



5., neu bearbeitete Auflage

# PARALLELE UND VERTEILTE ANWENDUNGEN IN Java

HANSER

Oechsle  
**Parallele und verteilte Anwendungen in Java**

## Bleiben Sie auf dem Laufenden!



Unser **Computerbuch-Newsletter** informiert Sie monatlich über neue Bücher und Termine. Profitieren Sie auch von Gewinnspielen und exklusiven Leseproben. Gleich anmelden unter



[www.hanser-fachbuch.de/newsletter](http://www.hanser-fachbuch.de/newsletter)



**Hanser Update** ist der IT-Blog des Hanser Verlags mit Beiträgen und Praxistipps von unseren Autoren rund um die Themen Online Marketing, Webentwicklung, Programmierung, Softwareentwicklung sowie IT- und Projektmanagement. Lesen Sie mit und abonnieren Sie unsere News unter



[www.hanser-fachbuch.de/update](http://www.hanser-fachbuch.de/update)



# 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, Beispieldaten mit Lösungen und weitere aktuelle Inhalte auf eigenen Websites oder zum Buch gehörigen CD-ROMs zur Verfügung.

## Lieferbare Titel in dieser Reihe

- Rainer Oechsle, Parallelle und verteilte Anwendungen in Java
- Wolfgang Riggert, Rechnernetze: Grundlagen – Ethernet – Internet
- Georg Stark, Robotik mit MATLAB
- Rolf Socher, Theoretische Grundlagen der Informatik

Rainer Oechsle

# Parallele und verteilte Anwendungen in Java

5., neu bearbeitete Auflage

Mit 160 Listings, 5 Tabellen und 66 Bildern

HANSER

**Der Autor:**

Prof. Dr. Rainer Oechsle, Hochschule Trier

**Die Herausgeber:**

Prof. Dr. Michael Lutz, Prof. Dr. Christian Märtin, Hochschule Augsburg



Alle in diesem Buch enthaltenen Informationen, Verfahren und Darstellungen wurden nach bestem Wissen zusammengestellt und mit Sorgfalt getestet. Dennoch sind Fehler nicht ganz auszuschließen. Aus diesem Grund sind die im vorliegenden Buch enthaltenen Informationen mit keiner Verpflichtung oder Garantie irgendeiner Art verbunden. Autoren und Verlag übernehmen infolgedessen keine juristische Verantwortung und werden keine daraus folgende oder sonstige Haftung übernehmen, die auf irgendeine Art aus der Benutzung dieser Informationen – oder Teilen davon – entsteht.

Ebenso übernehmen Autoren und Verlag keine Gewähr dafür, dass beschriebene Verfahren usw. frei von Schutzrechten Dritter sind. Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Buch berechtigt deshalb auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürfen.

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.

ISBN: 978-3-446-45118-6

E-Book-ISBN: 978-3-446-45603-7

Dieses Werk ist urheberrechtlich geschützt.

Alle Rechte, auch die der Übersetzung, des Nachdruckes und der Vervielfältigung des Buches, oder Teilen daraus, vorbehalten. Kein Teil des Werkes darf ohne schriftliche Genehmigung des Verlages in irgendeiner Form (Fotokopie, Mikrofilm oder ein anderes Verfahren), auch nicht für Zwecke der Unterrichtsgestaltung – mit Ausnahme der in den §§ 53, 54 URG genannten Sonderfälle –, reproduziert oder unter Verwendung elektronischer Systeme verarbeitet, vervielfältigt oder verbreitet werden.

© 2018 Carl Hanser Verlag München

Internet: <http://www.hanser-fachbuch.de>

Lektorat: Mirja Werner, M.A., Dipl.-Ing. Natalia Silakova-Herzberg

Herstellung: Dipl.-Ing. (FH) Franziska Kaufmann

Satz: Kösel Media GmbH, Krugzell

Coverconcept: Marc Müller-Bremer, [www.rebranding.de](http://www.rebranding.de), München

Coverrealisierung: Stephan Rönigk

Druck und Bindung: Hubert & Co., Göttingen

Printed in Germany

# Vorwort zur 5. 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 sechs 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 mehrfach 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 CompletableFuture 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 mit Hilfe 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

## 6 Vorwort zur 5. Auflage

das MVP-Architekturmuster (MVP: Model – View – Presenter) lassen sich auf verteilte Anwendungen übertragen.

- Entwicklung verteilter Anwendungen in Java (Kapitel 5, 6 und 7): 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 7) 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.

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 bezüglich der Synchronisation falsch sind, was beim 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 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 verteilte Programmierung behandle ich auch in dieser Auflage wieder RMI sehr intensiv. Man mag der Meinung sein, dass RMI inzwischen veraltet ist, aber aus meiner Sicht ist RMI immer noch 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. Viele der vorgestellten Konzepte finden sich in anderen Umgebungen wieder. Dies gilt insbesondere für C# und .NET.

Für diese fünfte Auflage wurden neben der Korrektur von Fehlern, die in der vierten Auflage bemerkt wurden, folgende Änderungen vorgenommen:

- Die mit Java 8 eingeführten Lambda-Ausdrücke werden zu Beginn von Kapitel 2 erläutert und dann an vielen Stellen im Buch genutzt.
- Das ebenfalls in Java 8 eingeführte Data-Streaming-Framework wird in dem vollständig neu geschriebenen Abschnitt 3.9 vorgestellt.
- Der neue Abschnitt 3.10 widmet sich den CompletableFuture, die seit Java 8 verfügbar sind.

- Es erfolgte eine Umstellung von Swing auf die modernere Oberflächenbibliothek JavaFX. Dies hat zur Folge, dass das alte Kapitel 4 vollständig ersetzt wurde. Alle Anwendungen in den folgenden Kapiteln, in denen Swing verwendet wurde, wurden entsprechend auf JavaFX umgestellt. Auch habe ich mich dazu entschlossen, statt des Architekturmusters MVC (Model – View – Controller) nun MVP (Model – View – Presenter) zu verwenden. Alle konzeptionellen Überlegungen in den folgenden Kapiteln, die auf MVC basierten, wurden auf MVP übertragen. Dies gilt vor allem für die Kapitel 6 und 7.
- Da JSP nicht mehr unterstützt wird, wurde der Abschnitt 7.8 auf JSF umgestellt und somit komplett neu geschrieben. Das Thema AJAX, dem in der vorhergehenden Auflage ein eigener Abschnitt (7.10 in Auflage 4) gewidmet war, wird nun vollständig im Kontext von JSF behandelt. Die Nutzung von AJAX wird damit extrem stark vereinfacht.
- Der Abschnitt 7.9, der ebenfalls vollständig neu entwickelt wurde, behandelt RESTful WebServices.
- Der Abschnitt 7.10 über WebSockets (in der vierten Auflage wurde dieses Thema in Abschnitt 7.11 besprochen) wurde überarbeitet.

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. Beachten Sie aber bitte, dass in der elektronischen Version, die Sie von der Webseite dieses Buches puva.hochschule-trier.de (puva: parallele und verteilte Anwendungen) 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 8 (genauer: 1.8.0\_65) übersetzt und ausprobiert. Die Servlets und JSF-Anwendungen wurden auf einem Tomcat-Server der Version 8 (genauer: 8.0.30) probeweise ausgeführt.

Von der soeben bereits erwähnten Webseite puva.hochschule-trier.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 Entdecker wie für die vorhergehende Auflage auf dieser Seite veröffentlichen. Ich habe zwar für diese Auflage alle entdeckten Fehler korrigiert, aber es ist sehr wahrscheinlich, dass bisher unentdeckte alte Fehler noch zu Tage treten werden, und 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 Leserkreis 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 aber nur die männliche und an wieder anderen Stellen 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 Programmietechniken (FOPT)“ im Rahmen des Informatik-

## 8 Vorwort zur 5. Auflage

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 unter [fernstudium.hochschule-trier.de](http://fernstudium.hochschule-trier.de).

Diese fünfte Auflage wäre ohne die Hilfe der nachfolgend genannten Personen nicht bzw. nicht in dieser Form möglich gewesen. Ich bedanke mich daher gerne

- bei der für dieses Buch verantwortlichen Lektorin des Hanser-Verlags, Frau Mirja Werner, für die Möglichkeit, dass das Buch in fünfter Auflage erscheinen kann sowie für die Erlaubnis, dass ich das Buch in moderatem Umfang erweitern durfte;
- bei Karl-Heinz Claas, Natalja Dyck, Niko Ehlen, Marc Fröwis, Albrecht Scholl, Thomas Zehrer und Veit Zoche-Golob für ihre Hinweise auf entdeckte Fehler und ihre Verbesserungsvorschläge, die alle auf der Webseite [puva.hochschule-trier.de](http://puva.hochschule-trier.de) veröffentlicht und in dieser fünften Auflage berücksichtigt wurden;
- 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](mailto:oechsle@hochschule-trier.de).

Konz-Oberemmel, im Januar 2018

*Rainer Oechsle*

# Inhalt

<b>1</b>	<b>Einleitung</b>	<b>15</b>
1.1	Parallelität, Nebenläufigkeit und Verteilung	15
1.2	Programme, Prozesse und Threads	16
<b>2</b>	<b>Grundlegende Synchronisationskonzepte in Java</b>	<b>20</b>
2.1	Erzeugung und Start von Java-Threads	20
2.1.1	Ableiten der Klasse Thread	20
2.1.2	Implementieren der Schnittstelle Runnable	22
2.1.3	Einige Beispiele	25
2.2	Probleme beim Zugriff auf gemeinsam genutzte Objekte	31
2.2.1	Erster Lösungsversuch	35
2.2.2	Zweiter Lösungsversuch	36
2.3	Synchronized und volatile	38
2.3.1	Synchronized-Methoden	38
2.3.2	Synchronized-Blöcke	39
2.3.3	Wirkung von synchronized	41
2.3.4	Notwendigkeit von synchronized	42
2.3.5	Volatile	43
2.3.6	Regel für die Nutzung von synchronized	44
2.4	Ende von Java-Threads	45
2.4.1	Asynchrone Beauftragung mit Abfragen der Ergebnisse	46
2.4.2	Zwangswise Beenden von Threads	52
2.4.3	Asynchrone Beauftragung mit befristetem Warten	57
2.4.4	Asynchrone Beauftragung mit Rückruf (Callback)	59
2.4.5	Asynchrone Beauftragung mit Rekursion	62
2.5	Wait und notify	65
2.5.1	Erster Lösungsversuch	66
2.5.2	Zweiter Lösungsversuch	67
2.5.3	Dritter Lösungsversuch	68
2.5.4	Korrekte und effiziente Lösung mit wait und notify	69

2.6	NotifyAll .....	77
2.6.1	Erzeuger-Verbraucher-Problem mit wait und notify .....	78
2.6.2	Erzeuger-Verbraucher-Problem mit wait und notifyAll .....	82
2.6.3	Faires Parkhaus mit wait und notifyAll .....	84
2.7	Prioritäten von Threads .....	86
2.8	Thread-Gruppen .....	93
2.9	Vordergrund- und Hintergrund-Threads .....	98
2.10	Weitere „gute“ und „schlechte“ Thread-Methoden .....	99
2.11	Thread-lokale Daten .....	101
2.12	Zusammenfassung .....	103
<b>3</b>	<b>Fortgeschrittene Synchronisationskonzepte in Java .....</b>	<b>108</b>
3.1	Semaphore .....	109
3.1.1	Einfache Semaphore .....	109
3.1.2	Einfache Semaphore für den gegenseitigen Ausschluss .....	110
3.1.3	Einfache Semaphore zur Herstellung vorgegebener Ausführungsreihenfolgen .....	112
3.1.4	Additive Semaphore .....	115
3.1.5	Semaphorgruppen .....	118
3.2	Message Queues .....	121
3.2.1	Verallgemeinerung des Erzeuger-Verbraucher-Problems .....	121
3.2.2	Übertragung des erweiterten Erzeuger-Verbraucher-Problems auf Message Queues .....	123
3.3	Pipes .....	126
3.4	Philosophen-Problem .....	129
3.4.1	Lösung mit synchronized - wait - notifyAll .....	130
3.4.2	Naive Lösung mit einfachen Semaphoren .....	132
3.4.3	Einschränkende Lösung mit gegenseitigem Ausschluss .....	134
3.4.4	Gute Lösung mit einfachen Semaphoren .....	135
3.4.5	Lösung mit Semaphorgruppen .....	138
3.5	Leser-Schreiber-Problem .....	140
3.5.1	Lösung mit synchronized - wait - notifyAll .....	141
3.5.2	Lösung mit additiven Semaphoren .....	145
3.6	Schablonen zur Nutzung der Synchronisationsprimitive und Konsistenzbetrachtungen .....	146
3.7	Concurrent-Klassenbibliothek aus Java 5 .....	150
3.7.1	Executors .....	151
3.7.2	Locks und Conditions .....	157
3.7.3	Atomic-Klassen .....	165
3.7.4	Synchronisationsklassen .....	169
3.7.5	Queues .....	172

3.8	Das Fork-Join-Framework von Java 7 .....	173
3.8.1	Grenzen von ThreadPoolExecutor .....	173
3.8.2	ForkJoinPool und RecursiveTask .....	175
3.8.3	Beispiel zur Nutzung des Fork-Join-Frameworks .....	177
3.9	Das Data-Streaming-Framework von Java 8 .....	180
3.9.1	Einleitendes Beispiel .....	180
3.9.2	Sequenzielles Data-Streaming .....	182
3.9.3	Paralleles Data-Streaming .....	186
3.10	Die CompletableFuture von Java 8 .....	187
3.11	Ursachen für Verklemmungen .....	194
3.11.1	Beispiele für Verklemmungen mit synchronized .....	195
3.11.2	Beispiele für Verklemmungen mit Semaphoren .....	198
3.11.3	Bedingungen für das Eintreten von Verklemmungen .....	199
3.12	Vermeidung von Verklemmungen .....	200
3.12.1	Anforderung von Betriebsmitteln „auf einen Schlag“ .....	203
3.12.2	Anforderung von Betriebsmitteln gemäß einer vorgegebenen Ordnung .....	204
3.12.3	Weitere Verfahren .....	205
3.13	Zusammenfassung .....	207

## **4 Parallelität und grafische Benutzeroberflächen .....** **209**

4.1	Einführung in die Programmierung grafischer Benutzeroberflächen mit JavaFX .....	210
4.1.1	Allgemeines zu grafischen Benutzeroberflächen .....	210
4.1.2	Erstes JavaFX-Beispiel .....	211
4.1.3	Ereignisbehandlung .....	212
4.2	Properties, Bindings und JavaFX-Collections .....	216
4.2.1	Properties .....	216
4.2.2	Bindings .....	219
4.2.3	JavaFX-Collections .....	221
4.3	Elemente von JavaFX .....	221
4.3.1	Container .....	221
4.3.2	Interaktionselemente .....	224
4.3.3	Grafikprogrammierung .....	226
4.3.4	Weitere Funktionen von JavaFX .....	232
4.4	MVP .....	233
4.4.1	Prinzip von MVP .....	234
4.4.2	Beispiel zu MVP .....	235
4.5	Threads und JavaFX .....	242
4.5.1	Threads für JavaFX .....	242
4.5.2	Länger dauernde Ereignisbehandlungen .....	243

4.5.3 Beispiel Stoppuhr .....	248
4.5.4 Tasks und Services in JavaFX .....	254
4.6 Zusammenfassung .....	263
<b>5 Verteilte Anwendungen mit Sockets .....</b>	<b>264</b>
5.1 Einführung in das Themengebiet der Rechnernetze .....	265
5.1.1 Schichtenmodell .....	265
5.1.2 IP-Adressen und DNS-Namen .....	269
5.1.3 Das Transportprotokoll UDP .....	270
5.1.4 Das Transportprotokoll TCP .....	271
5.2 Socket-Schnittstelle .....	272
5.2.1 Socket-Schnittstelle zu UDP .....	272
5.2.2 Socket-Schnittstelle zu TCP .....	274
5.2.3 Socket-Schnittstelle für Java .....	276
5.3 Kommunikation über UDP mit Java-Sockets .....	277
5.4 Multicast-Kommunikation mit Java-Sockets .....	286
5.5 Kommunikation über TCP mit Java-Sockets .....	290
5.6 Sequenzielle und parallele Server .....	300
5.6.1 TCP-Server mit dynamischer Parallelität .....	301
5.6.2 TCP-Server mit statischer Parallelität .....	305
5.6.3 Sequenzieller, „verzahnt“ arbeitender TCP-Server .....	310
5.7 Zusammenfassung .....	314
<b>6 Verteilte Anwendungen mit RMI .....</b>	<b>315</b>
6.1 Prinzip von RMI .....	315
6.2 Einführendes RMI-Beispiel .....	318
6.2.1 Basisprogramm .....	318
6.2.2 RMI-Client mit grafischer Benutzeroberfläche .....	322
6.2.3 RMI-Registry .....	327
6.3 Parallelität bei RMI-Methodenaufrufen .....	331
6.4 Wertübergabe für Parameter und Rückgabewerte .....	335
6.4.1 Serialisierung und Deserialisierung von Objekten .....	336
6.4.2 Serialisierung und Deserialisierung bei RMI .....	341
6.5 Referenzübergabe für Parameter und Rückgabewerte .....	345
6.6 Transformation lokaler in verteilte Anwendungen .....	360
6.6.1 Rechnergrenzen überschreitende Synchronisation mit RMI .....	360
6.6.2 Asynchrone Kommunikation mit RMI .....	363
6.6.3 Verteilte MVP-Anwendungen mit RMI .....	364
6.7 Dynamisches Umschalten zwischen Wert- und Referenzübergabe – Migration von Objekten .....	365
6.7.1 Das Exportieren und „Unexportieren“ von Objekten .....	365

6.7.2	Migration von Objekten . . . . .	368
6.7.3	Eintrag eines Nicht-Stub-Objekts in die RMI-Registry . . . . .	375
6.8	Laden von Klassen über das Netz . . . . .	376
6.9	Realisierung von Stubs und Skeletons . . . . .	377
6.9.1	Realisierung von Skeletons . . . . .	378
6.9.2	Realisierung von Stubs . . . . .	378
6.10	Verschiedenes . . . . .	381
6.11	Zusammenfassung . . . . .	382
<b>7</b>	<b>Webbasierte Anwendungen mit Servlets und JSF . . . . .</b>	<b>383</b>
7.1	HTTP und HTML . . . . .	384
7.1.1	GET . . . . .	384
7.1.2	Formulare in HTML . . . . .	387
7.1.3	POST . . . . .	389
7.1.4	Format von HTTP-Anfragen und -Antworten . . . . .	390
7.2	Einführende Servlet-Beispiele . . . . .	391
7.2.1	Allgemeine Vorgehensweise . . . . .	391
7.2.2	Erstes Servlet-Beispiel . . . . .	392
7.2.3	Zugriff auf Formulardaten . . . . .	394
7.2.4	Zugriff auf die Daten der HTTP-Anfrage und -Antwort . . . . .	396
7.3	Parallelität bei Servlets . . . . .	397
7.3.1	Demonstration der Parallelität von Servlets . . . . .	397
7.3.2	Paralleler Zugriff auf Daten . . . . .	399
7.3.3	Anwendungsglobale Daten . . . . .	403
7.4	Sessions und Cookies . . . . .	406
7.4.1	Sessions . . . . .	407
7.4.2	Realisierung von Sessions mit Cookies . . . . .	411
7.4.3	Direkter Zugriff auf Cookies . . . . .	413
7.4.4	Servlets mit länger dauernden Aufträgen . . . . .	414
7.5	Asynchrone Servlets . . . . .	420
7.6	Filter . . . . .	424
7.7	Übertragung von Dateien mit Servlets . . . . .	425
7.7.1	Herunterladen von Dateien . . . . .	425
7.7.2	Hochladen von Dateien . . . . .	428
7.8	JSF (Java Server Faces) . . . . .	431
7.8.1	Einführendes Beispiel . . . . .	431
7.8.2	Managed Beans und deren Scopes . . . . .	438
7.8.3	MVP-Prinzip mit JSF . . . . .	442
7.8.4	AJAX mit JSF . . . . .	443
7.9	RESTful WebServices . . . . .	447
7.9.1	Definition von RESTful WebServices . . . . .	448

7.9.2 JSON .....	449
7.9.3 Beispiel .....	451
7.10 WebSockets .....	456
7.11 Zusammenfassung .....	460
<b>Literatur .....</b>	<b>463</b>
<b>Index .....</b>	<b>465</b>

# 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, sind heute auch nahezu alle den Umgang mit verteilten Anwendungen wie der elektronischen Post 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 erklärt.

## ■ 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.

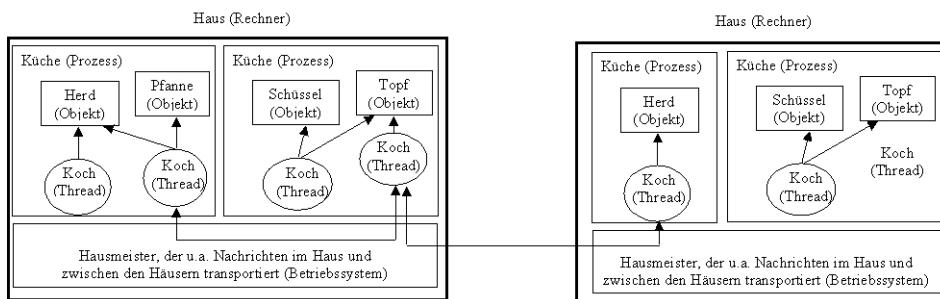
## ■ 1.2 Programme, Prozesse und Threads

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 Häusern 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 mit Hilfe 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 Pro-

grammcode 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, so dass 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 so genannten 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 oder über RMI (Remote Method Invocation). 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.

