Thread auf die Freigabe der Sperre wartet, kann dieser jetzt fortgesetzt werden.

Das geschilderte Problem der Ineffizienz (Problem Nr. 2) besteht bei dieser Lösung nicht mehr, da die Threads nicht aktiv auf die Freigabe der Sperre warten. Das Problem der Inkorrektheit (Problem Nr. 3) besteht ebenfalls nicht mehr. Der Programmierer kann sich darauf verlassen, dass der Sperrmechanismus von Java in korrekter Weise umgesetzt wird, worauf im Rahmen dieses Buches aber nicht näher eingegangen werden soll.

Das Problem der unnötigen Einschränkung der Parallelität (Problem Nr. 1) ist damit noch nicht gelöst. Es ist naheliegend, statt der Sperre des Bank-Objekts die Sperren der Konto-Objekte zu verwenden. Ein erster Versuch dafür ist, dass man synchronized aus der TransferMoney-Methode der Klasse Bank wieder entfernt und stattdessen die Methoden getBalance und setBalance der Klasse Account mit synchronized kennzeichnet. Dies hilft allerdings gar nichts. Buchungen können wieder verloren gehen wie im ursprünglichen Bankprogramm. Denn wenn auch die Methoden zum Setzen und Abfragen synchronized sind, so können dennoch zwischen dem Aufruf der beiden Methoden getBalance und setBalance Aktionen anderer Threads ausgeführt werden wie zuvor:

```
float oldBalance = account[accountNumber].getBalance();
float newBalance = oldBalance + amount;
account[accountNumber].setBalance(newBalance);
```

Eine korrekte Lösung besteht darin, dass für die Klasse Account statt getBalance und setBalance eine TransferMoney-Methode wie in Abschnitt 2.2.1 definiert wird und diese als synchronized gekennzeichnet wird. Damit werden die Sperren der Konto-Objekte in korrekter Weise genutzt.

2.3.2 Synchronized-Blöcke

Falls man diese Lösung nicht einsetzen will oder kann, weil man z. B. keinen Zugriff auf den Quellcode der Klasse Account hat, so können *Synchronized-Blöcke* verwendet werden. Mit Synchronized-Blöcken kann eine beliebige Gruppe von Anweisungen durch { und } zu einem Anweisungsblock zusammengefasst und als synchronized gekennzeichnet werden. Dabei muss hinter dem Schlüsselwort synchronized eine Referenz auf ein Objekt angegeben werden, dessen Sperre zunächst frei sein muss und gesetzt wird, bevor mit der Ausführung des Anweisungsblocks begonnen wird. Am Ende des Anweisungsblocks wird die Sperre wieder freigegeben.

Im folgenden Beispielprogramm (Listing 2.10) ist dies umgesetzt, wobei wieder nur die Klasse Bank gezeigt wird und die Änderungen gegenüber der ursprünglichen Version fett gedruckt sind. In der Methode transferMoney wird zunächst die Sperre für das entsprechende Konto-Objekt belegt und dann wird, während diese Sperre gehalten wird, der Kontostand abgefragt und neu gesetzt. Wenn jetzt zwischen dem Abfragen und dem Setzen auf einen anderen Thread umgeschaltet wird, so kann dieser Thread zwar auf anderen Konten buchen, nicht aber auf demjenigen, für das der erste Thread die Sperre belegt hat. Zur Verdeutlichung sei erwähnt, dass die Methoden der Klasse Account immer noch so aussehen wie im ursprünglichen Bankprogramm, also nicht synchronized sind.

Listing 2.10

```
class Bank
{
    private Account[] account;

    public Bank()
    {
       ... // wie im ursprünglichen Bankenprogramm
    }

    public void transferMoney(int accountNumber, float amount)
    {
        synchronized(account[accountNumber])
        {
            float oldBalance = account[accountNumber].getBalance();
            float newBalance = oldBalance + amount;
            account[accountNumber].setBalance(newBalance);
        }
    }
}
```

Damit haben wir nun eine korrekte Lösung für unsere Bank, bei der auch die Parallelität nicht unnötig eingeschränkt wird.

Synchronized-Blöcke stellen ein allgemeineres Konzept dar als *Synchronized-Methoden*, denn man kann jede Synchronized-Methode als Nicht-Synchronized-Methode und einem Synchronized-Block realisieren. Statt

```
class C
{
    public synchronized void m()
    {
        ...
    }
}
```

kann man auch schreiben:

```
class C
{
    public void m()
    {
        synchronized(this)
        {
            ...
        }
    }
}
```

Beim Übersetzen eines Programms wird diese Umwandlung tatsächlich auch so vorgenommen.

In Abschnitt 2.1.4 wurden einige Beispiele für die Anzahl der möglichen Ablaufreihenfolgen angegeben, wenn Threads unsynchronisiert laufen. Eines dieser Beispiele betrachtete zwei Threads, wobei jeder Thread jeweils zehn Aktionen ausführt. Nehmen wir an, bei Thread 1 sind es die Aktionen A1 bis A10 und bei Thread 2 die Aktionen B1 bis B10. Wenn nun sowohl A1 bis A10 als auch B1 bis B10 jeweils in einen Synchronized-Block gesetzt werden, wobei von beiden Threads eine Referenz auf dasselbe Objekt bei synchronized angegeben wird, dann reduziert sich die Anzahl der möglichen Abläufe von 184 756 auf zwei, nämlich auf A1 – ... – A10 – B1 – ... – B10 und B1 – ... – B10 – A1 – ... – A10. Ich hoffe, dass damit nachträglich die Bemerkung aus Abschnitt 2.1.4 verstanden wird, dass durch Synchronisation die Anzahl der möglichen Abläufe eingeschränkt werden kann.

