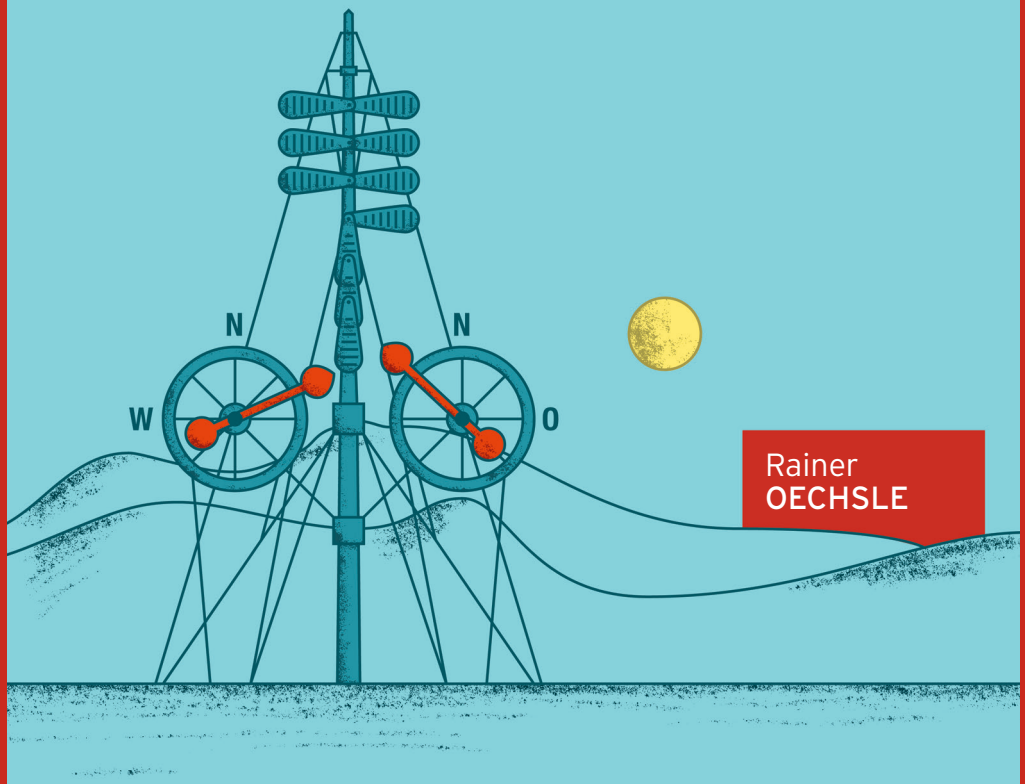
6., überarbeitete und erweiterte Auflage



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.

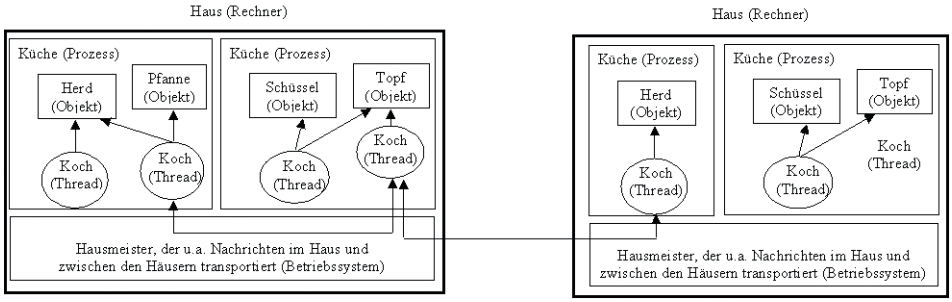


Bild 1.1 Häuser, Küchen, Köche und Hausmeister als Metapher für Rechner, Prozesse, Threads und Betriebssysteme

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)
```