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 mit Hilfe 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 selbstständig

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 übersetzbare 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. Abschnitte 2.2 und 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 Methodename 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, so dass 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 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 Platzsparends 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)

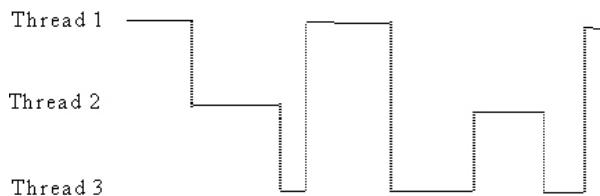
```

```

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 (wovon wir im Folgenden ausgehen), dann wird dies durch das automatische Umschalten zwischen den verschiedenen Threads realisiert, so dass sich der Eindruck eines parallelen Ablaufs ergibt. Dabei ist die Anzahl der Schleifendurchläufe, die ein Thread durchführen kann, bevor auf einen anderen Thread umgeschaltet wird, nicht immer gleich. Wie Sie anhand der Ausgabe sehen können, kann der erste Thread seine Schleife 46 Mal durchlaufen. Danach wird auf den zweiten Thread umgeschaltet. Diesem wird aber nur ein einziger Schleifendurchlauf 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. Man sieht, welcher der Threads zu einem bestimmten Zeitpunkt ausgeführt wird. 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

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.

Das Attribut name ist nicht nötig, da die Klasse Thread bereits ein solches String-Attribut für den Namen des Threads besitzt. Der Wert des Namens-Attribut kann im Konstruktor als Argument angegeben werden und später durch die Methode `setName` verändert werden. Mit Hilfe 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 10-mal 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 8 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 Ausgabeausweisung 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.

## ■ 2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte

In den bisherigen Beispielen arbeiten die einzelnen Threads weitgehend unabhängig von einander, 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, so dass 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“ mit Hilfe 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 Account[] account;

    public Bank()
    {
        account = new Account[100];
        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
```

## 2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte

```

{
    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.

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 just in diesem Augenblick auf den anderen Thread umgeschaltet wird. Petra Schmitt führt nun mehrere Buchungen durch, u. a. sollen 1000 € 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 1000 ausgeführt. Dort wird nun zunächst der aktuelle Kontostand, der natürlich immer noch 0 ist, gelesen. Anschließend wird der neue Stand zu 1000 berechnet und entsprechend gesetzt. Petra Schmitt führt nun noch weitere Buchungen auf anderen Konten durch. Irgendwann wird wieder auf den Thread von Andrea Müller geschaltet. 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 1000 €, die Petra Schmitt ausgeführt hat, ist verloren gegangen; der aktuelle Kontostand beträgt -100 € statt +900 €. 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 im Abschnitt 2.3 werden Sie eine korrekte Lösung des geschilderten Problems kennen lernen.

### 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 wieder hergestellt. 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 mit Hilfe 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 anzugeben, 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 grundsätzlich zu vermeiden.
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 durch-

führen, 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 Inkorrekttheit (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 kann dennoch zwischen dem Aufruf der beiden Methoden getBalance und setBalance umgeschaltet 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)
        {
            ...
        }
    }
}
```

### 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 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 so genannte 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 zurück bekommt, 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 fehlgeschlagen – 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 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 dem 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 im 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 Nano sekunden 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.

## 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 = 1000000000;
    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) / 1000.0f;
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. Feldelement auf true gesetzt wird (alternativ könnte man z. B. mit Hilfe 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 Auftragsteilung 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	0,391
2	0,251
3	0,203

**Tabelle 2.1** Laufzeiten des Programms AsynchRequest (*Fortsetzung*)

NUMBER_OF_SERVERS	Gemessene Zeit (in Sekunden)
4	0,187
5	0,188
10	0,203
50	0,204
100	0,203
1.000	0,312
10.000	0,984
100.000	10,727

Wie man sieht, ergibt sich eine deutliche Reduktion der Laufzeit bei der Benutzung von zwei Threads gegenüber einem Thread. Weitere Laufzeitverkürzungen, 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 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, so dass das Starten von 100.000 Threads deutlich aufwändiger 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. 1.000 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
1.000	0,266
5.000	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:

$$10.000 \times 10 \text{ ms} = 100.000 \text{ ms} = 100 \text{ s.}$$

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 1.000 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.

## 2.4.2 Zwangsweises Beenden von Threads

In manchen Fällen soll es möglich sein, das Beenden eines Threads „von außen“ anzustößen. 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 des vorigen Abschnitts 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

nur gelesen wird wie in der Methode `isStopped` oder ob das Attribut verändert wird wie in der Methode `stopThread`. Wie erwähnt sind zwar lesende und schreibende Zugriffe auf ein Attribut des Typs `boolean` atomar, aber wegen möglicher Compiler-Optimierungen müsste man das Attribut zumindest als `volatile` kennzeichnen. Da wir aber `volatile` in diesem Buch nicht verwenden, haben wir die Zugriffe entsprechend synchronisiert.

Das Programm gibt einem Thread 5 Sekunden Zeit (genauer: mindestens 5 Sekunden, s. Erläuterungen zu der Messreihe von Tabelle 2.2), wiederholt „Hallo Welt“ mit einer Nummerierung auszugeben. Eine solche Ausgabe könnte so aussehen:

```
Hallo Welt (1)
...
Hallo Welt (144026)
Thread endet jetzt ...
```

Diese Ausgabe ist in der Regel von Ausführung zu Ausführung unterschiedlich: Bei wiederholter Ausführung dieses Programms ergeben sich unterschiedliche Anzahlen von Ausgabezeilen. Auch hängt die Ausgabe sehr stark von der Größe des Fensters ab; bei kleineren Fenstern ist die Anzahl der ausgegebenen Zeilen größer als bei größeren Fenstern. Dies ist dadurch zu erklären, dass bei der Ausgabe einer Zeile das gesamte Fenster neu auf den Bildschirm gezeichnet werden muss. Ein großes Fenster neu zu zeichnen ist aufwändiger als ein kleines Fenster neu zu zeichnen und dauert daher länger. Noch mehr Zeilen werden ausgegeben, wenn das Programmfenster direkt nach dem Start des Programms zu einem Icon verkleinert wird (auf meinem PC über 180.000 statt wie zuvor ca. 144.000 Zeilen).

Allerdings ist es nicht notwendig, ein Attribut wie `stopped` in Listing 2.16 selbst zu definieren. Die Klasse `Thread` besitzt nämlich bereits ein solches Attribut (diese Situation ist ähnlich wie zuvor beim Namensattribut, das wir in Listing 2.3 selbst definiert haben, bevor wir dann in Listing 2.4 das Namensattribut der Klasse `Thread` verwendet haben). Das Attribut wird als *Interrupt-Flag* bezeichnet. Mit der Methode `isInterrupted` (entspricht `isStopped`) kann das Interrupt-Flag abgefragt und mit der Methode `interrupt` (entspricht `stopThread`) auf `true` gesetzt werden. Die Verwendung des Interrupt-Flags eines Threads hat aber nicht nur den Vorteil, dass man selbst kein Attribut und keine eigenen Methoden definieren muss. Mit dem Interrupt-Flag ist es zusätzlich möglich, einen Thread aus einem Zustand, in dem der Thread nicht laufen kann (z.B. weil er sich gerade in `sleep`, `join` oder `wait` [s. Abschnitt 2.5] befindet), herauszuholen, die Blockade eines Threads also sozusagen zu unterbrechen (daher der Name Interrupt-Flag).

Wir ändern das Programm aus Listing 2.16 nun dahingehend, dass wir statt dem selbst definierten Attribut `stopped` das Interrupt-Flag benutzen. Um den Unterbrechungseffekt zusätzlich zu demonstrieren, lassen wir den Thread, den wir beenden wollen, wiederholt 3 Sekunden in `sleep` schlafen (siehe Listing 2.17).

### **Listing 2.17**

```
public class StopThreadByInterrupt extends Thread
{
    public StopThreadByInterrupt()
    {
        start();
    }
}
```

```

public void run()
{
    int i = 0;
    try
    {
        while(!isInterrupted())
        {
            i++;
            System.out.println("Hallo Welt (" + i + ")");
            Thread.sleep(3000);
        }
    }
    catch(InterruptedException e)
    {
        System.out.println(e.getMessage());
    }
    System.out.println("thread terminating ...");
}

public static void main(String[] args)
{
    StopThreadByInterrupt st = new StopThreadByInterrupt();
    try
    {
        Thread.sleep(9100);
    }
    catch(InterruptedException e)
    {
    }
    st.interrupt();
}
}

```

Hätten wir dieses sleep auch in die Run-Methode von Listing 2.16 eingebaut, so hätte es im ungünstigsten Fall sein können, dass wir nach dem Aufruf der Methode stopThread noch 3 Sekunden warten müssen, bis der Thread tatsächlich zu Ende gelaufen wäre (nämlich dann, wenn stopThread ausgeführt worden wäre, kurz nachdem der andere Thread sein Attribut stopped abgefragt hatte). Beim Programm aus Listing 2.17 wird das Programm immer sehr schnell beendet, denn das Setzen des Interrupt-Flags unterbricht die Sleep-Methode mit einer Ausnahme des Typs *InterruptedException*. Dabei ist es unerheblich, ob das Interrupt-Flag gesetzt wird, wenn der Thread bereits im Schlafenzustand ist, oder ob das Interrupt-Flag schon vorher gesetzt war. Wenn das Interrupt-Flag schon gesetzt war, dann wird unmittelbar beim Aufruf von sleep die Ausnahme ausgelöst.

In der Run-Methode von Listing 2.17 ist die While-Schleife in den Try-Block geschachtelt. Die umgekehrte Schachtelung (d.h. try-catch innerhalb der While-Schleife) würde übrigens in den meisten Fällen nicht funktionieren, denn mit dem Behandeln der Ausnahme *InterruptedException* (d.h. dem „Fangen“ im Catch-Block) wird das Interrupt-Flag wieder zurück (d.h. auf false) gesetzt. Somit würde nach Ausführung des Catch-Blocks die erneute Abfrage des Interrupt-Flags durch die Methode *isInterrupted* wieder false liefern; die Schleife und damit die Run-Methode würden nicht beendet werden.

Das Abfragen des Interrupt-Flags verändert den Wert des Flags nicht. Es gibt allerdings auch die Static-Methode *interrupted* zum Abfragen des Interrupt-Flags des ausführenden

Threads. Dabei wird das Interrupt-Flag allerdings zurückgesetzt. Damit haben wir also drei weitere Methoden der Klasse Thread kennengelernt:

```
public class Thread
{
    public void interrupt() {...}
    public boolean isInterrupted() {...}
    public static boolean interrupted() {...}
    ...
}
```

Zur Sicherheit sei noch erwähnt, dass man natürlich in dem Programm aus Listing 2.17 nicht davon ausgehen kann, dass unmittelbar nach Ausführung der Methode interrupt der unterbrochene Thread beendet ist. Denn dieser unterbrochene Thread muss ja selbst zu Ende laufen. Das heißt, dass ihm erst wieder der Prozessor vom Betriebssystem zugewiesen werden muss, damit er laufen kann, und er dann seine restlichen Anweisungen vollends ausführen und seine Run-Methode beenden muss. Dies demonstriert das Beispielprogramm aus Listing 2.18:

### **Listing 2.18**

```
public class Polling extends Thread
{
    public Polling()
    {
        start();
    }

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

    public static void main(String[] args)
    {
        Polling p = new Polling();
        try
        {
            Thread.sleep(5000);
        }
        catch(InterruptedException e)
        {
        }
        System.out.println("isAlive liefert: " + p.isAlive());
        int i = 0;
        p.interrupt();
        while(p.isAlive())
        {
            i++;
            System.out.println("Thread lebt immer noch");
        }
    }
}
```

```
        System.out.println("Thread lebte noch " + i +  
                           " Durchläufe/Durchlauf");  
    }  
}
```

In diesem Programm wird nach Ausführung der Methode interrupt in einer While-Schleife mit isAlive auf das Ende des unterbrochenen Threads gewartet. Auf das Ende eines Threads sollte man im Normalfall natürlich mit join warten. Das aktive Warten wird hier lediglich zur Demonstration verwendet, um zu zeigen, dass der unterbrechende Thread noch mehrere Anweisungen ausführen kann, bevor der unterbrochene Thread wirklich zu Ende ist. Eine Ausführung des Programms aus Listing 2.18 könnte folgende Ausgabe produzieren:

```
Hallo Welt (1)  
...  
Hallo Welt (274087)  
isAlive liefert: true  
Thread lebt immer noch  
Thread lebt immer noch  
...  
Hallo Welt (274088)  
...  
Thread lebt immer noch  
Thread lebte noch 16 Durchläufe/Durchlauf
```

Der unterbrechende Thread konnte seine While-Schleife nach dem Interrupt-Aufruf also noch 16 Mal ausführen, bis der unterbrochene Thread wirklich zu Ende war. Entfernt man die Ausgabe in der While-Schleife, so ergeben sich deutlich mehr Schleifendurchläufe (ca. 700 in der Regel, in einem Extremfall habe ich sogar einmal mehr als 6.000 Schleifendurchläufe beobachtet).

Selbstverständlich kann das Interrupt-Flag nicht nur wie hier gezeigt zum Beenden von Threads eingesetzt werden, sondern auch in beliebiger anderer Weise. Wenn z.B. in der obigen Run-Methode nach der While-Schleife weiterer Programmcode folgt, so würde ein Aufruf von interrupt bewirken, dass der Thread zur Ausführung dieses Programmcodes übergeht.

### 2.4.3 Asynchrone Beauftragung mit befristetem Warten

Durch die Möglichkeit, laufende Threads abzubrechen, kann man nun einem Thread, den man zur Lösung einer Aufgabe beauftragt hat, eine begrenzte Zeit zur Erledigung der Aufgabe einräumen. Ist der Thread dann noch nicht zu Ende, dann wird er abgebrochen. Dies kann zum Beispiel angewendet werden, wenn der Thread ein gewisses Ergebnis näherungsweise berechnet und diese Berechnung ständig verfeinert und damit genauer wird. Man kann so einem Thread eine gewisse Zeit zur Berechnung gewähren und dann das Ergebnis mit der Genauigkeit, die in der verfügbaren Zeit erreicht werden konnte, benutzen. Das Schema eines solchen Programms ist im folgenden Beispiel (Listing 2.19) gezeigt:

**Listing 2.19**

```
class Server extends Thread
{
    private double result;

    public Server(...)
    {
        ...
    }

    public double getResult()
    {
        return result;
    }

    public void run()
    {
        boolean morePrecisionPossible = true;
        while(!isInterrupted() && morePrecisionPossible)
        {
            // nächste Iteration: result genauer berechnen
            ...
            if(...)
            {
                morePrecisionPossible = false;
            }
        }
    }

    public class BoundedTime
    {
        private static final int TIMEOUT = 10000;

        public static void main(String[] args)
        {
            Server server = new Server(...);
            server.start();

            // auf Serverende beschränkte Zeit warten
            try
            {
                server.join(TIMEOUT);
            }
            catch(InterruptedException e)
            {
            }

            // Server sanft abbrechen
            server.interrupt();

            // Synchronisation mit Server
            try
            {
                server.join();
            }
            catch(InterruptedException e)
            {
            }
        }
    }
}
```

```

    }

    // Ergebnis abholen und ausgeben
    double result = server.getResult();
    System.out.println("Ergebnis: " + result);
}
}

```

In diesem Programm wird auf das Ende des Server-Threads eine bestimmte Zeit (10 Sekunden) gewartet. Danach wird der Server-Thread abgebrochen. Sollte der Server zu diesem Zeitpunkt schon zu Ende gelaufen sein, so hat dieses Abbrechen keine Wirkung. Anschließend muss nochmals auf das Ende des Threads gewartet werden, bevor das Resultat abgeholt werden kann, denn man möchte ja das vollständige zuletzt errechnete Resultat haben. Zur Erinnerung sei nochmals erwähnt, dass der Thread nicht sofort nach dem Abbrechen zu Ende läuft, sondern erst wieder an die Reihe kommen muss und dann auch noch einige Anweisungen ausführen muss, bevor er sich selbst beendet.

#### 2.4.4 Asynchrone Beauftragung mit Rückruf (Callback)

Bisher wurde in der Klasse `AsynchRequest` (Listing 2.14) auf das Ende der Server-Threads gewartet und danach wurden alle Ergebnisse abgefragt. Eine mögliche Alternative besteht darin, dass die Server-Threads von sich aus ihre Ergebnisse nach der Berechnung melden. Die Server rufen sozusagen zurück. Man bezeichnet dieses Verhalten daher als *Callback*. In einem solchen Fall muss der Auftraggeber nicht auf das Ende der Threads warten. Das Prinzip ist beispielhaft im folgenden Programm (Listing 2.20) dargestellt, das das frühere Beispiel `AsynchRequest` zum Zählen der True-Werte in einem booleschen Feld variiert.

**Listing 2.20**

```

interface ResultListener
{
    public void putResult(int result);
}

class ServiceCallback implements Runnable
{
    private boolean[] array;
    private int start;
    private int end;
    private ResultListener h;

    public ServiceCallback(boolean[] array, int start, int end,
                          ResultListener listener)
    {
        this.array = array;
        this.start = start;
        this.end = end;
        this.h = listener;
    }

    public void run()
{

```

```
int result = 0;
for(int i = start; i <= end; i++)
{
    if(array[i])
    {
        result++;
    }
}
h.putResult(result);
}

class ResultHandler implements ResultListener
{
    private int result;
    private int numberOfResults;
    private int expectedNumberOfResults;

    public ResultHandler(int r)
    {
        expectedNumberOfResults = r;
    }

    public synchronized void putResult(int r)
    {
        result += r;
        numberOfResults++;
        if(numberOfResults == expectedNumberOfResults)
        {
            System.out.println("Ergebnis: " + result);
        }
    }
}

public class AsynchRequestCallback
{
    private static final int ARRAY_SIZE = 500000000;
    private static final int NUMBER_OF_SERVERS = 10;

    public static void main(String[] args)
    {
        /*
         * Feld erzeugen, Werte sind abwechselnd true und false
         */
        boolean[] array = new boolean[ARRAY_SIZE];
        for(int i = 0; i < ARRAY_SIZE; i++)
        {
            if(i % 2 == 0)
            {
                array[i] = true;
            }
            else
            {
                array[i] = false;
            }
        }

        // ResultHandler erzeugen
    }
}
```

```
ResultHandler h = new ResultHandler(NUMBER_OF_SERVERS);

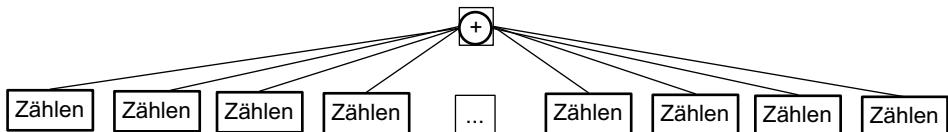
// Threads erzeugen
int start = 0;
int end;
int howMany = ARRAY_SIZE / NUMBER_OF_SERVERS;

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

In diesem Programm wird den Objekten der Klasse ServiceCallback neben den Argumenten für das boolesche Feld, dem Start- und Endeindex noch eine Referenz auf ein Objekt übergeben, das die Schnittstelle ResultListener implementiert. Diese Schnittstelle besitzt eine Methode namens putResult, mit der ein Resultat gemeldet werden kann. Die Klasse ServiceCallback enthält im Vergleich zu der Klasse Service die Methode getResult nicht mehr. Da das Resultat nicht mehr abgefragt werden kann, ist auch result kein Attribut mehr dieser Klasse. Stattdessen genügt es, das Resultat in der Methode run in einer lokalen Variablen zu berechnen und diesen Wert am Ende durch Aufruf der Methode putResult zu melden. Die Klasse ResultHandler implementiert die Schnittstelle ResultListener. Im Konstruktor wird angegeben, wie viele Resultatmeldungen erwartet werden. Die Methode putResult addiert das Argument zu dem bisherigen Ergebnis und zählt die Anzahl der gemeldeten Resultate um eins hoch. Sind alle erwarteten Resultatmeldungen eingetroffen, so wird bei der letzten Meldung das Gesamtergebnis ausgegeben. Die Main-Methode ist ähnlich wie zuvor in der Klasse AsynchRequest. Nur wird jetzt ein ResultHandler-Objekt erzeugt und allen ServiceCallback-Objekten im Konstruktor mitgegeben. Dafür ist es jetzt nach dem Starten der Threads nicht mehr nötig, auf das Ende der Threads zu warten und die Ergebnisse aufzusammeln, da jeder Thread sein Ergebnis von sich aus meldet. Folglich müssen auch die Service- und Thread-Objekte nicht mehr in Feldern gespeichert werden. Stattdessen ist jetzt aber eine Synchronisation beim Melden der Ergebnisse notwendig (die Methode putResult der Klasse ResultHandler ist synchronized).