2.3.3 Wirkung von synchronized

Zur Überprüfung, ob die Bedeutung von synchronized verstanden wurde, betrachten wir folgende Klasse C:

```
class C
{
    public void m1() {...}
    public void m2() {...}
    public synchronized void ms1() {...}
    public synchronized void ms2() {...}
}
```

Wenn ein Thread die Methode ms1 auf ein Objekt o1 der Klasse C aufruft, und wenn während dieser Ausführung eine Umschaltung auf einen anderen Thread stattfindet, so kann dieser weder die Methode ms1 noch ms2 auf o1 anwenden, wohl aber die Methoden m1 und m2. Die Sperre für das Objekt o1 ist ja vom ersten Thread gesetzt worden, deshalb kann

beim Aufruf von ms1 oder ms2 die Sperre nicht gesetzt werden und der aufrufende Thread wird blockiert. Es ist dabei gleichgültig, ob ms1 oder ms2 aufgerufen wird; die Sperre hängt nicht von der Methode ab, sondern vom Objekt. Der Aufruf der Methoden m1 und m2 kann aber erfolgen, weil hier der Zustand der Sperre gar nicht geprüft wird. Umgekehrt gilt, dass ein zweiter Thread alle Methoden m1, m2, ms1 und ms2 bei einem Objekt o1 aufrufen kann, während ein erster Thread gerade die Methode m1 (oder m2) auf o1 anwendet.

Bisher haben wir nur ein Objekt betrachtet. Wenn ein Thread gerade dabei ist, die Methode ms1 auf ein Objekt o1 anzuwenden, so kann ein anderer Thread alle Methoden m1, m2, ms1 und ms2 auf ein anderes Objekt o2 anwenden, da dieses ja nicht gesperrt ist. Die Sperre bezieht sich also auf ein Objekt. Dies gilt allerdings nur für Synchronized-Methoden, die nicht static sind. Static-Attribute und -Methoden sind ja sogenannte Klassenattribute bzw. Klassenmethoden, die sich auf die Klasse und nicht auf einzelne Objekte beziehen. Entsprechend wird beim Aufruf einer Static-Synchronized-Methode eine Sperre für die Klasse geprüft und gesetzt. Es existieren also Sperren für jedes Objekt einer Klasse und für die Klasse selbst. Diese sind alle unabhängig voneinander. Das heißt, dass eine Synchronized-Methode, die nicht static ist, auf ein Objekt angewendet und die Sperre für dieses Objekt gesetzt werden kann, auch wenn die Sperre für die Klasse schon gesetzt ist. Das Setzen der Sperre für die Klasse bedeutet also nicht, dass auf allen Objekten dieser Klasse die entsprechenden Sperren gesetzt werden.

Wenn ein Objekt gesperrt ist und durch Aufruf einer Synchronized-Methode auf dieses Objekt die Sperre erneut gesetzt werden soll, dann hängt das Ergebnis dieser Prüfung davon ab, welcher Thread das Objekt momentan gesperrt hat. Ist es derselbe Thread, der jetzt erneut die Sperre anfordert, dann wird der Thread nicht blockiert, sondern kann die Methode ausführen. Andernfalls würde es hier zu einer Blockierung kommen, die nie wieder aufgelöst werden könnte. Das folgende Programm (Listing 2.11) verdeutlicht diesen Sachverhalt; die Main-Methode kann bis zum Ende ausgeführt werden.

Listing 2.11

```
class MyClass
{
    public synchronized void m1()
    {
        m2();
    }

    public synchronized void m2()
    {
    }

    public static void main(String[] args)
    {
        MyClass obj = new MyClass();
        obj.m1();
        //diese Stelle wird erreicht!!!
    }
}
```

In diesem Fall erfolgt also keine endlose Blockierung. Man bezeichnet diese Eigenschaft der Synchronized-Sperren als *reentrant* (wiederbetretbar; ein Thread kann eine Sperre mehrfach setzen). Damit soll aber nicht gesagt werden, dass solche endlosen Blockierungen grundsätzlich nie vorkommen können. Im Gegenteil: Diese als Verklemmung bekannte Problematik wird später ausführlich behandelt. In solchen Verklemmungssituationen befinden sich dann aber immer mindestens zwei Threads.

Konstruktoren können nie als synchronized gekennzeichnet werden. Man sollte immer so programmieren, dass dies auch nicht notwendig ist. Das heißt, die Initialisierung eines Objekts sollte abgeschlossen sein, bevor andere Threads damit arbeiten. Sollte es in einem extremen Fall aber dennoch notwendig sein, so kann man sich mit einem Synchronized(this)-Block behelfen.

2.3.4 Notwendigkeit von synchronized

Die Problematik der verloren gegangenen Buchung sollte nicht dazu führen, bei der Nutzung von Threads grundsätzlich alle Methoden mit synchronized zu kennzeichnen. Zum einen können dadurch eher Verklemmungen entstehen. Zum anderen ist die Verwendung von synchronized nicht ganz kostenlos. Es ist beim Aufruf der entsprechenden Methoden die Sperre zu prüfen und zu setzen. Entsprechend muss am Ende die Sperre wieder freigegeben werden. Man sollte sich also im Zusammenhang mit Threads immer genau überlegen, ob synchronized notwendig ist oder nicht.

Wie schon einige Seiten zuvor erwähnt sind lokale Variablen von Methoden für jeden Aufruf vorhanden und daher nie ein Grund für den Einsatz von synchronized. Dies gilt in jedem Fall für die primitiven Datentypen (int, boolean, short, float usw.). Für Referenzen gilt dies auch, nicht aber in jedem Fall für die Objekte, auf die solche Referenzen zeigen. Werden die Objekte, auf die die lokalen Referenzvariablen zeigen, in der Methode neu erzeugt, dann gibt es auch hier keine Probleme bzgl. der Parallelität. Die gemeinsame Nutzung bezieht sich also immer auf Attribute von Objekten und Klassen. Hier ist im Einzelfall zu analysieren, ob eine Synchronisation nötig ist.

Man kann sich nun fragen, ob für das Lesen und Schreiben von Attributen der Basisdatentypen wie boolean, int, long usw. sowie Referenzen auch synchronized notwendig ist. Um uns einer Antwort auf diese Frage zu nähern, betrachten wir zunächst, dass bei Java garantiert wird, dass das Lesen und Schreiben von Referenzen sowie aller Basisdatentypen außer long und double *atomar* (d. h. *unteilbar*) erfolgt. Für long und double wird diese *Atomarität* nicht gewährleistet, weil diese beiden Datentypen durch 64 Bits repräsentiert werden. Deshalb kann es sein, dass das Lesen und Schreiben in zwei Schritten für jeweils 32 Bits erfolgt. Dadurch könnte ohne Synchronisation z. B. Folgendes passieren: Angenommen, die Bits eines Attributs des Typs wären alle null. Der Wert würde nun so verändert, dass alle Bits eins sind. Wenn ein anderer Thread diesen Wert liest, so liest er entweder vor oder nach der Änderung, woraus folgt, dass er entweder lauter Nullen oder lauter Einsen liest. Wenn nun aber das Lesen und Schreiben nicht atomar ist und nicht synchronisiert erfolgt, kann es dazu kommen, dass ein lesender Thread einen Wert zurückbekommt, bei dem 32 Bits null und 32 Bits eins sind. Das ist ein Wert, den es nie gegeben hat und der deshalb nie gelesen werden sollte. Also muss auch in diesem Fall synchronized verwendet werden.

2.3.5 volatile

Nun scheint es also so zu sein, dass für das Lesen und Schreiben von Attributen der Basisdatentypen außer long und double sowie Referenzen kein synchronized notwendig ist. Aber auch dies kann im Allgemeinen nicht angenommen werden, denn es kann zu einem weiteren Problem kommen, das wir noch nicht diskutiert haben. Angenommen, wir haben eine Klasse mit einem Int-Attribut und einer Get- und Set-Methode zum Lesen und Schreiben des Attributs. Nehmen wir an, beide Methoden seien nicht synchronized. Nun kann es sein, dass ein Thread wiederholt die Get-Methode aufruft. Der Compiler könnte nun folgende Optimierung beim Erzeugen von Byte-Code vorgenommen haben: Da der Wert des Attributs scheinbar zwischen den Get-Aufrufen nicht verändert wird, muss der Wert des Attributs nicht jedes Mal aus dem Hauptspeicher geladen werden, sondern der Wert könnte in einem Register des Prozessors gespeichert und immer wieder verwendet werden. Dadurch würde dieser Thread nicht bemerken, wenn der Wert durch einen anderen Thread verändert wird. Diese Optimierung kann vermieden werden, indem das Attribut mit dem Schlüsselwort *volatile* gekennzeichnet wird (s. Listing 2.12):

Listing 2.12

```
class VolatileExample
{
    private volatile int value;

    public void setValue(int value)
    {
        this.value = value;
    }

    public int getValue()
    {
        return value;
    }
}
```

Das Wort „volatile“ bedeutet flüchtig. Damit ist gemeint, dass man nicht davon ausgehen kann, dass sich dieser Wert zwischen zwei lesenden Zugriffen nicht ändert, falls er vom betrachteten Thread nicht verändert wird, sondern dass er von anderen Threads verändert werden könnte und deshalb bei jedem Lesezugriff aus dem Hauptspeicher geholt werden soll. Die Synchronized-Blöcke und Synchronized-Methoden haben neben dem Behandeln der Sperre zusätzlich den Volatile-Effekt. Das heißt, synchronized wirkt so, als ob alle betroffenen Attribute mit volatile gekennzeichnet wären. Der Einfachheit halber werden wir im weiteren Verlauf des Buchs volatile nicht mehr verwenden, sondern nur noch synchronized.