## 2.4.5 Asynchrone Beauftragung mit Rekursion

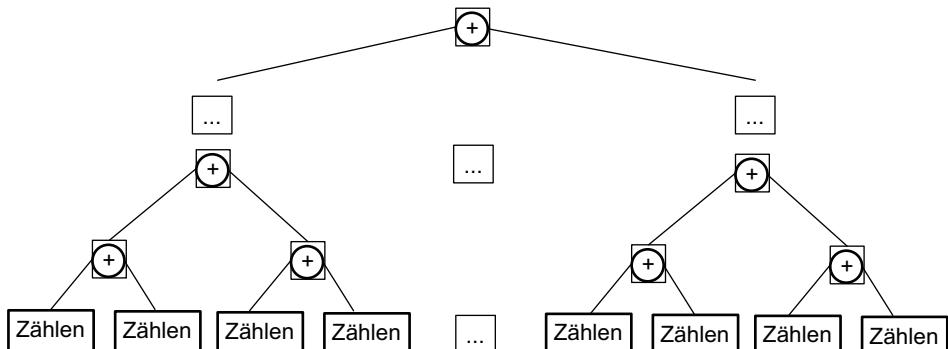
Bei unserem ursprünglichen Beispiel der asynchronen Beauftragung in Listing 2.14 gibt es den Nachteil, dass selbst dann, wenn sich ein sehr hoher Parallelisierungsgrad rentiert, das Kombinieren der Einzelergebnisse zu einem Gesamtergebnis in sequenzieller Weise erfolgt, nachdem alle Threads zu Ende gelaufen sind. Dies wird insbesondere dann zum Problem, wenn diese Kombination sich als aufwändig herausstellt. Was man durch die Parallelisierung gewinnt, könnte man durch die sequenzielle Kombination der Teilergebnisse zu einem Gesamtergebnis zumindest teilweise wieder verlieren. Die Situation ist in Bild 2.2 für das Zählen der True-Werte in einem booleschen Feld visualisiert.



**Bild 2.2** Paralleles Zählen mit sequenziellem Addieren

Bei der Variante der asynchronen Beauftragung mit Rückruf (Listing 2.20) existiert dieses Problem übrigens auch: Hier wird zwar die Kombination der Teilergebnisse nicht durch den Main-Thread nach dem Ende aller Threads durchgeführt, sondern von den Arbeiter-Threads selbst. Allerdings muss diese Kombination in synchronisierter Weise vorgenommen werden und erfolgt somit auch in diesem Fall sequenziell.

Eine Verbesserung dieser Situation kann man dadurch herbeiführen, dass man auch die Kombination der Teilergebnisse zu einem Gesamtergebnis parallelisiert. Wenn man diese Idee auf das Beispiel des Zählens der True-Werte in einem sehr großen Feld des Typs boolean anwendet, dann heißt dies, dass die Zählergebnisse von jeweils einigen wenigen Threads (zum Beispiel von je zwei Threads) parallel addiert werden, die Ergebnisse dieser Additionen können dann anschließend wiederum parallel addiert werden, bis irgendwann das Gesamtergebnis von einem einzigen Thread bestimmt wird, wobei dieser Thread dann aber auf die Vorarbeiten der anderen Threads zurückgreifen kann und somit nur noch relativ wenige Werte addieren muss. Wie die Leserinnen und Leser sicher erkannt haben, ergibt sich aus diesen Überlegungen, dass die Threads in einer Baumstruktur angeordnet sind, wobei sich an den Blättern des Baums die Threads für das Zählen befinden und die anderen Knoten des Baumes die Teilergebnisse ihrer Nachfolger addieren und als Eingabe für ihren Vorgänger zur Verfügung stellen (s. Bild 2.3).



**Bild 2.3** Paralleles Zählen mit parallelem Addieren

Im Folgenden soll diese Idee in ein Java-Programm umgesetzt werden, wobei folgende Entwurfsentscheidungen getroffen werden:

- Alle Knoten des Baums sind Threads.
- Die Anwendung soll so programmiert werden, dass es einen Teil gibt, der unabhängig von der konkreten Anwendung ist (also eine Art Framework), und einen anwendungsspezifischen Teil.
- Alle Knoten eines solchen Baums sollen vom selben Typ sein: Ob ein Knoten tatsächlich arbeitet (in unserem Beispiel True-Werte zählt) oder seinen Auftrag in Teilaufträge zerstückelt, an seine Nachfolger delegiert und deren Teilergebnisse dann kombiniert (in unserem Beispiel addiert), soll durch eine Fallunterscheidung im Code des Knotens vorgenommen werden.

Aus dem bisher Gesagten können wir damit eine erste Skizze in Form von Pseudocode für die Klasse der Baumknoten angeben:

```
public class TaskNodeExecutor extends Thread
{
    privates Attribut für Auftrag;
    privates Attribut für Ergebnis;

    Konstruktor: Auftrag durch Parameter setzen

    public void run()
    {
        if(Auftrag ist aufteilbar)
        {
            teile Auftrag in Teilaufträge auf;
            für jeden Teilauftrag
            {
                erzeuge TaskNodeExecutor;
                starte erzeugten TaskNodeExecutor;
            }
            für jeden erzeugten Thread
            {
                warte auf das Ende des Threads;
                hole Ergebnis von diesem Thread ab;
            }
            kombiniere Teilergebnisse zu Gesamtergebnis;
            speichere Gesamtergebnis in Attribut;
        }
        else
        {
            führe Auftrag selber aus;
            speichere Ergebnis in Attribut;
        }
    }

    Get-Methode für Ergebnis
}
```

Im obigen Pseudocode sind die Teile hervorgehoben, die anwendungsspezifisch sind. Im realen Code werden diese Teile durch Aufrufe entsprechender Methoden ersetzt, die in der Schnittstelle Task (Auftrag) zusammengefasst werden. Da in dieser Schnittstelle auch ein Typ für das Ergebnis benötigt wird, nutzen wir in diesem Fall das Generics-Konzept von

Java und geben in Listing 2.21 eine typparametrisierte Schnittstelle für Task an, wobei der Typparameter für den Ergebnistyp steht. Die Methoden sind im Stil des Pseudocodes von oben kommentiert.

### **Listing 2.21**

```
public interface Task<T>
{
    public boolean isDivisible(); //ist Auftrag teilbar?
    public List<Task<T>> split(); //teile Auftrag in mehrere
                                  //Teilaufträge auf
    public T combine(List<T> results); //kombiniere mehrere
                                       //Teilergebnisse zu einem
                                       //Gesamtergebnis
    public T execute(); //führe Auftrag selber aus
}
```

Unter Nutzung dieser Schnittstelle ist es damit einfach möglich, den oben angegebenen Pseudocode in realen Java-Code umzuwandeln (s. Listing 2.22). Wie Task hat auch die Klasse TaskNodeExecutor einen Typparameter für den Ergebnistyp.

### **Listing 2.22**

```
public class TaskNodeExecutor<T> extends Thread
{
    private Task<T> task;
    private T result;

    public TaskNodeExecutor(Task<T> task)
    {
        this.task = task;
    }

    public void run()
    {
        if(task.isDivisible())
        {
            List<Task<T>> subtasks = task.split();
            List<TaskNodeExecutor<T>> threads = new ArrayList<>();
            for(Task<T> subtask: subtasks)
            {
                TaskNodeExecutor<T> thread =
                    new TaskNodeExecutor<>(subtask);
                threads.add(thread);
                thread.start();
            }

            List<T> subresults = new ArrayList<>();
            for(TaskNodeExecutor<T> thread: threads)
            {
                try
                {
                    thread.join();
                }
                catch(InterruptedException e)
                {
                }
            }
        }
    }
}
```

```

        subresults.add(thread.getResult());
    }
    result = task.combine(subresults);
}
else //!task.isDivisible()
{
    result = task.execute();
}
}
public T getResult()
{
    return result;
}

public static <T> T executeAll(Task<T> task)
{
    TaskNodeExecutor<T> root = new TaskNodeExecutor<>(task);
    root.run();
    return root.getResult();
}
}

```

Die letzte Methode executeAll ist eine statische, generische Methode, die dafür verwendet werden soll, den Gesamtauftrag als Parameter zu übergeben und von dieser Methode das Gesamtergebnis als Rückgabewert zu erhalten. Diese Methode startet die Rekursion, indem sie den Wurzelknoten des Baums erzeugt. Da es auf der Wurzelebene keine Parallelität gibt, muss nicht unbedingt ein Thread erzeugt werden, auf dessen Ende man anschließend wartet. Stattdessen wird die Methode run für den Wurzelknoten von demselben Thread ausgeführt, der auch executeAll aufgerufen hat. Aufgrund des vorhergehenden Pseudocodes sollte der restliche Code in Listing 2.22 gut verständlich sein. Bitte beachten Sie, dass im obigen Programmcode kein synchronized vorkommt und auch nicht nötig ist. Die Begründung dafür ist dieselbe wie für das Programm in Listing 2.14 aus Abschnitt 2.4.1.

Um etwas Platz zu sparen, wird die Klasse, die die Schnittstelle Task für das Zählen der booleschen Werte implementiert, hier nicht angegeben. In den Beispielprogrammen, die Sie von der Webseite zu diesem Buch herunterladen können, finden Sie diese Klasse aber zusammen mit einem Hauptprogramm zum Ausprobieren.

## ■ 2.5 Wait und notify

Mit den bisher eingeführten Konzepten kann bereits eine ganze Reihe von Anwendungen mit Threads realisiert werden. Die gemeinsam benutzten Objekte kapseln ihre Zustände (die Werte der Attribute). Für diese Zustände gelten in der Regel gewisse *Konsistenzbedingungen (Invarianten)*. Für eine doppelt verkettete Liste z.B. ist eine Konsistenzbedingung, dass der Vorgänger des Nachfolgers eines Elements e dieses Element e ist. Mit den Methoden wird der Zustand eines Objekts von einem *konsistenten Zustand* in einen anderen konsistenten Zustand überführt, wobei zwischenzeitlich die *Konsistenz* verletzt sein kann. Bei einer doppelt verketteten Liste gilt die obige Bedingung vor und nach dem Einfügen eines

Elements. Allerdings werden während des Einfügens die Konsistenzbedingungen in der Regel verletzt sein, da der Einfügevorgang aus mehreren Einzelschritten besteht. Werden Objekte von mehreren Threads aus benutzt und dabei deren Zustände auch verändert, dann gewährleisten Synchronized-Methoden, dass ein Thread beim Aufruf einer Methode immer einen konsistenten Zustand des Objekts vorfindet. Wenn nämlich ein anderer Thread gerade dabei ist, den Zustand dieses Objekts zu verändern, dann ist dieses Objekt gesperrt und der Aufruf der Synchronized-Methode wird solange verzögert, bis der andere Thread die Änderung durchgeführt hat und damit der Zustand des Objekts wieder konsistent ist.

In manchen Situationen will man erreichen, dass gewisse Methoden nur dann ausgeführt werden können, wenn zusätzlich zu dem konsistenten Zustand gewisse weitere anwendungsabhängige Bedingungen erfüllt sind. Sind diese Bedingungen nicht erfüllt, so soll der Thread so lange warten, bis diese Bedingungen gelten. Anhand eines Beispiels aus dem täglichen Leben soll dieser Sachverhalt veranschaulicht werden. Wir wollen ein Parkhaus (ParkingGarage) modellieren, wobei wir das Parkhaus so weit abstrahieren, dass wir für den Zustand des Parkhauses lediglich die Anzahl der noch freien Parkplätze festhalten. Verschiedene Threads, welche die Autos darstellen, können die Methoden enter (Einfahrt in das Parkhaus) und leave (Ausfahrt aus dem Parkhaus) aufrufen. Beim Einfahren wird die Anzahl der freien Plätze um eins vermindert, beim Ausfahren entsprechend um eins erhöht. Allerdings kann die Anzahl der freien Plätze nicht um eins vermindert werden, wenn sie bereits null ist. Dieser Effekt dürfte den meisten Leserinnen bekannt sein; man kann in ein volles Parkhaus nicht mehr einfahren, da die Anzahl der freien Plätze nicht negativ werden kann. Da das Parkhaus von mehreren Auto-Threads benutzt wird und dabei der Zustand auch verändert wird, müssen die Methoden enter und leave beide synchronized sein.

In den folgenden drei Implementierungen werden zuerst wieder einmal vergebliche bzw. nicht ganz zufriedenstellende Versuche unternommen, das Problem des Wartens auf einen freien Platz zu lösen.

### 2.5.1 Erster Lösungsversuch

Hier ist der erste Versuch (Listing 2.23):

**Listing 2.23**

```
class ParkingGarage
{
    private int places; //Anzahl freier Plätze

    public ParkingGarage(int places)
    {
        if(places < 0)
        {
            throw new IllegalArgumentException("Parameter < 0");
        }
        this.places = places;
    }

    public synchronized void enter()
    {
```

```

        while(places == 0);
        places--;
    }

    public synchronized void leave()
    {
        places++;
    }
}

```

Falls Konstruktoren oder Methoden mit nicht passenden Parametern aufgerufen werden, wird in diesem Buch in der Regel die Ausnahme `IllegalArgumentException` geworfen. Diese Ausnahme ist eine sogenannte `Unchecked Exception` (abgeleitet aus `RuntimeException`), die in der Signatur eines Konstruktors bzw. einer Methode nicht mit `throws` deklariert werden muss. In Listing 2.23 wird diese Ausnahme dann ausgelöst, wenn man versucht, ein Parkhaus mit einer negativen Anzahl freier Plätze zu erzeugen.

Dieser Lösungsversuch hat zwei Probleme:

1. Wieder einmal wird *aktives Warten* benutzt, was sehr ineffizient ist.
2. Das noch größere Problem bei dieser Lösung ist, dass sie nicht korrekt arbeitet. Sobald nämlich ein Auto in ein volles Parkhaus durch Aufruf der Methode `enter` einfahren will, kann dieser Thread die Methode `enter` nie mehr verlassen und somit die Sperre des Parkhausobjekts nie mehr freigeben. Deshalb können auch keine Autos mehr aus dem Parkhaus herausfahren; das Parkhaus kann folglich nicht mehr benutzt werden.

## 2.5.2 Zweiter Lösungsversuch

Versuchen wir zuerst das größere Problem (Problem Nr. 2) zu lösen. Offenbar ist die Ursache des geschilderten Problems das Warten auf einen freien Platz innerhalb der `Synchronized`-Methode. Deshalb wird im Folgenden (Listing 2.24) versucht, das Warten auf einen freien Platz nicht innerhalb einer `Synchronized`-Methode durchzuführen.

**Listing 2.24**

```

class ParkingGarage
{
    private int places; //Anzahl freier Plätze

    public ParkingGarage(int places)
    {
        if(places < 0)
        {
            throw new IllegalArgumentException("Parameter < 0");
        }
        this.places = places;
    }

    private synchronized boolean isFull()
    {
        return (places == 0);
    }
}

```

```

private synchronized void decrementPlaces()
{
    places--;
}

public void enter()
{
    while(isFull());
    decrementPlaces();
}

public synchronized void leave()
{
    places++;
}
}

```

Im Vergleich zu der vorigen Lösung ist die Methode enter nun nicht mehr synchronized. Um auf den Zustand des Parkhausobjekts zuzugreifen, werden deshalb neue, private Synchronized-Methoden isFull und decrementPlaces bereitgestellt, um den Zustand des Parkhauses abzufragen bzw. die Anzahl der freien Plätze um eins zu reduzieren. Aber auch diese Lösung hat zwei große Nachteile:

1. Wie zuvor wird aktiv gewartet, was nicht verwunderlich ist, da wir dieses Problem aus dem ersten Lösungsversuch nicht angegangen sind.
2. Auch diese Lösung ist nicht korrekt. Das Herausziehen des Wartens aus synchronized bringt uns zurück zum Problem der verlorenen Buchung. Angenommen, es ist noch ein Platz im Parkhaus frei. Ein Auto, das einfahren will, ruft die Methode isFull auf und bekommt den Wert false zurückgeliefert. Deshalb wird die While-Schleife verlassen. Wir befinden uns jetzt in der Ausführung zwischen den beiden Anweisungen der Methode enter. Wenn jetzt auf einen anderen Thread umgeschaltet wird, der ebenfalls in das Parkhaus einfahren will, so kann er dies tun, da immer noch ein Platz frei ist. Wird danach wieder auf den ersten Thread geschaltet, so setzt dieser seine Ausführung an der Stelle, an der er vorher war, fort, indem die Methode decrementPlaces aufgerufen wird. Das Auto kann jetzt fälschlicherweise auch noch einfahren und die Anzahl der freien Plätze des Parkhauses ist nun -1.

### 2.5.3 Dritter Lösungsversuch

Die ersten beiden Lösungsversuche führen nun vielleicht zu der Vermutung, dass mit synchronized allein keine korrekte Lösung möglich ist. Dass diese Vermutung nicht stimmt, beweist Listing 2.25:

**Listing 2.25**

```

class ParkingGarage
{
    private int places; //Anzahl freier Plätze
}

```

```

public ParkingGarage(int places)
{
    if(places < 0)
    {
        throw new IllegalArgumentException("Parameter < 0");
    }
    this.places = places;
}

public void enter()
{
    while(!tryToEnter());
}

private synchronized boolean tryToEnter()
{
    boolean success = false;
    if(places > 0)
    {
        places--;
        success = true;
    }
    return success;
}

public synchronized void leave()
{
    places++;
}
}

```

In der privaten Methode `tryToEnter`, die `synchronized` ist, wird geprüft, ob eine Einfahrt möglich ist. Sollte dies der Fall sein, wird die Anzahl der freien Plätze um eins verringert und Erfolg gemeldet. Im anderen Fall wird durch den Rückgabewert angezeigt, dass die Einfahrt nicht geklappt hat. In der Methode `enter`, die nicht `synchronized` ist, wird `tryToEnter` solange aufgerufen, bis die Einfahrt möglich war.

Diese Lösung ist korrekt. Sie besitzt aber immer noch den Nachteil, dass auf eine Einfahrt aktiv gewartet wird und so unter Umständen viel Rechenzeit vergeudet wird. Man könnte nun auf die Idee kommen, die Aktivität beim Warten durch einen Aufruf von `sleep` zu verringern. Wenn man die Schlafenszeit sehr klein macht, ist der Spareffekt relativ gering. Ist die Schlafenszeit dagegen groß, so kann es unter Umständen lange dauern, bis ein wartendes Auto bemerkt, dass es einfahren kann. Alles in allem ist somit auch diese Lösung nicht das, was wir wollen.

## 2.5.4 Korrekte und effiziente Lösung mit wait und notify

Alle drei Lösungsversuche verwenden das unerwünschte aktive Warten. Außerdem ist das Warten auf einen freien Platz sowohl im gesperrten Zustand des Parkhausobjekts im ersten Lösungsversuch als auch im nicht gesperrten Zustand im zweiten Lösungsversuch nicht korrekt. Eine korrekte und zudem effiziente Lösung ohne aktives Warten ist aber möglich, und zwar unter Verwendung der Methoden `wait` und `notify` der Klasse `Object`:

```
public class Object
{
    ...
    public final void wait() throws InterruptedException {...}
    public final void notify() {...}
}
```

Beide Methoden müssen auf ein Objekt angewendet werden, das der aufrufende Thread zum Zeitpunkt des Aufrufs durch synchronized gesperrt hat (sonst wird eine *IllegalMonitorStateException* geworfen). Im nachfolgenden Beispiel (Listing 2.26) ist dies der Fall: Die fehlende Angabe des Objekts beim Aufruf bedeutet bekanntlich this.wait() und this.notify(). Die Aufrufe von wait und notify werden also auf das ParkingGarage-Objekt angewendet, das in diesem Augenblick in der Tat gesperrt ist, da sich die Aufrufe in den Synchronized-Methoden enter und leave befinden. Die Methode wait blockiert den aufrufenden Thread und fügt ihn in eine Warteschlange des Objekts ein, auf das die Wait-Methode aufgerufen wird (d.h. in diesem Fall also in eine Warteschlange des entsprechenden ParkingGarage-Objekts). Die Methode notify entfernt einen Thread aus der Warteschlange des Objekts, auf das diese Methode angewendet wird. Ist diese Warteschlange leer, so hat der Aufruf von notify keine Wirkung. Es gibt keine Garantie, dass der Thread, der sich am längsten in der Warteschlange befindet, geweckt wird, sondern es wird irgendein Thread geweckt.

Im Folgenden ist eine korrekte Lösung des Parkhausproblems mit wait und notify zu sehen. Der Vollständigkeit halber sind neben der Klasse ParkingGarage auch noch die Auto-Threads der Klasse Car zu sehen, die ein Parkhaus benutzen, sowie eine Main-Methode zum Ausprobieren der Klassen.

### **Listing 2.26**

```
class ParkingGarage
{
    private int places;

    public ParkingGarage(int places)
    {
        if(places < 0)
        {
            throw new IllegalArgumentException("Parameter < 0");
        }
        this.places = places;
    }

    public synchronized void enter()
    {
        while(places == 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        places--;
    }
}
```

```
}

public synchronized void leave()
{
    places++;
    notify();
}

class Car extends Thread
{
    private ParkingGarage garage;
    public Car(String name, ParkingGarage garage)
    {
        super(name);
        this.garage = garage;
        start();
    }

    public void run()
    {
        while(true)
        {
            try
            {
                sleep((int) (Math.random() * 10000));
            }
            catch(InterruptedException e)
            {
            }
            garage.enter();
            System.out.println(getName() + ": eingefahren");
            try
            {
                sleep((int) (Math.random() * 20000));
            }
            catch(InterruptedException e)
            {
            }
            garage.leave();
            System.out.println(getName() + ": ausgefahren");
        }
    }
}

public class Parking
{
    public static void main(String[] args)
    {
        ParkingGarage garage = new ParkingGarage(10);
        for(int i = 1; i <= 40; i++)
        {
            new Car("Auto " + i, garage);
        }
    }
}
```

Es ist sehr wichtig, dass beim Aufruf von `wait` gleichzeitig die Sperre für das Objekt, auf das `wait` angewendet wird, aufgehoben wird. Denn sonst hätte diese Lösung dasselbe Problem wie der erste vergebliche Lösungsversuch, dass nämlich das Parkhaus bei dem Versuch einer Einfahrt in das volle Parkhaus gesperrt bleibt, folglich kein Auto mehr das Parkhaus verlassen und damit auch kein Auto mehr einfahren könnte. Wenn ein Thread aus `wait` durch `notify` von einem anderen Thread geweckt wird, dann muss der geweckte Thread zuerst das Objekt erneut sperren, bevor er weiteren Code der Anwendung ausführen kann.