2.3.6 Regel für die Nutzung von synchronized

Aus den bisherigen Ausführungen ergibt sich, dass der Zugriff auf gemeinsam benutzte Attribute durch mehrere Threads immer synchronized erfolgen muss, auch wenn es nur um das Lesen und Schreiben eines einfachen Int-Attributs geht (Listing 2.13):

Listing 2.13

```
class SynchronizedExample
{
    private int value;

    public synchronized void setValue(int value)
    {
        this.value = value;
    }

    public synchronized int getValue()
    {
        return value;
    }
}
```

Falls aber die Attribute eines Objekts von einem Thread verändert werden und erst anschließend werden weitere Threads erzeugt, die diese Attribute lediglich lesen, dann muss man nicht synchronisieren. Denn zum einen ist keine Gleichzeitigkeit im Spiel, zum anderen garantiert das Neustarten von Threads einen Volatile-Effekt (dasselbe gilt übrigens für das Warten auf das Ende von Threads mit join, s. Abschnitt 2.4). Synchronisation ist immer nur dann nötig, falls mehrere Threads auf gemeinsame Daten zugreifen und mindestens einer dieser Threads verändert die Daten. In diesem Fall ist es wichtig, alle Methoden, die auf Attribute des Objekts zugreifen, als synchronized zu kennzeichnen, gleichgültig, ob in einer Methode die Attribute des Objekts nur gelesen oder auch verändert werden. Die schreibenden Methoden überführen den Zustand eines Objekts (repräsentiert durch die aktuellen Werte seiner Attribute) von einem konsistenten Zustand in einen anderen konsistenten Zustand. In der Regel erfolgt dieser Zustandswechsel durch mehrere Einzelschritte. Wenn nun eine rein lesende Methode den Zustand des Objekts in einem Zwischenzustand sehen könnte, dann wird ein inkonsistenter Zustand des Objekts ausgelesen. Dies ist in der Regel nicht erwünscht und kann zu Folgefehlern führen. Nehmen wir als Beispiel eine Bank, in der nur Überweisungen von einem Konto dieser Bank auf ein anderes Konto dieser Bank durchgeführt werden können. Dies bedeutet, dass die Summe aller Kontostände immer konstant bleibt. Eine Überweisung wird durchgeführt, indem zuerst ein Betrag von einem Konto abgebucht und einem anderen Konto gutgeschrieben wird (oder in umgekehrter Reihenfolge). Wenn nun ein Thread eine Überweisung vornimmt und nach der Änderung des ersten Kontos auf einen rein lesenden Thread umgeschaltet wird, der überprüft, ob die Summe aller Kontostände noch denselben Wert hat, dann wird diese Überprüfung fehlschlagen – es fehlt scheinbar Geld (oder es ist scheinbar zu viel Geld vorhanden).

Es ist also folgende Regel einzuhalten:

Wenn von mehreren Threads gleichzeitig auf ein Objekt zugegriffen werden kann, wobei mindestens ein Thread den Zustand des Objekts ändert (das heißt, dessen Attribute schreibt), dann müssen alle lesenden und schreibenden Zugriffe auf das Objekt synchronisiert werden (z. B. mit synchronized).

2.4 Ende von Java-Threads

Ein Thread ist zu Ende, wenn seine Run-Methode bzw. die Main-Methode im Falle des Ursprungs-Threads beendet ist. Sind alle Threads zu Ende, so ist der gesamte Prozess zu Ende (diese Aussage ist nicht ganz korrekt; sie wird in Abschnitt 2.9 präzisiert werden). Betrachten Sie nochmals die Beispiele aus Abschnitt 2.1.3. In der Main-Methode werden nur die Thread-Objekte erzeugt und die Threads gestartet. Auch wenn danach die Main-Methode zu Ende ist, so existiert der Prozess noch weiter, und zwar so lange, bis die gestarteten Threads ebenfalls zu Ende gelaufen sind.

Für die weiteren Überlegungen sollte man sich nochmals den Unterschied zwischen dem Thread-Objekt und dem eigentlichen Thread verdeutlichen. In der Küchen-Köche-Metapher könnte man das Thread-Ende mit dem Tod eines Kochs vergleichen, wobei der Körper, der dem Thread-Objekt entspricht, nach dem Tod des Kochs noch vorhanden ist. Wie ein Toter nicht wieder zum Leben erweckt werden kann, kann auf ein Thread-Objekt die Start-Methode maximal einmal angewendet werden. Zu einem Thread-Objekt gibt es also maximal einen Thread.

Durch die Methode *isAlive* der Klasse Thread kann ein Thread-Objekt befragt werden, ob der dazugehörige Thread lebt. Die Methode liefert true oder false zurück, je nachdem, ob der entsprechende Thread läuft oder beendet ist bzw. nie gestartet wurde. Damit diese Methode angewendet werden kann, muss natürlich noch das dazugehörige Thread-Objekt vorhanden sein. Die Methode isAlive ist in der Klasse Thread wie folgt definiert:

```
public class Thread
{
      public final boolean isAlive() {...}
      ...
}
```

Man sollte diese Methode allerdings nicht einsetzen, um in einer Schleife auf das Ende eines Threads zu warten:

```
//MyThread sei eine aus Thread abgeleitete Klasse
MyThread t = new MyThread();
t.start();
while(t.isAlive());
//jetzt gilt: t.isAlive() == false; also ist der Thread beendet
...
```