Um das Verständnis dieses Sachverhalts zu überprüfen, wollen wir jetzt zwei Änderungen des Programms durchführen und prüfen, ob das resultierende Programm immer noch korrekt ist:

- Angenommen, wir vertauschen die beiden Anweisungen in der Methode `leave` der Klasse `ParkingGarage`. Im ersten Augenblick könnte es so aussehen, dass damit das Programm nicht mehr korrekt arbeitet, denn es wird zuerst ein wartender Thread geweckt und dann erst der Zähler erhöht. Tatsächlich ist das Programm aber weiterhin korrekt, denn der geweckte Thread kann erst weiterlaufen, wenn er die Sperre für das Parkhaus setzen kann. Dies ist aber frühestens dann der Fall, wenn der weckende Thread aus der Methode `leave` zurückgekehrt ist und damit dann auch der Zähler erhöht wurde. Die ursprüngliche Schreibweise wird aber bevorzugt, weil sie naheliegender und daher besser verständlich ist.
- Eine andere mögliche Änderung des Parkhausprogramms besteht darin, das `while` in der `Enter`-Methode durch `if` zu ersetzen. Auf den ersten Blick scheint diese Änderung an der Korrektheit des Programms nichts zu verändern, denn wenn ein Thread nicht einfahren kann, wird er durch `wait` blockiert. Dieser Thread wird aber genau dann geweckt, wenn ein anderer Thread ausfährt und die Anzahl der freien Plätze um eins erhöht hat. Man könnte meinen, dass damit ein geweckter Thread in jedem Fall einen freien Platz vorfinden muss. Diese Überlegung ist aber nicht korrekt und zeigt die Schwierigkeit im Umgang mit parallelen Abläufen. Es ist nämlich nicht gesagt, dass der geweckte Thread auch tatsächlich als nächster läuft. Es könnte genauso gut sein, dass ein dritter Thread den Prozessor zugeteilt bekommt und in das Parkhaus einfahren will. Da soeben ein Auto das Parkhaus verlassen hat, ist ein Platz frei und die Einfahrt möglich. Wenn dann etwas später auf den geweckten Thread umgeschaltet wird, so sind bereits wieder alle Plätze belegt und er muss erneut warten. Die Ersetzung von `while` durch `if` wäre also nicht korrekt. Die beschriebene Situation entspricht einer Situation, wo ein Auto vor einem vollen Parkhaus wartet, bis ein anderes Auto ausfährt. Nachdem ein Auto ausgefahren ist, kommt von hinten ein Auto und fährt vor dem schon seit längerer Zeit wartenden Auto in das Parkhaus ein. Diese ärgerliche Situation ist in unserem Parkhausprogramm also durchaus möglich. Will man das verhindern, so muss dies explizit programmiert werden; im Abschnitt 2.6.3 wird eine faire Variante eines Parkhauses vorgestellt.

Die Auto-Threads schlafen eine zufällig gewählte Zeit zwischen 0 und 10 Sekunden, bevor sie in das Parkhaus einfahren. Dies soll das Umherfahren auf den Straßen simulieren. Im Parkhaus hält sich ein Auto eine zufällig gewählte Zeit zwischen 0 und 20 Sekunden auf. Außerdem ist die Anzahl der Autos (40) deutlich größer als die Anzahl der freien Plätze im Parkhaus (10). Diese Werte wurden deshalb so gewählt, damit beim Ablauf des Programms mit hoher Wahrscheinlichkeit das Parkhaus voll wird und damit überprüft werden kann, ob sich zu jedem Zeitpunkt höchstens 10 Autos im Parkhaus befinden.

Wenn wir zur Überprüfung dieses Sachverhalts das Parkhausprogramm übersetzen und ausführen, erhalten wir z. B. folgende Ausgabe, wobei am Ende jeder Zeile manuell als Java-Kommentar die Anzahl der Autos angegeben ist, die sich scheinbar nun im Parkhaus befinden:

```
Auto 14: eingefahren // 1
Auto 23: eingefahren // 2
Auto 5: eingefahren // 3
Auto 12: eingefahren // 4
Auto 20: eingefahren // 5
Auto 26: eingefahren // 6
Auto 33: eingefahren // 7
Auto 12: ausgefahren // 6
Auto 38: eingefahren // 7
Auto 2: eingefahren // 8
Auto 30: eingefahren // 9
Auto 27: eingefahren // 10
Auto 25: eingefahren // 11
Auto 20: ausgefahren // 10
...
...
```

Bei der Betrachtung der Ausgabe des Programms fällt auf, dass sich nach Einfahrt des Autos 25 scheinbar 11 Autos im Parkhaus befinden. Programmieranfänger werden vielleicht relativ lange den „Programmierfehler“ vergeblich suchen. Hier liegt aber ein relativ tückischer Fall vor, bei dem das Programm nur scheinbar fehlerhaft ist. In Wirklichkeit ist das Programm korrekt und lediglich dessen Ausgabe verwirrend. Das Problem der Ausgabe ist am Ende der While-Schleife in der Run-Methode der Klasse Car zu suchen. Zuerst verlässt ein Auto das Parkhaus und danach erfolgt die Ausgabe. Wenn also im obigen Beispiel das Auto 20 das Parkhaus verlassen hat, wird auf das wartende Auto 25 umgeschaltet. Dieses fährt ein und gibt seinen Erfolg aus (vorletzte Zeile in der obigen Ausgabe). Danach wird auf den schon zuvor ausgefahrenen Thread mit dem Namen „Auto 20“ zurückgeschaltet und dieser gibt erst jetzt bekannt, dass er aus dem Parkhaus herausgefahren ist. Manchmal macht man beim Programmieren Fehler und erkennt sie nicht. In diesem Fall ist es umgekehrt: Man sieht Fehler, die gar nicht da sind.

Obwohl wir nun also hoffentlich doch von der Korrektheit des Programms überzeugt sind, ist die vorliegende Situation dennoch unbefriedigend, denn man möchte ein Verhalten des Programms sehen, das auch den Tatsachen entspricht und weniger irreführend ist. Offenbar ist die Ursache des Problems, dass Ein- und Ausfahrvorgänge und deren Ausgabe nicht eng genug gekoppelt sind und dazwischen eine Thread-Umschaltung erfolgen kann. Folgende Lösungsversuche kann man unternehmen:

1. Wenn die Ausgaben von der Run-Methode in die Enter- und Leave-Methoden verlagert werden, so entspricht die Ausgabe wesentlich besser dem tatsächlichen Verlauf. Allerdings ist davon abzuraten, wenn die Klasse ParkingGarage anderen Anwendungen zur Verfügung gestellt wird, die diese Ausgaben nicht brauchen oder nicht wollen. Eine Anwendung könnte das Verhalten z. B. grafisch darstellen, so dass die Ausgabe unnötig ist. Eine andere Anwendung könnte mit Hilfe der ParkingGarage-Klasse eine Simulation über die benötigte Parkplatzkapazität einer Stadt durchführen, in der Tausende von Ein- und Ausfahrten simuliert werden, um statistische Auswertungen vornehmen zu können. In einem solchen Fall verlangsamt diese Ausgabe die Simulation erheblich. Im Allgemeinen sollten keine Ausgaben in derartige Methoden eingebaut werden.

2. Um die Ein- und Ausfahrt mit der jeweiligen Ausgabe zu koppeln, könnte man auf die Idee kommen, beide Vorgänge jeweils in einem Synchronized-Block zusammenzufassen. Und zwar so, dass jeweils vor einer Ein- bzw. Ausfahrt eine Sperre gesetzt und nach der Ausgabe diese Sperre wieder freigegeben wird. Dabei müssen alle Threads dasselbe Objekt sperren und entsperren. Eine mögliche Lösung ist, hierfür ein Klassenobjekt der Klasse Car zu verwenden. Dies ist im folgenden Programm realisiert (Änderungen fett gedruckt):

```
class Car extends Thread
{
    private ParkingGarage garage;
    private static Object lock = new Object();

    public Car(String name, ParkingGarage garage)
    {
        ... //wie zuvor
    }

    public void run()
    {
        while(true)
        {
            ... //sleep wie zuvor
            synchronized(lock)
            {
                garage.enter();
                System.out.println(getName() + ": eingefahren");
            }
            ... //sleep wie zuvor
            synchronized(lock)
            {
                garage.leave();
                System.out.println(getName() + ": ausgefahren");
            }
        }
    }
}
```

Jetzt kann es offenbar nicht mehr passieren, dass ein Auto ausfährt und – bevor die entsprechende Ausgabe auf dem Bildschirm zu sehen ist – ein anderes Auto in das Parkhaus einfährt. Wird das Programm übersetzt und ausgeführt, so erlebt man allerdings eine Enttäuschung:

```
Auto 34: eingefahren // 1
Auto 6: eingefahren // 2
Auto 38: eingefahren // 3
Auto 21: eingefahren // 4
Auto 12: eingefahren // 5
Auto 26: eingefahren // 6
Auto 16: eingefahren // 7
Auto 19: eingefahren // 8
Auto 12: ausgefahren // 7
Auto 37: eingefahren // 8
Auto 40: eingefahren // 9
Auto 12: eingefahren // 10
Auto 40: ausgefahren // 9
Auto 31: eingefahren // 10
```

Hier wurden keine Zeilen für die Wiedergabe in diesem Buch weggelassen; dies ist tatsächlich die gesamte Ausgabe (ohne dass das Programm beendet ist). Das Programm scheint zu stehen. Dies ist bei näherer Betrachtung aber auch verständlich. Ein Auto, das in ein volles Parkhaus einfahren will, wird in der Methode enter blockiert. Dabei wird die Wait-Methode aufgerufen und die Sperre für das ParkingGarage-Objekt freigegeben. Da aber die Methode enter innerhalb des Synchronized(lock)-Blocks aufgerufen wird, wird die Sperre des Objekts lock nicht freigegeben. Damit kann aber kein Auto mehr das Parkhaus verlassen, denn dazu müsste erst die Sperre des Objekts lock gesetzt werden, um dann die Methode leave aufzurufen zu können. Somit kann also kein weiteres Auto mehr ein- oder ausfahren. Man spricht in diesem Fall von einer *Verklemmung*. Verklemmungen werden in Kapitel 3 ausführlich besprochen. Dieser Lösungsversuch ist also gescheitert.

3. Für diesen speziellen Fall gibt es eine wesentlich einfachere Lösung. Man vertauscht in der Run-Methode einfach die beiden letzten Anweisungen des ursprünglichen Programms (Änderungen fett gedruckt):

```
class Car extends Thread
{
    private ParkingGarage garage;

    public Car(String name, ParkingGarage garage)
    {
        ... //wie zuvor
    }

    public void run()
    {
        while(true)
        {
            ... //sleep wie zuvor
            garage.enter();
            System.out.println(getName() + ": eingefahren");
            ... //sleep wie zuvor
System.out.println(getName() + ": wird ausfahren");
garage.leave();
        }
    }
}
```

Damit erhalten wir eine stimmige Ausgabe. Beachten Sie, dass eine Vertauschung der Ausgabe mit dem Aufruf der Enter-Methode bei der Einfahrt wieder zu einer irreführenden Ausgabe führen würde.

4. Alle bisherigen Lösungsversuche sind nicht besonders befriedigend, denn Lösungsversuch Nr. 1 ist unschön, Lösungsversuch Nr. 2 ist nicht korrekt, und Lösungsversuch Nr. 3 funktioniert zwar in diesem speziellen Fall, lässt sich aber nicht verallgemeinern. Wir betrachten nun als letzte Variante eine allgemeingültige Lösung des Problems, die dem 2. Lösungsversuch sehr ähnlich ist: Wieder wird der Methodenaufruf zum Ein- bzw. Ausfahren in einem Synchronized-Block mit der entsprechenden Ausgabe gekoppelt. Der wesentliche Unterschied ist nun aber, dass als Sperrobjekt dieses Mal das Parkhaus-Objekt selbst verwendet wird.

```

class Car extends Thread
{
    private ParkingGarage garage;

    public Car(String name, ParkingGarage garage)
    {
        ... //wie zuvor
    }

    public void run()
    {
        while(true)
        {
            ... //sleep wie zuvor
            synchronized(garage)
            {
                garage.enter();
                System.out.println(getName() + ": eingefahren");
            }
            ... //sleep wie zuvor
            synchronized(garage)
            {
                garage.leave();
                System.out.println(getName() + ": ausgefahren");
            }
        }
    }
}

```

Betrachtet man den zweiten Synchronized-Block, so wird deutlich, dass nach dem Herausfahren eines Autos kein anderer Thread eine Ausgabe machen kann, da das Parkhaus-Objekt noch so lange gesperrt bleibt, bis auch die dazugehörige Ausgabe erfolgt ist. Die zuvor geschilderte Problematik einer möglichen Verklemmung existiert allerdings nicht mehr: Wenn nämlich nun die Methode enter aufgerufen wird, dann ist das Parkhaus zwar schon gesperrt, aber von demselben Thread, der jetzt enter aufruft, so dass das Parkhaus-Objekt sozusagen nochmals gesperrt wird und die Methode enter betreten werden kann (zur Erinnerung: Synchronized-Sperren sind reentrant). Der entscheidende Unterschied zu Lösungsversuch Nr. 2 ist nun, dass beim Aufruf von wait in enter die Doppelsperre aufgehoben wird und damit ein anderer Thread den zweiten Synchronized-Block, in dem leave aufgerufen wird, betreten kann. Wenn ein in enter wartender Thread später dann weiterläuft, so wird das Parkhaus erneut doppelt gesperrt (d.h. es wird derselbe Zustand wieder hergestellt, der vor dem Aufruf von wait vorlag). Sollte der Thread dann in das Parkhaus einfahren können und die Methode enter verlassen, so wird der Sperrenzähler von 2 auf 1 erniedrigt; das Parkhaus bleibt aber gesperrt, da der Sperrenzähler immer noch größer als 0 ist. Erst wenn die Ausgabe über das Einfahren erfolgt ist, wird der Synchronized-Block verlassen, der Sperrenzähler geht auf 0 und die Sperre des Parkhauses wird freigegeben.

Zum Abschluss dieses Abschnitts sei noch darauf hingewiesen, dass die Methode *wait* wie *join* mehrfach überladen ist, um das Warten zeitlich zu begrenzen (wie üblich durch Angabe der Wartezeit in Millisekunden oder in Milli- und Nanosekunden):

```

public class Object
{
    ...
    public final void wait(long millis)
        throws InterruptedException {...}
    public final void wait(long millis, int nanos)
        throws InterruptedException {...}
}

```

Damit scheinen die beiden überladenen Varianten von `wait` den beiden `Sleep`-Methoden ähnlich zu sein. Es gibt allerdings signifikante Unterschiede zwischen `sleep` und `wait`, die in der folgenden **Tabelle 2.3** zusammenfassend aufgeführt sind:

**Tabelle 2.3** Gegenüberstellung von `sleep` und `wait`

sleep	wait
Methoden der Klasse Thread	Methoden der Klasse Object
Klassenmethoden (static)	Instanzenmethoden (nicht static)
Keine Bedingungen für Aufruf	Kann nur angewendet werden auf ein Objekt, das im Augenblick durch <code>synchronized</code> gesperrt ist.
Falls Objekte gesperrt sind, so bleiben sie gesperrt.	Die Sperre des Objekts, auf das der Aufruf angewendet wird, wird aufgehoben (andere eventuell gesetzte Sperren bleiben allerdings gesetzt).

## ■ 2.6 NotifyAll

Mit `synchronized`, `wait` und `notify` lässt sich eine große Zahl von Synchronisationsaufgaben lösen. Betrachten wir nun folgende Aufgabe: Ein Thread, im Folgenden *Erzeuger (Producer)* genannt, will Messwerte (z.B. Temperatur, Geschwindigkeit eines vorbeifahrenden Autos) einem anderen Thread, im Folgenden *Verbraucher (Consumer)* genannt, zur Weiterverarbeitung (Anzeige, Mittelwertberechnung, Speichern) übergeben. Man könnte dazu ein von beiden Threads genutztes Objekt verwenden, in das mit `put` ein Messwert abgelegt und mit `get` ein Messwert ausgelesen werden kann (beide Methoden müssen dann natürlich `synchronized` sein). Wir nehmen an, dass dieses von beiden Threads genutzte Objekt nur Platz zum Speichern eines einzigen Messwerts hat. Dabei soll es aber nicht vorkommen, dass ein Messwert verloren geht. Dies könnte passieren, wenn der Erzeuger einen Messwert in das gemeinsam benutzte Objekt ablegt, obwohl der vorige Wert vom Verbraucher noch nicht gelesen wurde. Der alte Wert würde dann überschrieben. Es soll außerdem nicht vorkommen, dass der Verbraucher einen einzigen Messwert mehrfach ausliest. Dies könnte dann passieren, wenn zwischen zwei Aufrufen der `Get`-Methode kein neuer Wert vom Erzeuger angeliefert wurde. Durch Vergleichen des neu gelesenen Werts mit dem alten kann man übrigens nicht erkennen, ob ein Wert mehrfach gelesen wurde, denn wenn zwei aufeinanderfolgende gelesene Werte gleich sind, kann dies auch vorkommen, wenn derselbe Wert tatsächlich zweimal hintereinander gemessen wurde.

Sowohl das Verlieren als auch das Mehrfachlesen von Messwerten soll durch Warten verhindert werden: Will der Erzeuger mit put einen Messwert ablegen, dann wird dies solange verzögert, bis ein zuvor abgelegter Wert vom Verbraucher gelesen wurde und der Puffer somit wieder frei ist. Will umgekehrt der Verbraucher mit get einen Messwert abholen, dann wird dies so lange verzögert, bis ein neuer Messwert im Puffer bereitsteht. Wie schon zu vermuten, soll das Warten nicht durch aktives Warten realisiert werden.

## 2.6.1 Erzeuger-Verbraucher-Problem mit wait und notify

Im Folgenden wird eine Lösung dieser Aufgabe mit synchronized, wait und notify angegeben. Die Klasse Buffer (siehe Listing 2.27) realisiert das Austauschen der Messwerte mit den Methoden put und get. Ein einziges Attribut zur Zwischenspeicherung eines Messwerts (es wird hier von ganzzahligen Messwerten ausgegangen) reicht nicht aus, sondern es muss auch erkennbar sein, ob der Puffer voll oder leer ist (d.h. ob der aktuell vorhandene Messwert schon ausgelesen wurde oder noch nicht). Dieser Sachverhalt kann dem Messwert selber nicht angesehen werden, wenn man davon ausgeht, dass alle Zahlen (also z.B. auch 0 und negative Werte) gültige Messwerte sein können. Aus diesem Grund besitzen alle Objekte der Klasse Buffer ein zweites Attribut des Typs boolean mit dem Namen available, das angibt, ob ein Messwert im Moment zur Verfügung steht (d.h. ob der Puffer voll ist) oder nicht (d.h. ob der Puffer leer ist). Entsprechend muss in der Put-Methode gewartet werden, so lange der Puffer voll ist (d.h. so lange available == true) und in der Get-Methode, so lange der Puffer leer ist (d.h. so lange available == false).

**Listing 2.27**

```
class Buffer
{
    private boolean available = false;
    private int data;

    public synchronized void put(int x)
    {
        while(available)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        data = x;
        available = true;
        notify();
    }

    public synchronized int get()
    {
        while(!available)
        {
```

```

        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    available = false;
    notify();
    return data;
}
}

```

Am Ende der While-Schleife in der Put-Methode gilt die Bedingung available == false, d.h. der Puffer ist leer. Er wird dann mit dem Wert des Arguments gefüllt. Ferner wird das Attribut available auf true gesetzt, um anzusehen, dass ein neuer Messwert verfügbar ist. Schließlich wird mit notify der eventuell wartende Verbraucher geweckt. Entsprechendes gilt für die Methode get. Der Notify-Aufruf kann hier nicht die letzte Anweisung sein, da die Methode get einen Int-Wert als Rückgabewert zurückliefern muss. Dies ist aber nicht kritisch, wie im letzten Abschnitt im Zusammenhang mit dem Parkhaus bereits erläutert wurde, denn erst wenn die Rückgabe erfolgt ist, wird die Sperre auf das Objekt aufgehoben, und erst dann kann ein anderer Thread mit dem Buffer-Objekt arbeiten.

Im Folgenden sind der Erzeuger- und Verbraucher-Thread dargestellt (Listing 2.28). Wie im Parkhausbeispiel und in diesem Buch allgemein üblich wird eine Referenz auf das gemeinsam benutzte Objekt der Klasse Buffer im Konstruktor des Threads übergeben und als Attribut gemerkt. In der Run-Methode der Threads kann dann auf dieses Objekt zugegriffen werden. Der Verbraucher verbraucht 100 Werte. Entsprechend erzeugt der Erzeuger auch 100 Werte. Der erste zu produzierende Wert wird im Konstruktor der Klasse Producer zusätzlich zu der Referenz auf das gemeinsam benutzte Objekt als Parameter übergeben. Die anderen erzeugten Werte sind dann jeweils um 1 größer.

### **Listing 2.28**

```

class Producer extends Thread
{
    private Buffer buffer;

    private int start;

    public Producer(Buffer b, int s)
    {
        buffer = b;
        start = s;
    }

    public void run()
    {
        for(int i = start; i < start + 100; i++)
        {
            buffer.put(i);
        }
    }
}

```

```

}

class Consumer extends Thread
{
    private Buffer buffer;

    public Consumer(Buffer b)
    {
        buffer = b;
    }

    public void run()
    {
        for(int i = 0; i < 100; i++)
        {
            int x = buffer.get();
            System.out.println("gelesen: " + x);
        }
    }
}

public class ProducerConsumer
{
    public static void main(String[] args)
    {
        Buffer b = new Buffer();
        Consumer c = new Consumer(b);
        Producer p = new Producer(b, 1);
        c.start();
        p.start();
    }
}

```

In der Klasse ProducerConsumer befindet sich die Main-Methode. In ihr werden ein Buffer-Objekt, ein Verbraucher- und Erzeuger-Thread erzeugt und beide Threads gestartet. Die Ausgabe ist wie erwartet:

```

gelesen: 1
gelesen: 2
...
gelesen: 100

```

Es ist interessant zu sehen, was passiert, wenn mehrere Erzeuger- und Verbraucher-Threads gestartet werden, die alle auf demselben Buffer-Objekt arbeiten. Dazu ist die Main-Methode entsprechend zu ändern. In Listing 2.29 ist das Ergebnis dieser Änderung gezeigt.

### **Listing 2.29**

```

public class ProducerConsumer
{
    public static void main(String[] args)
    {
        Buffer b = new Buffer();
        Consumer c1 = new Consumer(b);
        Consumer c2 = new Consumer(b);
        Consumer c3 = new Consumer(b);
    }
}

```

```
    Producer p1 = new Producer(b, 1);
    Producer p2 = new Producer(b, 101);
    Producer p3 = new Producer(b, 201);
    c1.start();
    c2.start();
    c3.start();
    p1.start();
    p2.start();
    p3.start();
}
}
```

Wenn Sie dieses Programm ausführen, kann es passieren (je nach Betriebssystem, Java-Version usw.), dass einige Zeilen an Ausgabe erzeugt werden und das Programm dann stehen bleibt (d.h. es ist keine neue Ausgabe zu sehen, obwohl das Programm noch nicht zu Ende gelaufen ist). Ein solcher pathologischer Fall kann eintreten, wenn durch notify der „falsche“ Thread geweckt wird. Dazu betrachten wir den folgenden möglichen Ablauf des Programms:

1. Die drei Verbraucher-Threads c1, c2 und c3 bekommen alle zuerst die Möglichkeit zu laufen. Da der Puffer am Anfang leer ist, werden alle drei Threads blockiert und in die Warteschlange des Buffer-Objekts aufgenommen.
2. Als nächster Thread läuft der Erzeuger-Thread p1. Er kann einen Wert im Puffer ablegen und weckt einen der Verbraucher-Threads, nämlich c1. In der Warteschlange befinden sich nun noch c2 und c3, ein Messwert ist verfügbar.
3. Der Thread p1 läuft nach dem Wecken von c1 weiter und will den nächsten Wert im Puffer ablegen. Dies ist aber nicht möglich, da der Puffer voll ist. Der Thread blockiert sich. Die Warteschlange beinhaltet nun p1, c2 und c3, ein Messwert ist weiterhin verfügbar.
4. Es wird nun auf den Erzeuger-Thread p2 geschaltet, der ebenfalls einen Wert im Puffer ablegen will. Wie p1 zuvor wird auch p2 blockiert. Dasselbe wiederholt sich für den Erzeuger-Thread p3. Die Warteschlange enthält nun p1, p2, p3, c2 und c3, ein Messwert ist immer noch verfügbar.
5. Der einzige Thread, auf den umgeschaltet werden kann, ist der zuvor geweckte Thread c1. Dieser kann nun den verfügbaren Messwert entnehmen. Ferner wird irgendeiner der in der Warteschlange wartenden Threads geweckt. Dies sei z.B. c2. Die Warteschlange enthält jetzt p1, p2, p3 und c3, ein Messwert steht nicht zur Verfügung.
6. Der Thread c1 arbeitet weiter und will den nächsten Wert aus dem Puffer entnehmen. Da dies nicht möglich ist, wird c1 blockiert und in die Warteschlange eingefügt. Damit enthält die Warteschlange nun p1, p2, p3, c1 und c3, ein neuer Messwert steht immer noch nicht zur Verfügung.
7. Der einzige lauffähige Thread ist c2. Dieser Verbraucher-Thread versucht ebenfalls einen Wert aus dem Puffer zu entnehmen und wird blockiert, da kein Messwert vorhanden ist. Jetzt befinden sich alle Threads p1, p2, p3, c1, c2 und c3 in der Warteschlange. Es tut sich nichts mehr.

Das entscheidende Ereignis passiert im fünften Schritt, in dem der Verbraucher-Thread c1 den „falschen“ Thread, nämlich einen anderen Verbraucher-Thread c2, weckt. Wie Sie sehen, ist ein solcher Ablauf aber durchaus möglich. Es gibt mehrere Möglichkeiten, diesen

Fehler zu beheben. Die einfachste Möglichkeit ist diejenige, jedes Mal alle wartenden Threads zu wecken statt nur irgendeinen. Damit ist man auf der sicheren Seite. Dass damit zu viele Threads geweckt werden, ist – unter Vernachlässigung von Effizienzgesichtspunkten – kein Problem, da der Aufruf von `wait` ohnehin in einer While-Schleife erfolgt und nach dem Erwachen die Wartebedingung erneut überprüft wird.

## 2.6.2 Erzeuger-Verbraucher-Problem mit `wait` und `notifyAll`

Mit der Methode `notifyAll` aus der Klasse `Object` können alle in der Warteschlange eines Objekts wartenden Threads geweckt werden. Es werden dabei aber nicht alle vorhandenen Threads des Prozesses oder gar des ganzen Systems geweckt, sondern eben nur alle an dem betreffenden Objekt Wartenden. Ist die Warteschlange leer, so ist `notifyAll` wirkungslos. Wie `wait` und `notify` muss `notifyAll` auf ein Objekt angewendet werden, das momentan mit `synchronized` gesperrt ist, ansonsten wird eine `IllegalMonitorStateException` ausgelöst.

```
public class Object
{
    ...
    public final void notifyAll() {...}
    ...
}
```

Wenn also ein Buffer-Objekt von mehreren Erzeuger- und Verbraucher-Threads benutzt wird, dann sollte eine korrekte Variante der Klasse Buffer aussehen wie in Listing 2.30 (Änderungen gegenüber der vorigen Version aus Listing 2.27 sind fett gedruckt):

**Listing 2.30**

```
class Buffer
{
    private boolean available = false;
    private int data;

    public synchronized void put(int x)
    {
        while(available)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        data = x;
        available = true;
    notifyAll();
    }

    public synchronized int get()
    {
```

```

        while(!available)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    available = false;
notifyAll();
    return data;
}
}

```

Wie schon erwähnt wurde, schadet es – von Effizienzgesichtspunkten abgesehen – nicht, wenn zu viele Threads geweckt werden, da alle Threads ihre Wartebedingungen nach dem Wecken nochmals überprüfen. Aus diesem Grund kann in einem Programm jeder Notify-Aufruf durch einen NotifyAll-Aufruf ersetzt werden. Das Umgekehrte gilt nicht. Dies wurde am Erzeuger-Verbraucher-Problem deutlich: Falls der Puffer von mehreren Erzeuger- und Verbraucher-Threads benutzt wird, muss notifyAll verwendet werden. Die ursprüngliche Lösung mit notify ist nicht korrekt.

Aus dem Gesagten könnte man folgern, dass man nur notifyAll verwenden sollte, um immer auf der sicheren Seite zu sein. Es ist aber weniger effizient, wenn immer alle an einem Objekt wartenden Threads geweckt werden, auch wenn nur einer weiterlaufen kann wie im Parkhausbeispiel. Also sollte man notifyAll tatsächlich nur dann verwenden, wenn man es wirklich braucht.

Es gilt folgende Regel:



Die Methode notifyAll muss statt notify benutzt werden, wenn mindestens einer der beiden folgenden Fälle gilt:

1. In der Warteschlange befinden sich Threads mit unterschiedlichen Wartebedingungen. In diesem Fall besteht bei Verwendung von notify die Gefahr, dass der „falsche“ Thread geweckt wird. Das Erzeuger-Verbraucher-Problem ist ein Beispiel dafür. Hier gibt es Threads, die darauf warten, dass der Puffer voll wird und andere Threads, die darauf warten, dass der Puffer leer wird.
2. Durch die Veränderung des Zustands eines Objekts können mehrere Threads ihre While-Wait-Schleife verlassen. Als Beispiel kann man sich eine Ampel vorstellen, an der mehrere Autos warten. Wird die Ampel auf grün geschaltet, können alle wartenden Autos weiterfahren.

Wir können nun zum Abschluss dieses Abschnitts das Erzeuger-Verbraucher-Programm übersetzen und ausführen. Es gibt drei Erzeuger, wobei der erste die Werte 1 bis 100, der zweite die Werte 101 bis 200 und der dritte die Werte 201 bis 300 erzeugt. Da zwischen den Threads umgeschaltet wird, müssen diese Werte nicht sortiert in der Reihenfolge 1 bis 300

in den Puffer abgelegt werden. Allerdings werden die Werte streng in aufsteigender Reihenfolge erzeugt, wenn man nur einen einzigen Erzeuger-Thread betrachtet. Das heißt also z.B., dass der Wert 97 vor dem Wert 98 erzeugt wird. Bei der Ausführung des Programms kann allerdings unter bestimmten Umständen folgende Ausgabe beobachtet werden:

```
...
gelesen: 98
gelesen: 97
...
```

Wie im Parkhausprogramm könnte man auch hier einen Fehler vermuten, der in Wirklichkeit aber nicht existiert. Das Programm ist korrekt. Die irreführende Ausgabe kommt nicht durch die Erzeuger zustande, sondern durch das Vorhandensein mehrerer Verbraucher. Es kann vorkommen, dass ein Verbraucher den Wert 97 aus dem Puffer entnimmt, allerdings ohne die dazugehörige Meldung auszugeben. Danach wird auf den Erzeuger-Thread, der die 97 produziert hat, umgeschaltet. Dieser Thread legt dann den Wert 98 in den Puffer ab. Jetzt wird wieder auf einen Verbraucher-Thread umgeschaltet, aber auf einen anderen als den, der zuvor die 97 aus dem Puffer gelesen hat. Dieser andere Verbraucher-Thread entnimmt dem Puffer nun den Wert 98 und gibt die entsprechende Meldung auf dem Bildschirm aus. Anschließend wird auf den Verbraucher-Thread geschaltet, der zuvor den Wert 97 gelesen hat. Dieser Thread gibt jetzt erst seinen zuletzt gelesenen Wert 97 aus. Somit wird 98 vor 97 ausgegeben, obwohl der Wert 97 vor dem Wert 98 aus dem Puffer gelesen wurde. Derartige Ausgaben können vermieden werden, indem dieselbe Strategie wie im Parkhausbeispiel (Lösungsversuch Nr. 4) angewendet wird: Der Aufruf von `buffer.get` und `System.out.println` kann in einem Synchronized-Block mit `buffer` als Sperrobject gekoppelt werden (`synchronized(buffer)`).

### 2.6.3 Faires Parkhaus mit wait und notifyAll

In Abschnitt 2.5 wurde ein Parkhaus vorgestellt, bei dem die Autos nicht in jedem Fall *fair* bedient werden. Das heißt, dass die Autos nicht unbedingt in der Reihenfolge, in der sie beim Parkhaus ankommen, einfahren dürfen. Es kann sogar sein, dass ein Auto, das neu in das Parkhaus einfahren will und bisher noch gar nicht gewartet hat, wartende Autos überholt. Im Folgenden geben wir eine mögliche Implementierung eines fairen Parkhauses an (s. Listing 2.31). Dabei wird die in manchen Supermärkten übliche Form der Bedienung an Wurst- und Käsetheken nachgeahmt. Jede Kundin, die bedient werden will, zieht sich zu Beginn eine Wartenummer. Hinter dem Bedienpersonal leuchtet die Nummer der Kundin auf, die gerade bedient wird. Man wird genau dann bedient, wenn die Nummer angezeigt wird, die man selbst gezogen hat.

#### **Listing 2.31**

```
public class ParkingGarageFair
{
    private int places;
    private int nextWaitingNumber;
    private int nextEnteringNumber;
```

```

public ParkingGarageFair(int places)
{
    if(places < 0)
    {
        throw new IllegalArgumentException("Parameter < 0");
    }
    this.places = places;
    this.nextWaitingNumber = 0;
    this.nextEnteringNumber = 0;
}

public synchronized void enter()
{
    int myNumber = nextWaitingNumber++;
    while(myNumber != nextEnteringNumber || places == 0)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    places--;
    nextEnteringNumber++;
    notifyAll(); // wichtig !!!!
}
public synchronized void leave()
{
    places++;
    notifyAll(); // wichtig !!!!
}
}

```

Wie zuvor wird mit dem Attribut places die Anzahl der freien Plätze im Parkhaus gezählt. Das Attribut nextWaitingNumber repräsentiert die Rolle, von der sich Kunden ihre Wartenummer abreißen. Schließlich steht das Attribut nextEnteringNumber für die Nummer, die gerade an der Reihe ist. Jeder Kunde speichert sich die von ihm gezogene Nummer in der lokalen Variable myNumber. Man kann in das Parkhaus einfahren, wenn es mindestens einen freien Platz gibt und man an der Reihe ist. Durch die Wartenummer wird die Wartebedingung parametrisiert. Deshalb muss beim Herausfahren notifyAll statt notify aufgerufen werden. Außerdem ist nun auch ein Aufruf von notifyAll nach erfolgreichem Einfahren notwendig, was auf den ersten Blick überraschend sein mag. Stellen Sie sich aber vor, dass ein Auto aus dem Parkhaus ausfährt und mit notifyAll alle wartenden Autos weckt. Anschließend fährt ein zweites Auto heraus, wobei der Aufruf von notifyAll dann keine Wirkung mehr hat, weil keine Autos mehr warten. Von den geweckten Autos kann zufällig zuerst das Auto laufen, das als übernächstes Auto an der Reihe ist. Folglich wartet dieses Auto erneut. Nachdem das Auto, das an der Reihe ist, in das Parkhaus eingefahren ist, muss es das nun nächste Auto, das inzwischen wieder blockiert ist, deblockieren, denn dieses darf jetzt auch einfahren.

## ■ 2.7 Prioritäten von Threads

Threads besitzen *Prioritäten*. Diese Prioritäten wirken sich auf die so genannte *Prozessorzuteilungsstrategie (Scheduling)* aus. Diese Strategie entscheidet darüber, wie lange ein Thread rechnend sein darf, bevor auf den nächsten Thread umgeschaltet wird, und welcher Thread als nächster an der Reihe ist. Die Prioritäten werden durch Zahlen repräsentiert, wobei bei den Java-Threads eine größere Zahl eine höhere Priorität bedeutet (in manchen Betriebssystemen ist dies umgekehrt). Der minimale und maximale Prioritätswert werden durch die Thread-Konstanten *MIN\_PRIORITY* bzw. *MAX\_PRIORITY* festgelegt. Der Wert von *MIN\_PRIORITY* ist auf 1 und der von *MAX\_PRIORITY* auf 10 gesetzt. Die Standard-Priorität, die alle Threads besitzen, falls man ihre Priorität nicht explizit ändert und die alle Threads in den bisher behandelten Beispielen hatten, wird durch die Konstante *NORM\_PRIORITY* der Klasse Thread angegeben. Ihr Wert ist auf 5 gesetzt.

Mit den Methoden *setPriority* und *getPriority* der Klasse Thread kann die Priorität eines Threads verändert bzw. gelesen werden:

```
public class Thread
{
    ...
    public final void setPriority(int newPriority) {...}
    public final int getPriority() {...}
    ...
}
```

Im Folgenden wird die Wirkung der Prioritäten anhand eines Beispiels beschrieben. Jeder Thread füllt sein eigenes Feld der Länge 1.000.000 mit dem jeweils aktuellen Zeitwert. Die aktuelle Zeit kann durch Methoden der Klasse System, die static sind, in Milli- oder Nanosekunden gelesen werden:

```
public class System
{
    ...
    public static long currentTimeMillis() {...}
    public static long nanoTime() {...}
    ...
}
```

Die Methode *currentTimeMillis* gibt die Anzahl der Millisekunden an, die seit dem 1. Januar 1970, 0 Uhr, vergangen sind. Die Methode *nanoTime* liefert einen Wert, dessen Bezug zur aktuellen Uhrzeit nicht spezifiziert ist. Beide Methoden können aber verwendet werden, um Zeiten durch mehrmaliges Aufrufen der jeweils selben Methode zu vergleichen (*currentTimeMillis* wurde in Listing 2.14 zum Messen der Ausführungszeit bereits benutzt). Im folgenden Beispiel verwenden wir die Methode *nanoTime* wegen der genaueren Auflösung. Nachdem alle Threads jeweils 1.000.000 Zeitwerte in ihr Feld geschrieben haben, kann anschließend festgestellt werden, welcher Thread wie viele Werte am Stück in sein Feld geschrieben hat und wann auf einen anderen Thread umgeschaltet wurde.

Betrachten wir zum besseren Verständnis ein einfaches Beispiel mit zwei Threads und einer Feldlänge von 10. Wir stellen uns vor, dass die gelesenen Zeitwerte kleine, fortlaufende

ganze Zahlen beginnend bei 1 sind (in Wirklichkeit handelt es sich natürlich um wesentlich größere Zahlen, wobei die Differenz zwischen zwei hintereinander gelesenen Werten in der Regel größer als eins ist). Angenommen, nach der Ausführung der beiden Threads würden die Felder der beiden Threads folgende Werte beinhalten:

```
Feld des Threads 0: {1, 2, 3, 4, 8, 9, 10, 15, 16, 17}
Feld des Threads 1: {5, 6, 7, 11, 12, 13, 14, 18, 19, 20}
```

Dann können wir daraus folgende Ausgabe erzeugen, die beschreibt, welcher Thread wie lange an der Reihe war:

```
Thread 0: 4
Thread 1: 3
Thread 0: 3
Thread 1: 4
Thread 0: 3
Thread 1: 3
```

Das bedeutet, dass Thread 0 zuerst 4 Mal an der Reihe war; dies entspricht den vier kleinsten Werten 1, 2, 3 und 4, die alle im Feld von Thread 0 zu finden sind. Danach wurde auf Thread 1 umgeschaltet. Thread 1 war 3 Mal an der Reihe, da die nächsten Werte 5, 6 und 7 alle aus dem Feld des Threads 1 stammen. Danach war Thread 0 wieder an der Reihe, und zwar ebenfalls 3 Mal; dies entspricht den Werten 8, 9 und 10 im Feld des Threads 0.

In Listing 2.32 ist der vollständige Programmtext zu finden. Mit Hilfe der Kommandozeilenargumente kann man angeben, wie viele Threads laufen und welche Priorität diese haben sollen. Gibt man als Argumente z.B. „2 5 3 5“ an, dann werden vier Threads gestartet, wobei der erste Thread die Priorität 2, der nächste Thread die Priorität 5, der dritte Thread die Priorität 3 und der vierte Thread die Priorität 5 hat.

### **Listing 2.32**

```
class TimestampThread extends Thread
{
    private long[] timestamps;

    public TimestampThread(int capacity)
    {
        timestamps = new long[capacity];
    }

    public void run()
    {
        for(int i = 0; i < timestamps.length; i++)
        {
            timestamps[i] = System.nanoTime();
        }
    }

    public long getTimestamps(int i)
    {
        if(i >= 0 && i < timestamps.length)
        {
            return timestamps[i];
        }
    }
}
```

```

        return Long.MAX_VALUE;
    }

}

public class SchedulingObserver
{
    private static final int NUMBER_OF_ITERATIONS = 1000000;

    public static void main(String[] args)
    {
        if(args.length == 0)
        {
            System.out.println("Argumente: Liste von Prioritäten");
            return;
        }

        Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
        TimestampThread[] t = new TimestampThread[args.length];
        for(int i = 0; i < t.length; i++)
        {
            t[i] = new TimestampThread(NUMBER_OF_ITERATIONS);

            int priority = Thread.NORM_PRIORITY;
            try
            {
                priority = Integer.parseInt(args[i]);
                if(priority < Thread.MIN_PRIORITY ||
                   priority > Thread.MAX_PRIORITY)
                {
                    throw new NumberFormatException();
                }
            }
            catch(NumberFormatException e)
            {
                System.out.println("ungültiger Prioritätswert: " +
                                   + args[i]);
                return;
            }
            t[i].setPriority(priority);
        }
        for(int i = 0; i < t.length; i++)
        {
            t[i].start();
        }
        for(int i = 0; i < t.length; i++)
        {
            try
            {
                t[i].join();
            }
            catch(InterruptedException e)
            {
            }
        }
    }

    for(int i = 0; i < t.length; i++)
    {
        System.out.println("Priorität von Thread " + i

```

```

        + " : " + t[i].getPriority());
    }
    System.out.println("=====");

    int currentThread = -1;
    int runs = 1;
    int[] currentIndices = new int[t.length];
    for(int i = 0; i < NUMBER_OF_ITERATIONS * t.length; i++)
    {
        int previousThread = currentThread;
        currentThread = -1;
        long minimum = Long.MAX_VALUE;
        for(int j = 0; j < t.length; j++)
        {
            if(t[j].getTimestamps(currentIndices[j]) <= minimum)
            {
                currentThread = j;
                minimum = t[j].getTimestamps(currentIndices[j]);
            }
        }
        currentIndices[currentThread]++;
        if(currentThread == previousThread)
        {
            runs++;
        }
        else if(previousThread != -1)
        {
            System.out.println("Thread " + previousThread
                               + " : " + runs);
            runs = 1;
        }
    }
    System.out.println("Thread " + currentThread
                       + " : " + runs);

    System.out.println("=====");
}
}

```

Die Thread-Klasse ist leicht verständlich; in der Run-Methode werden die aktuellen Zeitwerte in das Feld geschrieben, und mit der Methode getTimestamps können die Zeitstempel wieder gelesen werden. Eine Synchronisation ist nicht nötig, da das Lesen der Werte erst nach dem Ende der schreibenden Threads erfolgt.