Denn diese Art des Wartens ist *aktives Warten* wie beim Versuch der Realisierung einer Sperre in Abschnitt 2.2; es wird unnötig Rechenzeit verbraucht. Wenn man also nicht nur zwischendurch wissen möchte, ob ein Thread zu Ende gelaufen ist, sondern wenn man auf das Ende eines Threads warten möchte, dann sollte man die *Join-Methoden* der Klasse Thread einsetzen:

```
public class Thread
{
      public final void join() throws InterruptedException {...}
      public final void join(long millis) throws InterruptedException {...}
      public final void join(long millis, int nanos)
```

```
            throws InterruptedException {...}
        ...
}
```

Alle Join-Methoden können eine Ausnahme der Art *InterruptedException* werfen wie sleep. Die Methode join ist mehrfach überladen. Die Variante ohne Argumente wartet auf das Ende eines Threads, wie lange das auch immer dauern mag. Im Extremfall kann das unendliches Warten bedeuten, wenn z. B. der Thread, auf den gewartet wird, eine Endlosschleife in seiner Run-Methode ausführt. Die beiden Varianten mit Argumenten geben eine maximale Wartezeit an, wobei einmal die Wartezeit in Millisekunden und einmal in Milli- und Nanosekunden angegeben werden kann. Der aufrufende Thread kehrt aus dem Aufruf der Join-Methode zurück, falls der Thread, auf dessen Ende gewartet wird, zu Ende gelaufen oder die angegebene Zeit vergangen ist, was immer auch zuerst eintreten mag. Dass der Aufruf auch durch eine Unterbrechung mit einer Ausnahme enden kann, wird später thematisiert.

2.4.1 Asynchrone Beauftragung mit Abfragen der Ergebnisse

Das folgende Programm zeigt ein Beispiel, in dem auf das Ende von Threads gewartet werden muss. Die Aufgabe besteht darin, in einem sehr großen Feld des Typs boolean die Anzahl der True-Werte zu zählen. Diese Aufgabe wird auf mehrere Threads aufgeteilt, wobei jeder Thread in einem gewissen Bereich dieses Feldes zählt. Nachdem alle Threads zu Ende gelaufen sind (auf dieses Ende wird mit join gewartet), werden die Zählergebnisse der Threads addiert. Im folgenden Beispiel (Listing 2.14) wird zur Abwechslung mit der Schnittstelle Runnable gearbeitet.

Listing 2.14

```
class Service implements Runnable
{
    private boolean[] array;
    private int start;
    private int end;
    private int result;

    public Service(boolean[] array, int start, int end)
    {
        this.array = array;
        this.start = start;
        this.end = end;
    }

    public int getResult()
    {
        return result;
    }

    public void run()
    {
        for(int i = start; i <= end; i++)
```

```
        {
            if(array[i])
            {
                result++;
            }
        }
    }
}

public class AsynchRequest
{
    private static final int ARRAY_SIZE = 1_000_000_000;
    private static final int NUMBER_OF_SERVERS = 10;

    public static void main(String[] args)
    {
        // Feld erzeugen, jeder 10. Wert ist true
        boolean[] array = new boolean[ARRAY_SIZE];
        for(int i = 0; i < ARRAY_SIZE; i++)
        {
            if(i % 10 == 0) //alternativ: if(Math.random() < 0.1)
            {
                array[i] = true;
            }
            else
            {
                array[i] = false;
            }
        }

        // Startzeit messen
        long startTime = System.currentTimeMillis();

        // Feld für Services und Threads erzeugen
        Service[] service = new Service[NUMBER_OF_SERVERS];
        Thread[] serverThread = new Thread[NUMBER_OF_SERVERS];

        // Threads erzeugen
        int start = 0;
        int end;
        int howMany = ARRAY_SIZE / NUMBER_OF_SERVERS;

        for(int i = 0; i < NUMBER_OF_SERVERS; i++)
        {
            if(i < NUMBER_OF_SERVERS-1)
            {
                end = start + howMany - 1;
            }
            else
            {
                end = ARRAY_SIZE-1;
            }
            service[i] = new Service(array, start, end);
            serverThread[i] = new Thread(service[i]);
            serverThread[i].start();
```

```
            start = end + 1;
        }

        // Synchronisation mit Servern (auf Serverende warten)
        for(int i = 0; i < NUMBER_OF_SERVERS; i++)
        {
            try
            {
                serverThread[i].join();
            }
            catch(InterruptedException e)
            {
            }
        }

        // Gesamtergebnis aus Teilergebnissen berechnen
        int result = 0;
        for(int i = 0; i < NUMBER_OF_SERVERS; i++)
        {
            result += service[i].getResult();
        }

        // Endzeit messen
        long endTime = System.currentTimeMillis();
        float time = (endTime - startTime) / 1_000.0f;
        System.out.println("Rechenzeit: " + time);

        // Ergebnis ausgeben
        System.out.println("Ergebnis: " + result);
    }
}
```

Die Klasse Service enthält als Attribute die Parameter des Auftrags, der von einem Thread bearbeitet werden soll, nämlich eine Referenz auf das boolesche Feld und die Indizes, in denen das Feld untersucht werden soll. Diese Attribute werden im Konstruktor gesetzt. Ein weiteres Attribut ist das errechnete Resultat, das mit der Methode getResult erfragt werden kann. Die Run-Methode zählt die True-Werte im angegebenen Bereich des Feldes. In der Main-Methode der Klasse AsynchRequest wird zunächst ein relativ großes boolesches Feld der Größe 1 000 000.000 erzeugt, wobei jedes 10. Feldelemente auf true gesetzt wird (alternativ könnte man z. B. mithilfe von Zufallszahlen auch im Mittel 1/10 der Werte auf true setzen, s. Kommentar im Programmcode). Anschließend wird je ein Feld für die Service-Objekte und die Thread-Objekte erzeugt. Danach erfolgt in einer Schleife die Erzeugung der Service-Objekte und der Thread-Objekte. Dem Konstruktor aller Service-Objekte wird eine Referenz auf dasselbe boolesche Feld übergeben, während die Bereiche, in denen nach True-Werten gesucht wird, jeweils neu berechnet werden. Alle Threads werden nach ihrer Erzeugung gestartet und können nun parallel ihre Suche durchführen. Wenn man wie in diesem Fall einen Auftrag startet und nicht direkt danach auf das Ergebnis wartet, sondern noch andere Tätigkeiten durchführt (in diesem Fall weitere Aufträge startet), so wird dies als asynchrone Auftragserteilung bezeichnet. Daher wurde diese Klasse AsynchRequest genannt. Anschließend wird gewartet, bis alle gestarteten Threads zu Ende gelaufen sind. Hierzu wird die neu eingeführte Methode join verwendet. Erst danach kann das Ergebnis, das die einzelnen Threads berechnet haben, abgefragt und aufsummiert werden. Zum Schluss wird das Endergebnis ausgegeben.

Obwohl jedes einzelne Ergebnisattribut von dem dazugehörigen Thread geschrieben und vom Main-Thread gelesen wird, ist an dieser Stelle kein synchronized notwendig. Dies liegt zum einen daran, dass start und join einen Volatile-Effekt besitzen. Dies bedeutet: Wenn ein Thread a Daten ändert, ein anderer Thread b danach von Thread a gestartet wird und dann diese von Thread a veränderten Daten liest, dann ist garantiert, dass Thread b die Änderung des Threads a sieht. Ähnliches gilt für join: Wenn ein Thread a Daten ändert, ein anderer Thread b auf das Ende von Thread a mit join wartet und dann diese von Thread a veränderten Daten liest, dann ist garantiert, dass Thread b die Änderung des Threads a sieht. In diesem Fall ist es so, dass die Threads vom Main-Thread erst gestartet werden, nachdem dieser das boolesche Feld beschrieben hatte, und dass das Attribut result vom Main-Thread erst abgefragt wird, nachdem dieser auf das Ende aller Threads mit join gewartet hat. Zum anderen greifen die parallel laufenden Threads auf unterschiedliche Bereiche des Feldes zu, und dies auch nur lesend.

Wie aus Listing 2.14 ersichtlich ist, befindet sich noch eine Zeitmessung in der Main-Methode. Damit kann man untersuchen, wie die Laufzeit für das Zählen der True-Werte von der Anzahl der eingesetzten Threads abhängt. Die Tabelle 2.1 zeigt einige Messergebnisse. Dabei wurde die Konstante NUMBER_OF_SERVERS der Reihe nach entsprechend variiert.

Tabelle 2.1 Laufzeiten des Programms AsynchRequest

NUMBER_OF_SERVERS	Gemessene Zeit (in Sekunden)
1	0,419
2	0,238
3	0,202
4	0,188
5	0,189
10	0,196
50	0,216
100	0,226
1000	0,327
10 000	1,275
100 000	13,673

Wie man sieht, ergibt sich eine deutliche Reduktion um ca. die Hälfte der Laufzeit bei der Benutzung von zwei Threads gegenüber einem Thread. Weitere Laufzeitverringerungen, wenn auch weniger stark ausgeprägt, erhält man bei der Nutzung von drei und vier Threads. Das ist nicht verwunderlich, denn die Messungen wurden auf einem Rechner mit einem Vierkern-Prozessor durchgeführt. Offenbar können also bis zu vier Threads echt parallel von diesem Prozessor ausgeführt werden.

Werden mehr als vier Threads verwendet, so sinkt die Ausführungszeit nicht weiter ab, sondern bleibt bis ca. 100 Threads in etwa gleich (die Schwankungen dürften statistischer Natur sein, denn es wird ja nicht wirklich die Ausführungszeit nur unseres Prozesses gemessen, sondern die vergangene Zeit, die natürlich auch davon abhängt, was während der Messung sonst noch alles auf dem Rechner passiert). Werden sehr viele Threads verwendet

(10 000 oder gar 100 000), so steigt die Laufzeit sehr deutlich an. Dies liegt zum einen daran, dass nun die Thread-Erzeugung deutlich ins Gewicht fällt. Außerdem sollte man sich vor Augen führen, dass beim Einsatz von sehr vielen Threads das Starten aller Threads relativ lange dauert, während jeder einzelne Thread nur noch eine kleine Ausführungszeit hat. Bei 100 000 Threads z. B. untersucht jeder Thread nur noch 10 000 Feldelemente, sodass das Starten von 100 000 Threads deutlich aufwendiger sein dürfte als das, was jeder einzelne Thread zu tun hat. Daraus folgt, dass z. B. bei 100 000 gestarteten Threads diese zu keinem Zeitpunkt gleichzeitig existieren, denn die zuerst gestarteten Threads dürften schon wieder zu Ende gelaufen sein, wenn die letzten Threads schließlich gestartet werden. Dies ist ein weiterer Grund, warum sich bei einer Vergrößerung der Anzahl von Threads keine weiteren Geschwindigkeitssteigerungen mehr ergeben können.