In der Main-Methode werden die Threads erzeugt, die Kommandozeilenargumente ausgewertet, die Prioritäten gesetzt, die Threads gestartet, auf deren Ende gewartet, die Felder mit den Zeitstempeln analysiert und eine Ausgabe produziert, die anzeigt, wie viele Runden ein Thread jeweils an der Reihe war (wie oben im Beispiel angegeben). Die Details mögen sich die Leserinnen und Leser selber erschließen. Lediglich auf einen Aspekt soll noch hingewiesen werden, der in der folgenden Anweisung enthalten ist:

```
Thread.currentThread().setPriority(Thread.MAX_PRIORITY);
```

Die Methode *currentThread* der Klasse Thread ist static. Sie liefert den aktuell laufenden Thread (d.h. den Thread, der diese Methode momentan aufruft) zurück:

```
public class Thread
{
    ...
    public static Thread currentThread() {...}
    ...
}
```

In obigem Programm bekommen wir damit eine Referenz auf den Thread, der die Main-Methode ausführt. Seine Priorität wird auf die maximale Priorität gesetzt. Dies geschieht deshalb, damit beim Starten der Threads möglichst alle Threads relativ schnell hintereinander gestartet werden. Sonst könnte es sein, dass der erste Thread relativ lange rechnend ist, wenn er eine höhere Priorität als der Main-Thread besitzt. Ein weiterer Thread mit noch höherer Priorität würde dann erst einmal längere Zeit gar nicht gestartet. So könnte es zu irreführenden Ausgaben kommen.

Betrachten wir nun die Ausgaben einiger Probeläufe unseres Programms aus Listing 2.32, die alle von einem Rechner mit einem einzigen Prozessor und einem einzigen Kern stammen. Für dieses Programm ist nämlich echte Parallelität eher störend, da damit die Threads mit den höchsten Prioritäten echt gleichzeitig laufen können und damit die Ausgabe des Programms sehr lang und unübersichtlich wird (man erhält sehr viele Zeilen mit einer 1 am Ende). Wir beginnen mit drei Threads mit Normalpriorität (d.h. das Programm wird mit den Kommandozeilenargumenten „5 5 5“ gestartet). Das Ergebnis sieht auf einem älteren Notebook unter Windows so aus (... deutet wieder auf nicht abgedruckte Ausgabezeilen hin):

```
Priorität von Thread 0: 5
Priorität von Thread 1: 5
Priorität von Thread 2: 5
=====
Thread 0: 38390
Thread 1: 40329
Thread 2: 41333
Thread 0: 46236
Thread 1: 51596
Thread 2: 51459
Thread 0: 51274
Thread 1: 51581
Thread 2: 51598
Thread 0: 51273
Thread 1: 51566
Thread 2: 51587
...
Thread 0: 51303
Thread 1: 51473
Thread 2: 51582
Thread 0: 41315
Thread 1: 33203
Thread 2: 31512
=====
```

Man sieht, dass nach einer Anlaufphase die Threads relativ gleichmäßig ca. 50.000 Schleifendurchläufe machen, bevor auf den nächsten Thread umgeschaltet wird. Ein weiterer Lauf des Programms mit den Kommandozeilenargumenten „2 3“ ergibt folgende Ausgabe:

```
Priorität von Thread 0: 2
Priorität von Thread 1: 3
=====
Thread 1: 1000000
Thread 0: 1000000
=====
```

Nicht ganz überraschend sieht man bei dieser Ausgabe, dass der Thread mit der größeren Priorität 3 zuerst ganz zu Ende lief, bevor der andere Thread an die Reihe kam. Wenn man jetzt vermutet, dass dies bei unterschiedlichen Prioritäten immer so sein muss, dann hat man sich aber getäuscht, wie die Ausgabe eines Laufs unseres Programms mit den Kommandozeilenargumenten „1 2“ zeigt:

```
Priorität von Thread 0: 1
Priorität von Thread 1: 2
=====
Thread 0: 30323
Thread 1: 39100
Thread 0: 49767
Thread 1: 51690
Thread 0: 51430
Thread 1: 51704
Thread 0: 51657
Thread 1: 51096
Thread 0: 51607
Thread 1: 51626
...
Thread 0: 51400
Thread 1: 51693
Thread 0: 44269
Thread 1: 33103
=====
```

Diese Ausgabe sieht genau so aus, als ob die beiden Threads dieselbe Priorität gehabt hätten. In Windows war dies wohl auch so. Daran sieht man, dass Java-Threads mit unterschiedlicher Priorität auf Threads derselben Priorität im darunter liegenden Betriebssystem abgebildet werden können. Aber selbst, wenn die Prioritäten offensichtlich auch im darunter liegenden Betriebssystem unterschiedlich sind, dann heißt das nicht, dass Threads mit kleinerer Priorität nie zum Zuge kommen, so lange es lauffähige Threads mit höherer Priorität gibt, wie die Ausgabe des letzten Probelaufs unseres Programms mit den Kommandozeilenargumenten „2 4 6 2 4 6“ zeigt:

```
Priorität von Thread 0: 2
Priorität von Thread 1: 4
Priorität von Thread 2: 6
Priorität von Thread 3: 2
Priorität von Thread 4: 4
Priorität von Thread 5: 6
=====
Thread 2: 36864
Thread 5: 41019
Thread 2: 48391
Thread 5: 51824
Thread 2: 52008
```

```
Thread 5: 51248
Thread 2: 51987
...
Thread 2: 32511
Thread 5: 28202
Thread 1: 50518
Thread 4: 51317
Thread 1: 51399
Thread 4: 51585
Thread 1: 51621
Thread 4: 51535
...
Thread 1: 50811
Thread 4: 51498
Thread 1: 51603
Thread 3: 95055
Thread 4: 51343
Thread 1: 51703
Thread 4: 51461
...
Thread 1: 22561
Thread 4: 23768
Thread 0: 99689
Thread 3: 51217
Thread 0: 51695
Thread 3: 51342
...
Thread 0: 51588
Thread 3: 30722
Thread 0: 27068
=====
=====
```

Zunächst laufen die Threads 2 und 5 mit der höchsten Priorität 6 bis zu ihrem Ende. Dann laufen die Threads 1 und 4 mit der nächst höheren Priorität 4. Hier schiebt sich aber an einer Stelle einmal der Thread 3 mit der niedrigsten Priorität 2 dazwischen.

Aus diesen Beobachtungen kann gefolgert werden, dass die Auswirkung von Thread-Prioritäten auf das Umschaltverhalten nicht exakt definiert ist. Man kann nur generell davon ausgehen, dass ein Thread mit einer höheren Priorität gegenüber einem Thread mit einer niedrigeren Priorität eventuell bevorzugt wird. Wie sich diese Bevorzugung auswirkt und wie stark diese Bevorzugung ist, ist nicht genau definiert und hängt von sehr vielen Umständen ab, u.a. auch vom Betriebssystem, auf dem das Programm ausgeführt wird. In keinem Fall dürfen daher Prioritäten zur Synchronisation verwendet werden. Wenn z.B. ein Thread eine Aktion a1 ausführt und ein anderer Thread eine Aktion a2 erst dann beginnen soll, nachdem a1 abgeschlossen ist, dann genügt es nicht, dem ersten Thread eine höhere Priorität als dem zweiten Thread zuzuweisen. In diesem Fall ist z.B. die Nutzung eines Semaphors (s. Kapitel 3) eine richtige Maßnahme.

Man fragt sich an dieser Stelle vielleicht nun, wozu Thread-Prioritäten denn dann überhaupt nützlich sind. Eine Einsatzmöglichkeit könnte eine Situation sein, in der einige Threads einer Anwendung Berechnungen durchführen (z.B. Simulationen) und andere Threads auf Ereignisse „von außen“ reagieren (z.B. Eingaben einer Benutzerin oder Ankunft von Daten über eine Netzverbindung). Die Threads, die Berechnungen durchführen, sind selten blockiert und die meiste Zeit rechenbereit. Man nennt solche Threads auch *rechen-*

*intensive Threads*. Die Threads, die auf Ereignisse „von außen“ warten, reagieren auf solche Ereignisse, indem auf eine Netzanfrage eine Antwort geschickt wird bzw. auf eine Benutzeingabe eine entsprechende Feedback-Meldung erfolgt. Danach wird auf das Eintreffen des nächsten Ereignisses gewartet. Solche Threads, die die meiste Zeit wegen des Wartens auf Ereignisse blockiert sind, werden *EA-intensive Threads* (EA: Ein-/Ausgabe) genannt. Im Allgemeinen ist es günstig, rechenintensiven Threads niedrigere Prioritäten und EA-intensiven Threads höhere Prioritäten zuzuordnen, denn die Anwendung sollte schnell auf äußere Ereignisse reagieren.

Zusammenfassend können folgende Erkenntnisse als Regeln für die Benutzung von Thread-Prioritäten festgehalten werden:



1. Die Korrektheit eines Programms darf nicht von der Zuteilung von Prioritäten an Threads und einem entsprechenden Umschaltverhalten des Systems abhängen. Prioritäten dürfen also z. B. nicht für die Synchronisation eingesetzt werden.
2. Prioritäten können verwendet werden, um die Effizienz eines Systems oder das Verhalten eines Systems nach außen zu verbessern, d. h. also zum so genannten Tunen eines Systems.

## ■ 2.8 Thread-Gruppen

Threads sind in Gruppen zusammengefasst. Ein neu erzeugter Thread wird im Normalfall zu der Gruppe hinzugefügt, in der sich der erzeugende Thread befindet. Beim Erzeugen eines Threads kann aber auch eine andere *Thread-Gruppe* angegeben werden, in der der neue Thread Mitglied wird. Diese Thread-Gruppe wird als erstes Argument der Thread-Konstruktoren angegeben. Das heißt, zu fast allen bereits kennengelernten Thread-Konstruktoren gibt es überladene Varianten mit einer Thread-Gruppe als erstem Argument, zu der der neue Thread hinzugefügt wird. Im Folgenden sind alle Thread-Konstruktoren dargestellt:

```
public class Thread
{
    public Thread() {...}
    public Thread(Runnable r) {...}
    public Thread(String name) {...}
    public Thread(Runnable r, String name) {...}
    public Thread(ThreadGroup group, Runnable r) {...}
    public Thread(ThreadGroup group, String name) {...}
    public Thread(ThreadGroup group, Runnable r, String name)
                 long stackSize) {...}
    ...
}
```

Bei dem zuletzt angegebenen Konstruktor kann als letzter Parameter die Kellergröße des Threads angegeben werden, die ohne diese explizite Festlegung auf einen Standardwert gesetzt wird. Auf den Keller wird bei jedem Methodenaufruf ein Datensatz gelegt, der bei der Rückkehr wieder entfernt wird. Da die zuletzt aufgerufene Methode als erste wieder verlassen wird, liegt hier das Kellerprinzip (LIFO: Last In, First Out) vor. Der Datensatz, der pro Methodenaufruf erzeugt wird, enthält im Wesentlichen alle lokalen Variablen der aufgerufenen Methode sowie die Rücksprungadresse. Das ist die Adresse der nächsten Anweisung im Programmcode, die der Methodenaufrufanweisung folgt.

Doch nun kehren wir zurück zu der Betrachtung von Thread-Gruppen: Die Zugehörigkeit eines Threads zu einer Thread-Gruppe kann nach dem Erzeugen nicht mehr verändert werden. Das heißt, es gibt keine Methoden zum Hinzufügen, Verschieben oder Entfernen von Threads.

Thread-Gruppen befinden sich selber wieder in Thread-Gruppen. Damit bilden die Threads und die Thread-Gruppen eine Baumhierarchie, wobei die Wurzel-Thread-Gruppe den Namen „system“ trägt. Neue Thread-Gruppen können explizit im Programm erzeugt werden. Die Klasse *ThreadGroup*, die in den Konstruktoren der Klasse *Thread* schon vorkam, hat zwei Konstruktoren:

```
public class ThreadGroup
{
    public ThreadGroup(String name) {...}
    public ThreadGroup(ThreadGroup parent, String name) {...}

    ...
}
```

Der zweite Konstruktor gibt neben dem Namen der neu erzeugten Thread-Gruppe die Gruppe an, in der die neue Gruppe Mitglied wird. Beim ersten Konstruktor, wo nur der Name der neuen Gruppe als Argument angegeben wird, wird die Thread-Gruppe zu der Gruppe hinzugefügt, in der sich auch der erzeugende Thread befindet.

Die Bedeutung von Thread-Gruppen ist eher als gering einzustufen. So gibt es z.B. für die Klasse *ThreadGroup* die Methoden *stop*, *suspend* und *resume*, die auf alle Thread-Mitglieder dieser Gruppe angewendet werden. Da diese Methoden aber auf Threads nicht mehr angewendet werden sollen (sie sind als „deprecated“ gekennzeichnet, s. Abschnitt 2.10), gilt dasselbe für die gleichnamigen Methoden für Thread-Gruppen (auch sie sind „deprecated“). Wie für Threads gibt es auch für Thread-Gruppen die Methode *interrupt*, welche die Methode *interrupt* auf alle Mitglieder der Gruppe anwendet (d.h. auf alle enthaltenen Threads und rekursiv alle enthaltenen Thread-Gruppen). Diese Funktion könnte aber bei Bedarf auch leicht selbst geschrieben werden.

Eine andere Methode der Klasse *ThreadGroup* ist *setMaxPriority*. Damit kann die maximal mögliche Priorität für alle Thread-Mitglieder dieser Gruppe festgelegt werden. Wenn ein Thread-Mitglied aber bereits eine höhere Priorität besitzt, so wird diese Priorität nicht reduziert, sondern so belassen.

Für Thread-Gruppen gibt es den Status eines *Daemons*. Dies bedeutet nicht, dass alle Thread-Mitglieder Daemon Threads (also Hintergrund-Threads, s. Abschnitt 2.9) sind, sondern dass eine solche Thread-Gruppe automatisch gelöscht wird, wenn sie keine Mitglieder mehr besitzt.

Der gesamte Baum der Thread-Gruppen und Threads lässt sich durchlaufen und anzeigen. Dazu gibt es Methoden, mit denen man sich im Baum „nach oben“ und „nach unten“ bewegen kann. Die Klasse `ThreadGroup` hat nur eine Methode, um sich „nach oben“ zu bewegen. Diese Methode heißt `getParent` und liefert die Thread-Gruppe zurück, in der sie sich befindet:

```
public class ThreadGroup
{
    ...
    public final ThreadGroup getParent() {...}
    ...
}
```

Zum Bewegen „nach unten“ kann man alle Mitglieder einer Thread-Gruppe erfragen:

```
public class ThreadGroup
{
    ...
    public int enumerate(Thread[] list) {...}
    public int enumerate(Thread[] list, boolean recurse) {...}
    public int enumerate(ThreadGroup[] list) {...}
    public int enumerate(ThreadGroup[] list, boolean recurse) {...}
    ...
}
```

Zum Abfragen der Mitglieder gibt es vier überladene `enumerate`-Methoden. Man kann entweder die Threads dieser Gruppe oder die Thread-Gruppen dieser Gruppe erfragen. Für beide Methoden gibt es eine Variante mit einem Argument des Typs `boolean`. Falls dieses Argument beim Aufruf `true` ist, so werden die entsprechenden Threads oder Thread-Gruppen rekursiv im Baum von diesem Knoten an abwärts ermittelt; ist das Argument `false`, werden nur die Mitglieder dieser Gruppe zurückgeliefert. Keine Angabe des zweiten Parameters bedeutet `true` (also rekursives Auflisten). Beim Auflisten der Threads werden nur solche Threads zurückgegeben, die gestartet wurden, aber noch nicht zu Ende gelaufen sind. In allen Fällen muss der Aufrufer ein Feld passender Länge zur Verfügung stellen. Ist das Feld zu klein, so werden nur so viele Mitglieder zurückgegeben, wie in das Feld passen. Ist das Feld zu groß, so werden nicht alle Feldelemente gefüllt. Der Rückgabewert aller `enumerate`-Methoden gibt die Anzahl der zurückgelieferten Mitglieder an. Zum Abfragen, wie viele Threads bzw. Thread-Gruppen eine Thread-Gruppe enthält, dienen die Methoden `activeCount` und `activeGroupCount`:

```
public class ThreadGroup
{
    ...
    public int activeCount() {...}
    public int activeGroupCount() {...}
    ...
}
```

Im folgenden Beispiel (Listing 2.33) werden alle Threads und alle Thread-Gruppen ausgegeben. Um die Wurzel aller Thread-Gruppen zu finden, wird diese mit Hilfe der Methode `getRoot` ermittelt, indem man im Baum so lange „nach oben“ geht, bis es nicht mehr weiter geht (die Methode `getParent` der Klasse `ThreadGroup` liefert null zurück). Als Startpunkt

der Suche wird dabei vom gerade laufenden Thread ausgegangen. Dieser wird mit der Static-Methode `currentThread` der Klasse `Thread` ermittelt (s. Abschnitt 2.7). Zu jedem Thread kann man mit der Methode `getThreadGroup` der Klasse `Thread` die Thread-Gruppe erfragen, in der dieser Thread Mitglied ist:

```
public class Thread
{
    public final ThreadGroup getThreadGroup() {...}
    ...
}
```

Im Beispiel werden zu Demonstrationszwecken einige Thread-Gruppen und Threads erzeugt. Da die Threads gestartet sein müssen, aber noch nicht zu Ende sein dürfen, damit sie aufgelistet werden, wird die `run`-Methode der Threads in diesem Beispiel so programmiert, dass sie nie zu Ende läuft. Dies wird realisiert, indem jeder Thread mit `join` auf sein eigenes Ende wartet. Nachdem die Threads erzeugt und gestartet sind, werden alle Thread-Gruppen und Threads ausgegeben.

### **Listing 2.33**

```
class NeverEndingThread extends Thread
{
    public NeverEndingThread(ThreadGroup group, String name)
    {
        super(group, name);
        start();
    }

    public void run()
    {
        try
        {
            this.join();
        }
        catch(InterruptedException e)
        {
        }
    }
}

public class GroupTree
{
    private static ThreadGroup getRoot()
    {
        ThreadGroup result =
            Thread.currentThread().getThreadGroup();
        while(result.getParent() != null)
        {
            result = result.getParent();
        }
        return result;
    }

    private static void dump(ThreadGroup group, int blanks)
    {
        for(int i = 0; i < blanks; i++)
    }
```

```

{
    System.out.print(" ");
}
System.out.println(group);

int numberOfThreads = group.activeCount();
Thread[] threadList = new Thread[numberOfThreads];
int threadNumber = group.enumerate(threadList, false);
for(int i = 0; i < threadNumber; i++)
{
    for(int j = 0; j < blanks + 3; j++)
    {
        System.out.print(" ");
    }
    System.out.println(threadList[i]);
}

int numberOfGroups = group.activeGroupCount();
ThreadGroup[] threadgroupList =
    new ThreadGroup[numberOfGroups];
int threadgroupNumber = group.enumerate(threadgroupList,
    false);
for(int i = 0; i < threadgroupNumber; i++)
{
    dump(threadgroupList[i], blanks + 3);
}
}

public static void dumpAll()
{
    dump(getRoot(), 0);
}

public static void main(String[] args)
{
    ThreadGroup group1 = new ThreadGroup("eigene Obergruppe");
    ThreadGroup group2 = new ThreadGroup(group1,
        "eigene Untergruppe");
    new NeverEndingThread(group1, "erster Thread");
    new NeverEndingThread(group2, "zweiter Thread");
    new NeverEndingThread(group2, "dritter Thread");

    dumpAll();
    System.exit(0);
}
}

```

Dieses Programm erzeugt folgende Ausgabe:

```

java.lang.ThreadGroup[name=system,maxpri=10]
    Thread[Reference Handler,10,system]
    Thread[Finalizer,8,system]
    Thread[Signal Dispatcher,10,system]
    Thread[Attach Listener,5,system]
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
java.lang.ThreadGroup[name=eigene Obergruppe,maxpri=10]
    Thread[erster Thread,5,eigene Obergruppe]

```

```
java.lang.ThreadGroup[name=eigene Untergruppe,maxpri=10]
    Thread[zweiter Thread,5,eigene Untergruppe]
        Thread[dritter Thread,5,eigene Untergruppe]
```

Die Ausgabe basiert auf überschriebenen `toString`-Methoden der Klassen `ThreadGroup` und `Thread`. Bei einer `ThreadGroup` wird offensichtlich der volle Klassename und in eckigen Klammern der Name der Gruppe sowie die maximale Priorität dieser Gruppe ausgegeben. Für einen `Thread` wird der Klassename und in eckigen Klammern der Name des `Threads`, dessen Priorität sowie der Name der `Thread-Gruppe`, in der sich dieser `Thread` befindet, ausgegeben. Neben dem `Main-Thread` und den durch das Programm erzeugten `Threads` und `Thread-Gruppen` sind u. a. auch die `Threads` mit den Namen „`Reference Handler`“ und „`Finalizer`“ zu sehen. Diese sind für die „`Abfallbehandlung`“ (`Garbage Collection`) von Java zuständig: der erste `Thread` sucht nach nicht mehr referenzierten Objekten, während der zweite `Thread` die Methode `finalize` auf jedes Objekt, das nicht mehr referenziert wird, vor dem Löschen anwendet.