Daraus könnte man nun folgern, dass man immer genau so viele Threads verwenden sollte, wie der Rechner Prozessoren bzw. Kerne besitzt. Durch die folgende Anweisung kann die Anzahl der Prozessoren bzw. Kerne erfragt werden:

```
int numberOfProcessors = Runtime.getRuntime().availableProcessors();
```

Diese Schlussfolgerung (Anzahl der Threads = Anzahl der Kerne bzw. Prozessoren) gilt aber nur für rechenintensive Threads. Damit sind Threads gemeint, die nie warten müssen (z. B. wegen einer Ein-/Ausgabe [EA]). Handelt es sich jedoch um EA-intensive Threads, die immer wieder warten müssen, dann gibt es auch dann noch Laufzeitverbesserungen, wenn man deutlich mehr Threads einsetzt, als es Prozessoren bzw. Kerne auf dem verwendeten Rechner gibt. Im folgenden Programm wird dieses Warten auf eine Ein-/Ausgabe simuliert, indem in der Schleife der Run-Methode der Klasse Service nach dem Zählen ein Aufruf der Sleep-Methode eingefügt wird (der eingefügte Programmcode ist in Listing 2.15 fett gedruckt):

Listing 2.15

```
public void run()
{
    for(int i = start; i <= end; i++)
    {
        if(array[i])
        {
            result++;
        }
        try
        {
            Thread.sleep(10);
        }
        catch(InterruptedException e)
        {
        }
    }
}
```

Wenn die Messungen jetzt wiederholt werden, dann ist natürlich die Laufzeit insgesamt wesentlich größer. Wir verwenden daher jetzt nur noch ein Feld der Größe 10 000 (der Wert der Konstanten ARRAY_SIZE wurde entsprechend verringert). Jetzt kann man die Geschwindigkeitssteigerungen auch beim Einsatz einer deutlich größeren Anzahl von Threads erkennen (bis ca. 1000 Threads), wie die Tabelle 2.2 zeigt:

Tabelle 2.2 Laufzeiten des Programms AsynchRequest bei einer Feldgröße von 10 000 mit zusätzlichem sleep(10)

NUMBER_OF_SERVERS	Gemessene Zeit (in Sekunden)
1	156,428
2	78,137
5	31,314
10	15,629
50	3,125
100	1,578
500	0,359
1000	0,266
5000	0,453
10 000	0,875

Die zu erwartende Laufzeit bei einem einzigen Thread ergibt sich im Wesentlichen aus der Zeit zum Zählen der True-Werte (diese ist bei 10 000 Werten so gering, dass man sie mit der hier verwendeten Methode nicht messen kann, d. h. man erhält den Wert 0) plus der aufsummierten Wartezeit:

Man würde also bei der Nutzung eines einzigen Threads einen Wert knapp über 100 s erwarten. Mit einem anderen Rechner konnte ein solcher Wert auch tatsächlich gemessen werden. Auf meinem Rechner ergibt sich eine unerwartet hohe Laufzeit von 156 s. Dies liegt daran, dass der Aufruf sleep(10) überraschenderweise jedes Mal ca. 15 oder 16 ms dauert, wie man ebenfalls messen kann. Diese Tatsache sollte Sie lehren, dass Sie bei einem üblichen Betriebssystem (das heißt: keinem Echtzeitbetriebssystem) nicht davon ausgehen können, dass Ihr Thread unmittelbar weiterarbeitet, sobald die Schlafenszeit abgelaufen ist. Dies zur Erklärung der absoluten Laufzeit, die aber hier nur in zweiter Linie interessant ist.

Bei dieser Messreihe geht es primär um den Vergleich der Rechenzeiten. Man erkennt sehr deutlich, dass die Lösung mit N Threads ca. um den Faktor N schneller ist als die Lösung mit einem Thread, da das Schlafen, welches den wesentlichen Teil der Laufzeit ausmacht, vollständig parallel erfolgt. Eine solche Verbesserung ist bis zum Einsatz von 1000 Threads erkennbar. Danach wird die Laufzeit wieder größer, weil jetzt der Overhead durch den Einsatz sehr vieler Threads dominiert und außerdem sehr viele Threads hintereinander ausgeführt werden.

Abschließend folgen noch zwei Bemerkungen zur Zeitmessung, die wir mit System.currentTimeMillis durchgeführt haben:

- Alternativ kann man auch zu Beginn und am Ende des Programms System.nanoTime aufrufen und wieder die Differenz beider Werte messen. Man erhält nun die vergangene Zeit in Nanosekunden (diese Zahl ist um den Faktor 1 000 000 größer als die zuvor gemessenen Millisekunden). Dieser Wert erscheint auf den ersten Blick wesentlich genauer zu sein. Allerdings werden Sie bei mehrfacher Messung ohnehin feststellen, dass die Werte bei wiederholter Ausführung schwanken, da eben die vergangene Zeit auch davon abhängt, welche Aufgaben Ihr Computer zur Zeit der Messung noch zu bewältigen hat. Deshalb bringt eine Messung in Nanosekunden für unseren Zweck keine wirklichen Vor-

teile; uns geht es ohnehin nicht um die exakten Werte, sondern nur um den Vergleich der Werte und die daraus abgeleiteten Folgerungen.

- Es gibt auch die Möglichkeit, die Rechenzeit eines Prozesses abzufragen (durch die Methode totalCpuDuration von ProcessHandle.Info). Dies wirkt auf den ersten Augenblick als die bessere Variante. Allerdings würden wir bei der Nutzung dieser Messmethode eine Enttäuschung erleben, denn die gemessene CPU-Zeit ist bei Nutzung eines einzigen Threads in etwa so groß wie die gemessene vergangene Zeit. Bei der Erhöhung der Anzahl von Threads sinkt aber die gemessene CPU-Zeit nicht ab. Im Gegenteil wird sie sogar größer. Dies ist aber nicht verwunderlich, denn ob ein Prozessor eine gewisse Zeit rechnet oder zwei Prozessoren jeweils die Hälfte oder vier Prozessoren jeweils ein Viertel dieser Rechenzeit, läuft in der Summe der Rechenzeiten auf dasselbe hinaus. Durch den zusätzlichen Aufwand für die Verwaltung von Threads wird die Rechenzeit umso größer, je mehr Threads genutzt werden. Bei der Parallelisierung geht es also nicht um die Ersparnis von Rechenzeit, also Arbeit, sondern es wird dieselbe Arbeit in kürzerer Zeit erbracht.

2.4.2 Zwangsweises Beenden von Threads

In manchen Fällen soll es möglich sein, das Beendigen eines Threads „von außen“ anzustoßen. Wenn etwa wie im letzten Beispiel ein Thread mit einem gewissen Auftrag gestartet wird, dann kann es sein, dass der Auftraggeber für die Ausführung eines Auftrags nur eine bestimmte Maximalzeit einräumt. Sollte der Thread innerhalb dieser Zeit seinen Auftrag nicht erledigt haben, soll er vom Auftraggeber abgebrochen werden. Die Klasse Thread besitzt zu diesem Zweck die Methode *stop*. Diese Methode ist allerdings „deprecated“ (missbilligt). Das bedeutet, dass empfohlen wird, sie nicht zu verwenden, zum einen, weil ihre Verwendung problematisch sein kann, zum anderen, weil sie in zukünftigen Java-Versionen nicht mehr vorhanden sein könnte und damit Programme, die diese Methode verwenden, nicht mehr lauffähig wären. Diese Methode wurde praktisch wieder aus der Klasse Thread herausgenommen, weil bei einer derartigen Beendigung alle von diesem Thread gesperrten Objekte wieder freigegeben werden. Wenn der Thread aber gerade dabei war, durch Aufruf einer Synchronized-Methode den Zustand eines Objekts zu verändern, so kann es sein, dass diese Methode nur zum Teil ausgeführt wurde und sich das Objekt danach in einem inkonsistenten Zustand befindet.

Das Problem eines inkonsistenten Zustands wurde schon am Ende von Abschnitt 2.3 durch folgendes Beispiel erläutert: Angenommen, bei einer Bank gebe es nur Umbuchungen von einem Konto der Bank auf ein anderes. Das heißt, dass der Betrag, der von einem Konto abgebucht wird, einem anderen gutgeschrieben wird. Für das System gilt die *Invariante*, dass die Summe aller Kontostände immer dieselbe ist. Wenn nun aber ein Thread während einer Umbuchung abgebrochen wird, so könnte eine der beiden Buchungen schon erfolgt sein, während die andere Buchung nicht mehr ausgeführt wird. Die Invariante der konstanten Kontostandssumme wäre damit nicht mehr gültig.

Da also durch die Möglichkeit des Abbruchs von Threads inkonsistente Zustände entstehen können, soll die Methode stop nicht benutzt werden. Stattdessen muss der Anwendungsprogrammierer die Möglichkeit des Abbrechens selber umsetzen. Dies kann z. B. durch ein boolesches Attribut des Thread-Objekts, das in der Run-Methode regelmäßig abgefragt

wird, realisiert werden. Ist der Attributwert false, so läuft der Thread weiter, andernfalls wird die Run-Methode beendet, wodurch dann auch der Thread zu Ende ist. Dieser Attributwert muss „von außen“ veränderbar sein, damit der Thread abgebrochen werden kann. Das folgende Programm (Listing 2.16) zeigt die Idee dieser Realisierung.

Listing 2.16

```
public class StopThread extends Thread
{
    private boolean stopped = false;

    public StopThread()
    {
        start();
    }

    public synchronized void stopThread()
    {
        stopped = true;
    }

    public synchronized boolean isStopped()
    {
        return stopped;
    }

    public void run()
    {
        int i = 0;
        while(!isStopped())
        {
            i++;
            System.out.println("Hallo Welt (" + i + ")");
        }
        System.out.println("Thread endet jetzt ...");
    }

    public static void main(String[] args)
    {
        StopThread st = new StopThread();
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException e)
        {
        }
        st.stopThread();
    }
}
```

Das Attribut stopped der Klasse StopThread wird von zwei Threads benutzt, wobei einer der beiden Threads, der Main-Thread, dieses Attribut durch Aufruf der Methode stopThread verändert. Nach unseren bisher eingeführten Prinzipien muss der Zugriff auf dieses Attribut daher in Synchronized-Methoden stattfinden. Dabei ist es gleichgültig, ob das Attribut