## ■ 2.9 Vordergrund- und Hintergrund-Threads

In Abschnitt 2.4 wurde gesagt, dass ein Java-Programm – oder genauer, ein Prozess, der ein Java-Programm ausführt – zu Ende ist, wenn alle `Threads` zu Ende sind. Dies passt aber bei näherem Hinsehen nicht damit zusammen, dass es u. a. einen `Thread` gibt, der nach nicht mehr referenzierten Objekten sucht und den belegten Speicherplatz solcher Objekte wieder dem Freispeicher hinzufügt („`Reference Handler`“ in Abschnitt 2.8), denn ein solcher `Thread` läuft in einer Endlosschleife und endet somit nicht von selber. Die Lösung dieses Widerspruchs ist, dass es zwei Arten von `Threads` gibt, so genannte *User Threads* und so genannte *Daemon Threads*, die wir als *Vordergrund- und Hintergrund-Threads* bezeichnen wollen. Die obige Aussage lautet präziser, dass ein Java-Programm genau dann zu Ende ist, wenn alle Vordergrund-`Threads` zu Ende sind. Hintergrund-`Threads` werden also nicht beachtet, wenn es darum geht, festzustellen, ob ein Java-Programm zu Ende ist oder nicht.

`Threads` wie der `Reference Handler` sind Hintergrund-`Threads`, während der `Thread`, der die `Main-Methode` ausführt, und alle explizit vom Anwendungsprogramm heraus gestarteten `Threads` im Normalfall Vordergrund-`Threads` sind. Der Programmierer bzw. die Programmiererin kann aber einen selbst erzeugten `Thread` zu einem Hintergrund-`Thread` machen. Dies ist z. B. dann sinnvoll, wenn dieser `Thread` in einer Endlos-Schleife läuft und wenn das Ende des Java-Programms nicht von diesem `Thread` abhängen soll.

In der Klasse `Thread` gibt es in diesem Zusammenhang zwei Methoden, mit denen der Status bzgl. Vordergrund- oder Hintergrund-`Thread` gesetzt und gelesen werden kann. Diese Methoden heißen `setDaemon` und `isDaemon`:

```
public class Thread
{
    ...
    public final void setDaemon(boolean on) {...}
    public final boolean isDaemon() {...}
    ...
}
```

Wendet man auf einen Thread die Methode `setDaemon` mit dem Argument `true` an, so wird dieser Thread ein Hintergrund-Thread. Mit dem Argument `false` legt man fest, dass der Thread ein Vordergrund-Thread sein soll. Ohne explizite Festlegung hat ein neuer Thread immer denselben Status wie der erzeugende Thread.

Zwei Punkte verdienen besondere Beachtung:

1. Der Status eines Threads (Vordergrund- oder Hintergrund-Thread) hat nichts damit zu tun, wie häufig ein Thread läuft. Im Abschnitt 2.7 wurden Prioritäten von Threads behandelt. Nur diese Prioritäten spielen für die Prozessorzuteilungsstrategie eine Rolle. Der Status eines Threads wirkt sich nur auf das Ende des Java-Programms aus und hat mit Prioritäten nichts zu tun. Es ist also durchaus möglich, einem Hintergrund-Thread eine sehr hohe Priorität zuzuweisen. Dies trifft beispielsweise für den Reference Handler zu, der als Hintergrund-Thread die höchste Priorität 10 hat (s. Abschnitt 2.8).
2. Der Status eines Threads kann verändert werden, solange der Thread noch nicht gestartet wurde. Der Thread, der die `Main`-Methode der im Java-Kommando angegebenen Klasse ausführt, der so genannte Main-Thread, ist ein Vordergrund-Thread. Sein Status kann nicht mehr verändert werden, da er ja schon läuft. Von diesem Main-Thread erzeugte Threads können vor dem Starten allerdings zu Hintergrund-Threads gemacht werden.

## ■ 2.10 Weitere „gute“ und „schlechte“ Thread-Methoden

Damit haben wir die meisten Methoden (nach Ansicht des Autors alle wichtigen Methoden) der Klasse `Thread` behandelt. Weitere Methoden sind z. B.

- `dumpStack`, um einen „Stack Trace“ des gerade laufenden Threads auszugeben (was jeder Java-Programmierer und jede Java-Programmiererin beim Auftreten einer nicht abgefangenen Ausnahme schon einmal gesehen haben dürfte),
- `holdsLock`, um abzufragen, ob der gerade laufende Thread das als Parameter angegebene Objekt momentan mit `synchronized` gesperrt hat oder nicht (Rückgabe `boolean`),
- `setUncaughtExceptionHandler` bzw. `setDefaultUncaughtExceptionHandler`, um ein Objekt anzumelden, dessen Methode `uncaughtException` vor dem Abbruch eines Threads aufgerufen wird, falls in dem Thread eine nicht behandelte Ausnahme ausgelöst wird,
- `getState`, um den Status eines Threads (erzeugt, laufend, blockiert, beendet usw.) abzufragen,
- und `yield`, um die Wahrscheinlichkeit, dass auf einen anderen Thread umgeschaltet wird, deutlich zu erhöhen.

Die Methode `yield` sollte für „produktiven Code“ nicht benutzt werden, da sie kein Umschalten garantiert und schon gar nicht, auf welchen Thread umgeschaltet wird. Sie kann aber zum Demonstrieren von Situationen, wie sie in Abschnitt 2.2 beschrieben wurden, gewinnbringend eingesetzt werden. Es handelt sich dabei um Situationen, die man umgangssprachlich mit „wenn jetzt an dieser Stelle im Programm umgeschaltet werden würde“

beschreibt. Man kann solche Situationen wie z.B. die verloren gegangene Buchung aus Abschnitt 2.2 durch yield mit größerer Wahrscheinlichkeit gezielt herbeiführen.

Im Abschnitt 2.4.2 wurde bereits erwähnt, dass die Thread-Methode *stop*, mit der ein Thread zwangsweise beendet werden kann, „schlecht“ ist und deshalb von deren Benutzung abgeraten wird. Wenn nämlich ein Thread gerade eine Methode aufruft, so kann es sein, dass diese Methode beim Abbruch des Threads nur zum Teil ausgeführt wurde und sich das Objekt danach in einem inkonsistenten Zustand befindet. Dies kann im Folgenden beim Zugriff auf das Objekt durch andere Threads zu Problemen führen.

Neben der Methode *stop* wird auch vor der Verwendung der Thread-Methoden *suspend* und *resume* gewarnt; auch sie sind wie *stop* „deprecated“. Mit der Methode *suspend* kann die Ausführung eines Threads angehalten werden, mit *resume* wird ein angehaltener Thread fortgesetzt. Warum die auf den ersten Blick nützlichen Methoden *suspend* und *resume* als „schlecht“ gelten, sei anhand von zwei Beispielen erläutert:

Zum einen kann die Benutzung von *suspend* und *resume* die Verklemmungsgefahr erhöhen. So kann z.B. ein Thread durch *suspend* angehalten werden, der gerade eine *Synchronized*-Methode ausführt. Dadurch bleibt aber das betreffende Objekt längere Zeit gesperrt, denn die *Synchronized*-Methode läuft so erst einmal nicht zu Ende und im Gegensatz zu *wait* wird bei *suspend* keine Sperre freigegeben. Dies erhöht die Gefahr von Verklemmungen.

Zum anderen sind die Methoden *suspend* und *resume* nicht geeignet für die Synchronisation von Threads. Nehmen wir an, dass eine Aktion *a1* in einem Thread ausgeführt werden soll und erst dann, wenn diese Aktion beendet wurde, eine Aktion *a2* in einem anderen Thread. Man könnte meinen, dass dieses Problem einfach mit *suspend* und *resume* gelöst werden könnte. Im folgenden Beispiel sei *t2* eine Referenz auf den Thread, der die zweite *Run*-Methode ausführt.

```
/*
 * Diese Run-Methode wird vom ersten Thread ausgeführt;
 * t2 sei eine Referenz auf den Thread,
 * der die zweite Run-Methode ausführt.
 */
public void run()
{
    a1();
    t2.resume(); // den zweiten Thread weiterlaufen lassen
}

//-----

/*
 * Diese Run-Methode wird vom zweiten Thread ausgeführt;
 * die dazugehörige Klasse sei von Thread abgeleitet. */
public void run()
{
    this.suspend(); //sich selber anhalten
    a2();
}
```

Diese Synchronisation funktioniert zwar dann richtig, falls zuerst *suspend* im zweiten Thread ausgeführt wird und danach *resume* im ersten Thread. Falls aber zuerst *resume* ausgeführt wird, so hat der Aufruf dieser Methode keine Wirkung. Wenn dann anschließend *suspend* aufgerufen wird, dann hält sich dieser Thread für immer an. Wie dieses Problem korrekt mit Semaphoren gelöst werden kann, erfahren Sie im 3. Kapitel.

## ■ 2.11 Thread-lokale Daten

Die Klasse ThreadLocal besitzt die Methoden set und get, mit denen ein Objekt abgespeichert und wieder ausgelesen werden kann. Das Besondere dabei ist, dass unterschieden wird, welcher Thread welches Objekt mit set abgelegt hat. Entsprechend bekommt ein Thread mit get genau das Objekt zurück, das er mit set gespeichert hat (bzw. null, falls der aufrufende Thread noch nichts in das ThreadLocal-Objekt abgelegt hat).

In Listing 2.34 ist die Klasse CommonAndThreadLocalData zu sehen, die ein Attribut des Typs int und ein Attribut des Typs ThreadLocal, das ein Integer-Objekt beinhalten kann, besitzt. Die Synchronized-Methode next erhöht sowohl den Wert des Attributs des Typs int als auch den Wert des Integer-Objekts, welches in dem ThreadLocal-Objekt gespeichert wird. Sollte das ThreadLocal-Objekt noch nichts enthalten, so wird es mit einem Integer-Objekt, das den Wert 0 hat, initialisiert. Anschließend werden beide erhöhten Werte ausgegeben.

Es werden drei Threads erzeugt, die die Methode next auf ein gemeinsam benutztes Objekt der Klasse CommonAndThreadLocalData jeweils drei Mal anwenden.

**Listing 2.34**

```
class CommonAndThreadLocalData
{
    private int common;
    private ThreadLocal<Integer> local;

    public CommonAndThreadLocalData()
    {
        common = 0;
        local = new ThreadLocal<Integer>();
    }

    public synchronized void next()
    {
        common++;
        Integer integer = local.get();
        if(integer == null)
        {
            integer = new Integer(0);
        }
        int localData = integer.intValue() + 1;
        local.set(new Integer(localData));
        System.out.println(Thread.currentThread().getName()
                           + ": common = " + common
                           + ", local = " + localData);
    }
}

class CommonAndThreadLocalThread extends Thread
{
    private CommonAndThreadLocalData data;

    public CommonAndThreadLocalThread(CommonAndThreadLocalData data,
                                      String name)
```

```

    {
        super(name);
        this.data = data;
    }

    public void run()
    {
        for(int i = 1; i <= 3; i++)
        {
            data.next();
            yield();
        }
    }
}

public class ThreadLocalDemo
{
    public static void main(String[] args)
    {
        CommonAndThreadLocalData data =
            new CommonAndThreadLocalData();
        for(int i = 1; i <= 3; i++)
        {
            CommonAndThreadLocalThread t =
                new CommonAndThreadLocalThread(data,
                                               "T" + i);
            t.start();
        }
    }
}

```

Wie zu sehen ist, wird in der Methode run die statische Thread-Methode yield aufgerufen, um ein Umschalten auf einen anderen Thread nahezulegen und so eine möglichst große Durchmischung der Ausgaben zu erhalten. Betrachtet man die Ausgabe, so sieht man, dass jeder Thread seinen eigenen Zähler durch das ThreadLocal-Objekt besitzt, während das Attribut common von allen Threads gemeinsam benutzt wird (daher auch die Wahl der Bezeichner common und local sowie der Name der Klasse):

```

T1: common = 1, local = 1
T3: common = 2, local = 1
T2: common = 3, local = 1
T1: common = 4, local = 2
T3: common = 5, local = 2
T1: common = 6, local = 3
T3: common = 7, local = 3
T2: common = 8, local = 2
T2: common = 9, local = 3

```

Die Implementierung der Klasse ThreadLocal ist relativ einfach, denn man kann eine Hash-Map verwenden, in der als Schlüssel das Thread-Objekt des jeweils aufrufenden Threads verwendet wird. Listing 2.35 zeigt eine eigene Implementierung dieser Klasse.

**Listing 2.35**

```
class MyThreadLocal<T>
{
    private HashMap<Thread, T> map;

    public MyThreadLocal()
    {
        map = new HashMap<Thread, T>();
    }

    public T get()
    {
        return map.get(Thread.currentThread());
    }

    public void set(T value)
    {
        map.put(Thread.currentThread(), value);
    }

    public void remove()
    {
        map.remove(Thread.currentThread());
    }
}
```

Wird diese Klasse statt ThreadLocal in der obigen Anwendung in Listing 2.34 verwendet, erhält man dieselbe Ausgabe wie zuvor.

## ■ 2.12 Zusammenfassung

Mit synchronized, wait, notify und notifyAll lassen sich alle gängigen Synchronisationsaufgaben in Java lösen. Zum Abschluss werden die wichtigsten Konzepte noch einmal wiederholt.

Ein zentrales Konzept im Zusammenhang mit Threads in Java ist synchronized. Dieses Konzept ist in die Sprache Java „eingebaut“. Methoden einer Klasse (static und nicht static) können mit diesem Schlüsselwort versehen werden.

```
class C
{
    public synchronized void sm(...) {...}
    public static synchronized void ssm(...) {...}
}
```

Außerdem können Anweisungsblöcke mit synchronized geklammert werden, wobei hierzu die Angabe einer Referenz auf ein Objekt nötig ist:

```
class C
{
    public void sm(...)
    {
        ...
        synchronized(objReference)
        {
            ...
        }
        ...
    }
}
```

Ferner sind die Schnittstelle Runnable und die Klassen Thread, ThreadGroup und Object aus dem Package java.lang mit den im Folgenden aufgeführten Methoden in diesem Kapitel eingeführt worden:

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

public class Thread
{
    public Thread() {...}
    public Thread(Runnable r) {...}
    public Thread(String name) {...}
    public Thread(Runnable r, String name) {...}
    public Thread(ThreadGroup group, Runnable r) {...}
    public Thread(ThreadGroup group, String name) {...}
    public Thread(ThreadGroup group, Runnable r, String name) {...}
    public Thread(ThreadGroup group, Runnable r, String name,
                  long stackSize) {...}

    public static Thread currentThread() {...}
    public static void dumpStack() {...}
    public final String getName() {...}
    public final int getPriority() {...}
    public Thread.State getState() {...}
    public final ThreadGroup getThreadGroup() {...}
    public static boolean holdsLock(Object obj) {...}
    public void interrupt() {...}
    public static boolean interrupted() {...}
    public final boolean isAlive() {...}
    public final boolean isDaemon() {...}
    public boolean isInterrupted() {...}
    public final void join()
        throws InterruptedException {...}
    public final void join(long millis)
        throws InterruptedException {...}
    public final void join(long millis, int nanos)
        throws InterruptedException {...}
    public void run() {}
    public final void setDaemon(boolean on) {...}
    public static void setDefaultUncaughtExceptionHandler(
        Thread.UncaughtExceptionHandler eh)
{...}
```

```

public final void setName(String name) {...}
public final void setPriority(int newPriority) {...}
public void setUncaughtExceptionHandler(
    Thread.UncaughtExceptionHandler eh)
{
    ...
}

public static void sleep(long millis)
    throws InterruptedException {...}
public static void sleep(long millis, int nanos)
    throws InterruptedException {...}
public void start() {...}
public String toString() {...}
public static void yield() {...}
...
}

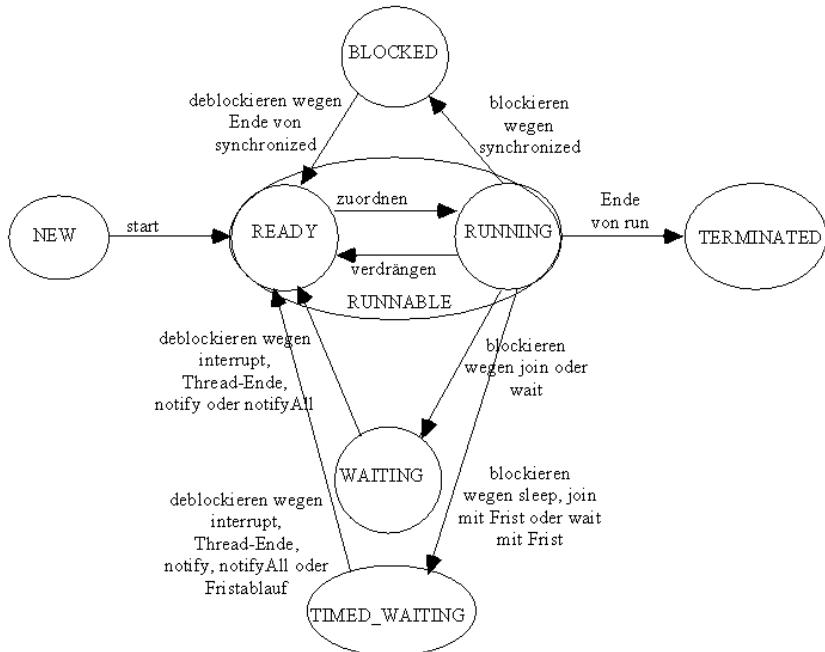
public class ThreadGroup
{
    public ThreadGroup(String name) {...}
    public ThreadGroup(ThreadGroup parent, String name) {...}

    public int activeCount() {...}
    public int activeGroupCount() {...}
    public int enumerate(Thread[] list) {...}
    public int enumerate(Thread[] list, boolean recurse) {...}
    public int enumerate(ThreadGroup[] list) {...}
    public int enumerate(ThreadGroup[] list, boolean recurse) {...}
    public final int getMaxPriority() {...}
    public final String getName() {...}
    public final ThreadGroup getParent() {...}
    public final void interrupt() {...}
    public final boolean isDaemon() {...}
    public final void setDaemon(boolean on) {...}
    public final void setMaxPriority(int newPriority) {...}
    public String toString() {...}
    ...
}

public class Object
{
    ...
    public final void notify() {...}
    public final void notifyAll() {...}
    public final void wait() throws InterruptedException {...}
    public final void wait(long millis)
        throws InterruptedException {...}
    public final void wait(long millis, int nanos)
        throws InterruptedException {...}
    ...
}

```

Die Wirkung der behandelten Mechanismen lässt sich zusammenfassend mit einem *Zustandsübergangsdiagramm* (siehe Bild 2.4) darstellen.



**Bild 2.4** Zustandsübergangsdiagramm für Java-Threads

Zunächst muss ein Thread-Objekt erzeugt werden. Damit läuft der Thread aber noch nicht. Der Thread befindet sich im Zustand NEW. Erst wenn die Methode start auf das Thread-Objekt angewendet wird, kann der Thread loslaufen. Das Loslaufen erfolgt aber in der Regel nicht sofort. Ein Thread gelangt deshalb durch start vom Zustand NEW erst in den Zustand READY. In diesem Zustand bleibt der Thread so lange, bis tatsächlich auf ihn umgeschaltet wird. Während der Thread dann tatsächlich ausgeführt wird, befindet er sich im Zustand RUNNING. Die Methode getState der Klasse Thread, mit der die Zustände eines Threads erfragt werden können, unterscheidet die hier eingeführten Zustände READY und RUNNING nicht; diese werden zu einem Zustand RUNNABLE zusammengefasst.

Der Zustand RUNNING kann auf fünf Arten verlassen werden: Erstens könnte die ausgeführte Run-Methode verlassen werden (z.B. durch return oder eine nicht behandelte Ausnahme). In diesem Fall gelangt der Thread in den Zustand TERMINATED. Zweitens könnte der Thread genügend lange ausgeführt worden sein, so dass entschieden wird, auf einen anderen Thread umzuschalten. In diesem Fall wird der Thread verdrängt und wechselt wieder in den Zustand READY (bleibt aber somit RUNNABLE). Die Gründe Nr. 3, 4 und 5 für das Verlassen des RUNNING-Zustands sind, dass der Thread nicht mehr weiterlaufen kann und sich blockiert. Grund Nr. 3 ist, dass ein Thread eine Synchronized-Methode bzw. einen Synchronized-Block betreten will (z.B. auch nach dem Benachrichtigen durch notify oder notifyAll in der Wait-Methode), das betreffende Objekt aber momentan gesperrt ist. In diesem Fall gelangt er in den Zustand BLOCKED, aus dem er wieder in den RUNNABLE-Zustand (genauer in den READY-Zustand) wechselt, wenn die Sperre des betreffenden Objekts freigegeben wird und von diesem Thread dann gesetzt werden kann. Der Grund Nr. 4, warum ein Thread nicht mehr weiterlaufen kann, besteht darin, dass der Thread

durch Aufruf der Methoden `wait` oder `join` in einen Wartezustand gelangt. Dieser Zustand wird `WAITING` genannt. Aus diesem Zustand kehrt der Thread in den `RUNNABLE`- bzw. `READY`-Zustand zurück, indem ein anderer Thread die Methode `notify` oder `notifyAll` aufruft, oder indem der Thread, auf dessen Ende mit `join` gewartet wird, terminiert. Eine weitere Möglichkeit für einen Thread, den Zustand `WAITING` zu verlassen, ist die Anwendung der Methode `interrupt` auf diesen Thread von einem anderen Thread aus. In diesem Fall wird `wait` oder `join` durch eine `InterruptedException` unterbrochen. Der fünfte Grund für eine Blockierung eines Threads ist der Aufruf von `wait` oder `join` mit Angabe einer maximalen Wartezeit oder durch Aufruf von `sleep`. In diesem Fall gelangt der Thread in den Zustand `TIMED_WAITING`. Gründe für das Verlassen dieses Zustands sind dieselben wie diejenigen für das Verlassen des Zustands `WAITING`. Hinzu kommt jetzt als Grund noch der Ablauf der Wartezeit.

In jedem Fall bewirkt das Verlassen des Zustands `RUNNING` durch einen Thread einen Übergang von `READY` nach `RUNNING` für einen anderen Thread.

Befindet sich ein Thread in einem der drei Zustände `WAITING`, `TIMED_WAITING` oder `BLOCKED`, kann nicht auf ihn umgeschaltet werden. Das heißt, wenn ein Thread blockiert ist, wird er nie rechnend und verbraucht keine Rechenzeit. Wir haben schon gesehen, dass dies eine sehr effiziente Art des Wartens auf ein Ereignis ist. Sobald der Grund für die Blockade eines Threads nicht mehr besteht, gelangt er wieder in den Zustand `READY`. Wie schon beim Starten bewirkt das Deblockieren eines Threads also nicht sein unmittelbares Weiterlaufen. Der Thread gelangt erst in den Zustand `READY` und muss sich gedulden, bis wieder auf ihn umgeschaltet wird (d.h. bis er wieder in den Zustand `RUNNING` gelangt).

Mit diesen Konzepten wurden in diesem Kapitel und werden im folgenden Kapitel eine Reihe von Anwendungen entwickelt. Dabei gab es immer *aktive* und *passive Klassen* bzw. *Objekte*. Aktive Klassen sind die Thread-Klassen bzw. die Klassen, welche die `Runnable`-Schnittstelle implementieren. Passive Klassen sind solche, deren Objekte von mehreren Threads benutzt werden. In Tabelle 2.4 sind einige der Anwendungen nach diesem Schema noch einmal zusammenfassend dargestellt.

**Tabelle 2.4** Zusammenfassung einiger Anwendungen dieses Kapitels

Aktive Klassen	Passive Klassen
Bankangestellte (Klasse Clerk)	Bank und Konten (Klassen Bank und Account)
Autos (Klasse Car)	Parkhaus (Klasse ParkingGarage)
Erzeuger und Verbraucher (Klassen Producer und Consumer)	Puffer (Klasse Buffer)

Bitte beachten Sie, dass die Synchronisation (`synchronized`, Aufrufe von `wait`, `notify` und `notifyAll`) in der Regel immer in den passiven Klassen realisiert wird.

# 3

## Fortgeschrittene Synchronisationskonzepte in Java

In diesem Kapitel werden wir weitere Parallelitäts- und Synchronisationskonzepte von Java kennen lernen. Diese basieren zum Teil auf Anwendungen der Synchronisationsprimitive synchronized, wait, notify und notifyAll, mit denen wir uns im vorigen Kapitel beschäftigt haben. Die Inhalte dieses Kapitels sind:

- Zunächst werden Synchronisations- und Kommunikationskonzepte, die aus Unix bzw. Linux bekannt sind, nachgeahmt. Um den Unterschied zwischen diesen von Betriebssystemen bereitgestellten Mechanismen und den bereits kennen gelernten Konzepten in Java deutlich zu machen, sei nochmals daran erinnert, dass es hier in diesem Kapitel immer noch um die Synchronisation von Threads innerhalb eines Prozesses geht, wobei der Zugriff auf gemeinsame Objekte möglich ist, da die Threads sich einen Adressraum teilen (Metapher: alle Köche sind in einer gemeinsamen Küche). Bei den Synchronisations- und Kommunikationsmechanismen der Betriebssysteme geht es vorwiegend um die Synchronisation und Kommunikation zwischen Threads unterschiedlicher Prozesse, die keinen gemeinsamen Adressraum besitzen. Die Nachbildung der Synchronisations- und Kommunikationskonzepte von Unix bzw. Linux in Java ist aber dennoch sinnvoll, denn die Leserinnen und Leser dürfen dadurch ein besseres Verständnis dieser Konzepte erreichen, sie bekommen weiteres Anschauungsmaterial im Umgang mit den Java-Synchronisationsprimitiven, und sie können nicht zuletzt die entwickelten Klassen auch gewinnbringend zur Synchronisation und Kommunikation der Threads innerhalb eines Prozesses einsetzen. Bei den Synchronisations- und Kommunikationskonzepten aus Unix bzw. Linux handelt es sich um Semaphore bzw. Semaphorgruppen (Abschnitt 3.1), Message Queues (Abschnitt 3.2) und Pipes (Abschnitt 3.3).
- Danach werden Lösungen für bekannte „klassische“ Synchronisationsaufgaben angegeben. Diese Aufgaben werden in Lehrbüchern häufig zur Illustration von Synchronisationskonzepten herangezogen. Die in diesem Kapitel besprochenen Lösungen dieser Aufgaben basieren nicht nur direkt auf den Java-Konzepten synchronized, wait, notify und notifyAll, sondern es werden auch Lösungen diskutiert, die die zuvor eingeführten Semaphore benutzen. Die behandelten „klassischen“ Synchronisationsaufgaben sind das Philosophenproblem (Abschnitt 3.4) und das Leser-Schreiber-Problem (Abschnitt 3.5).
- Nachdem eine ganze Reihe von Programmen mit synchronized, wait, notify bzw. notifyAll besprochen wurde, werden anschließend Schablonen für Synchronized-Methoden vorgestellt (Abschnitt 3.6), die typisch sind und die den Lesern helfen sollen, eigene Anwendungen zu programmieren. Alle besprochenen Synchronized-Methoden passen auf diese

Schablonen. In diesem Zusammenhang wird auch das Thema Konsistenz von Objektzuständen angesprochen.

- Seit der Version 5 von Java wurde die Klassenbibliothek um viele nützliche Klassen und Schnittstellen zum Umgang mit Parallelität erweitert (u. a. auch Semaphore). In Abschnitt 3.7 wird ein Überblick über diesen Teil der Klassenbibliothek, die Concurrent-Bibliothek, gegeben. Auf einige ausgewählte Aspekte wird etwas ausführlicher anhand von Beispielprogrammen eingegangen. In Version 7 von Java wurde diese Concurrent-Bibliothek um das Fork-Join-Framework ergänzt. Diesem Framework ist Abschnitt 3.8 gewidmet. Schließlich brachte Java 8 weitere wichtige Neuerungen zur Fließbandverarbeitung in Form des Data-Streaming-Frameworks und der CompletableFuture, die in Abschnitt 3.9 und 3.10 präsentiert werden.
- Zum Abschluss wird der Themenbereich Verklemmungen behandelt (Ursachen, s. Abschnitt 3.11, und Strategien zur Vermeidung, s. Abschnitt 3.12).

## ■ 3.1 Semaphore

### 3.1.1 Einfache Semaphore

Das klassische Synchronisationskonzept schlechthin ist der *Semaphor* (engl. *semaphore*). Glücklicherweise kennen wir dieses Konzept schon, und zwar als Parkhaus. Aus der Klasse ParkingGarage kann lediglich durch Änderung der verwendeten Bezeichner für die Klasse, das Attribut und die Methoden eine korrekte Implementierung eines Semaphors erzeugt werden (s. Listing 3.1):

**Listing 3.1**

```
public class Semaphore
{
    private int value;

    public Semaphore(int init)
    {
        if(init < 0)
        {
            throw new IllegalArgumentException("Parameter < 0");
        }
        value = init;
    }

    public synchronized void p()
    {
        while(value == 0)
        {
            try
            {
                wait();
            }
```

```

        catch(InterruptedException e)
        {
        }
    }
    value--;
}

public synchronized void v()
{
    value++;
    notify();
}
}

```

Ein Semaphor ist von der Idee her ein abstraktes Konzept. Als Attribut enthält jedes Objekt der Klasse Semaphore einen Wert des Typs int, der nie negativ werden kann. Über den Konstruktor lässt sich ein initialer Wert einstellen. Ferner existieren Methoden zum Herunter- bzw. Hochzählen des Attributwerts. Die „klassische“ Bezeichnung für diese Methoden lautet *p* und *v* (manchmal auch *down* und *up*, seit Java 5 *acquire* und *release*, s. Abschnitt 3.7). Die Bezeichnungen *p* und *v* stammen von den holländischen Begriffen *passeeren* (passieren) und *vrijgeven* (freigeben). Beim Aufruf der Methode *p* wird der aufrufende Thread blockiert, falls der Wert des Semaphors durch das Erniedrigen negativ würde. Ein von einem anderen Thread ausgeführter *V*-Aufruf weckt einen wartenden Thread wieder auf. Eine Rückbesinnung auf das Parkhaus hilft den Leserinnen und Lesern hoffentlich, dieses etwas abstraktere Konzept eines Semaphors und die Wirkung der Methoden *p* und *v* zu verstehen und in Erinnerung zu behalten.

### 3.1.2 Einfache Semaphore für den gegenseitigen Ausschluss

Semaphore werden häufig zur Realisierung des *gegenseitigen Ausschlusses* (engl. *mutual exclusion*) eingesetzt. Damit ist gemeint, dass ein bestimmtes Programmstück zu einer Zeit nur von höchstens einem Thread ausgeführt werden kann (s. Listing 3.2). Eine Synchronized-Methode stellt ebenfalls eine Form des gegenseitigen Ausschlusses dar: Auf ein Objekt kann diese Methode zu einem Zeitpunkt höchstens von einem Thread angewendet werden. Im Kontext unterschiedlicher Prozesse und im Kontext anderer Programmiersprachen als Java kann auf synchronized nicht zurückgegriffen werden. Eine Standard-Lösung ist der Einsatz eines so genannten *Mutex-Semaphors* (*Mutex* steht für MUTual EXclusion), der im Folgenden wieder für Threads eines einzigen Prozesses und mit Java gezeigt wird. Die Idee dieses Beispiels lässt sich aber auf einen allgemeineren Kontext übertragen.

#### **Listing 3.2**

```

class MutexThread extends Thread
{
    private Semaphore mutex;

    public MutexThread(Semaphore mutex)
    {
        this.mutex = mutex;
        start();
    }
}

```

```

    }

    public void run()
    {
        while(true)
        {
            mutex.p();
            System.out.println("kritischen Abschnitt betreten");
            try
            {
                sleep((int)(Math.random() * 1000));
            }
            catch(InterruptedException e)
            {
            }
            System.out.println("kritischer Abschnitt wird "
                               + "verlassen");
            mutex.v();
        }
    }

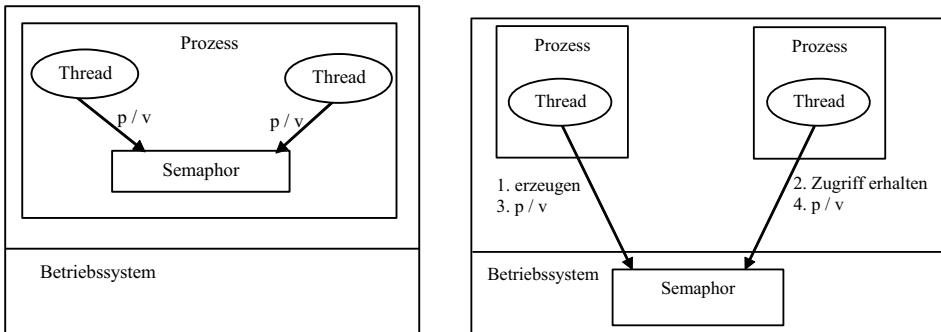
    public class MutualExclusion
    {
        public static void main(String[] args)
        {
            Semaphore mutex = new Semaphore(1);
            for(int i = 1; i <= 10; i++)
            {
                new MutexThread(mutex);
            }
        }
    }
}

```

Das Programmstück, das zu einem Zeitpunkt nur von höchstens einem Thread ausgeführt werden darf, wird in der Regel *kritischer Abschnitt* genannt. In diesem Beispiel schläft der ausführende Thread eine zufällig gewählte Zeit von maximal einer Sekunde. In einer realen Anwendung würde an dieser Stelle zum Beispiel der exklusive Zugriff auf gemeinsame Daten erfolgen. Eine Referenz auf den von allen Threads benutzten Semaphor wird auch in diesem Programm wieder im Konstruktor übergeben. Außerdem befindet sich im Konstruktor der Aufruf der Methode start. In der Main-Methode wird der Semaphor erzeugt. Wichtig hierbei ist der Initialisierungswert 1. Der Initialisierungswert gibt an, wie viele Threads maximal gleichzeitig den kritischen Abschnitt durchlaufen können (größere Werte als 1 sind denkbar, man kann damit festlegen, dass z.B. maximal 5 Threads gleichzeitig etwas tun dürfen). Ist dieser Wert 0, so kann nie ein Thread den kritischen Abschnitt betreten.

Es sei daran erinnert, dass der hier implementierte Semaphor zwar dem Vorbild eines Betriebssystem-Semaphors nachempfunden wurde, dass aber nur Threads desselben Prozesses auf Objekte dieser Klasse zugreifen können (s. Bild 3.1 links). Threads aus unterschiedlichen Prozessen können ein Semaphor-Objekt unserer Klasse jedoch nicht gemeinsam nutzen (diese Einschränkung wird in Kapitel 6 aufgehoben). Ein Betriebssystem-Semaphor ist aber sehr wohl von Threads unterschiedlicher Prozesse zugreifbar. In einem solchen Fall existiert der Semaphor im Betriebssystemkern (s. Bild 3.1 rechts). Er wird nicht wie in obigem Beispiel über eine direkte Referenz angesprochen, sondern durch eine Art von Ken-

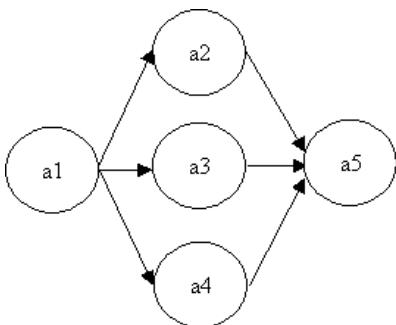
nung (oder Nummer), die als Parameter des Systemaufrufs mit übergeben wird. Ferner ist übrigens der Zugriff auf gemeinsame Daten von Threads unterschiedlicher Prozesse möglich über die Nutzung von gemeinsamem Speicher, der ebenfalls über Systemaufrufe eingerichtet wird. Wir wollen auf diese Aspekte aber nicht näher eingehen.



**Bild 3.1** Nutzung eines Java-Semaphors durch Threads desselben Prozesses (links) und Nutzung eines Betriebssystem-Semaphors durch Threads unterschiedlicher Prozesse (rechts)

### 3.1.3 Einfache Semaphore zur Herstellung vorgegebener Ausführungsreihenfolgen

Ein zweiter gängiger Einsatz von Semaphoren neben dem gegenseitigen Ausschluss ist der zur Herstellung einer bestimmten Ausführungsreihenfolge von Aktionen in unterschiedlichen Threads. Als Beispiel sind in Bild 3.2 unterschiedliche Aktionen a1 bis a5 gezeigt, die alle in jeweils unterschiedlichen Threads t1 bis t5 ausgeführt werden sollen. Die Pfeile geben eine zeitliche Relation vor. So muss zum Beispiel a1 vor a2 ausgeführt werden, während es zwischen a2 und a3 keine Vorschrift für die zeitliche Beziehung gibt. Das bedeutet, dass a2 und a3 echt parallel oder in einer nicht vorgegebenen zeitlichen Beziehung ausgeführt werden können.



**Bild 3.2**

Beispiel für eine vorgegebene zeitliche Beziehung von auszuführenden Aktionen in unterschiedlichen Threads

Das folgende Programm (Listing 3.3) realisiert die vorgegebene zeitliche Relation zwischen den Aktionen. Dabei wird für jeden Pfeil aus Bild 3.2 ein Semaphor verwendet. Bevor eine Aktion ausgeführt werden kann, wird die P-Methode für alle Semaphore, die den ankom-

menden Pfeilen entsprechen, ausgeführt. Entsprechend wird nach einer Aktion die V-Methode auf allen Semaphoren ausgeführt, die den von dieser Aktion abgehenden Pfeilen entsprechen. Der Einheitlichkeit und Einfachheit halber werden allen Threads Referenzen auf alle Semaphore in einem Feld übergeben, obwohl nicht alle Threads alle Semaphore benutzen.

**Listing 3.3**

```
class T1 extends Thread
{
    private Semaphore[] sems;

    public T1(Semaphore[] sems, String name)
    {
        super(name);
        this.sems = sems;
        start();
    }

    private void a1()
    {
        System.out.println("a1");
    }

    public void run()
    {
        a1();
        sems[0].v();
        sems[1].v();
        sems[2].v();
    }
}

class T2 extends Thread
{
    private Semaphore[] sems;

    public T2(Semaphore[] sems, String name)
    {
        super(name);
        this.sems = sems;
        start();
    }

    private void a2()
    {
        System.out.println("a2");
    }

    public void run()
    {
        sems[0].p();
        a2();
        sems[3].v();
    }
}
```

```
class T3 extends Thread
{
    private Semaphore[] sems;

    public T3(Semaphore[] sems, String name)
    {
        super(name);
        this.sems = sems;
        start();
    }

    private void a3()
    {
        System.out.println("a3");
    }

    public void run()
    {
        sems[1].p();
        a3();
        sems[4].v();
    }
}

class T4 extends Thread
{
    private Semaphore[] sems;

    public T4(Semaphore[] sems, String name)
    {
        super(name);
        this.sems = sems;
        start();
    }

    private void a4()
    {
        System.out.println("a4");
    }

    public void run()
    {
        sems[2].p();
        a4();
        sems[5].v();
    }
}

class T5 extends Thread
{
    private Semaphore[] sems;

    public T5(Semaphore[] sems, String name)
    {
        super(name);
        this.sems = sems;
        start();
    }
}
```

```

private void a5()
{
    System.out.println("a5");
}

public void run()
{
    sems[3].p();
    sems[4].p();
    sems[5].p();
    a5();
}
}

public class TimingRelation
{
    public static void main(String[] args)
    {
        Semaphore[] sems = new Semaphore[6];
        for(int i = 0; i < sems.length; i++)
        {
            sems[i] = new Semaphore(0);
        }
        new T1(sems, "T1");
        new T2(sems, "T2");
        new T3(sems, "T3");
        new T4(sems, "T4");
        new T5(sems, "T5");
    }
}

```

Wie leicht nachzuvollziehen ist, wird der Semaphor im Feld mit dem Index 0 für die zeitliche Relation zwischen a1 und a2 verwendet. Entsprechendes gilt für die Semaphore mit den Indizes 1, 2, 3, 4 und 5, die für die zeitliche Relation zwischen a1 und a3, a1 und a4, a2 und a5, a3 und a5 sowie a4 und a5 eingesetzt werden. Beachten Sie den richtigen Initialisierungswert 0 beim Erzeugen der Semaphore in main (im Unterschied zum Initialisierungswert 1 beim gegenseitigen Ausschluss).

### 3.1.4 Additive Semaphore

Ein Sonderfall der Semaphore, den wir nicht extra programmieren, stellen *binäre Semaphore* dar. In binären Semaphoren kann das Semaphorattribut nur die beiden Werte 0 oder 1 annehmen. Dies reicht in den beiden obigen Beispielen, gegenseitiger Ausschluss und vorgegebene Reihenfolge, aus. Im Gegensatz zu dieser Spezialisierung sind *additive Semaphore* eine Verallgemeinerung der bisher kennen gelernten Semaphore. Die Verallgemeinerung besteht darin, dass der Wert des Semaphors nicht nur um eins erhöht oder erniedrigt werden kann, sondern um beliebige Werte. Additive Semaphore können realisiert werden, indem die Methoden p und v einen Int-Parameter erhalten, der angibt, um wie viel der Attributwert des Semaphors erniedrigt bzw. erhöht werden soll. Alternativ dazu genügt eine einzige Methode mit einem Int-Argument, das die Wertänderung (positiv oder negativ)

angibt. Im folgenden Programm (Listing 3.4) werden beide Alternativen gleichzeitig realisiert, wobei zusätzlich die „alten“ P- und V-Methoden ohne Argumente aus Kompatibilitätsgründen auch noch vorhanden sind.

**Listing 3.4**

```
public class AdditiveSemaphore
{
    private int value;

    public AdditiveSemaphore(int init)
    {
        if(init < 0)
        {
            throw new IllegalArgumentException("Parameter < 0");
        }
        this.value = init;
    }

    public synchronized void p(int x)
    {
        if(x <= 0)
        {
            throw new IllegalArgumentException("Parameter <= 0");
        }
        while(value - x < 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        value -= x;
    }

    public synchronized void v(int x)
    {
        if(x <= 0)
        {
            throw new IllegalArgumentException("Parameter <= 0");
        }
        value += x;
        notifyAll(); // nicht notify
    }

    public void p()
    {
        p(1);
    }

    public void v()
    {
        v(1);
    }
}
```

```

public void change(int x)
{
    if(x > 0)
    {
        v(x);
    }
    else if(x < 0)
    {
        p(-x);
    }
}

```

Offensichtlich sind nicht alle Methoden mit synchronized gekennzeichnet. Dies ist auch nicht nötig, da nur in den Methoden p und v mit Int-Argument auf das Value-Attribut zugegriffen wird. Es wäre allerdings nicht falsch, alle Methoden mit synchronized zu versehen, denn wie in Abschnitt 2.3 besprochen wurde, ergeben sich keine Probleme, wenn eine Synchronized-Methode eine andere Synchronized-Methode für dasselbe Objekt aufruft. Beachten Sie, dass in der V-Methode mit Int-Argument notifyAll verwendet wird. Die Verwendung von notify wäre hier nicht korrekt, da in diesem Fall unter Umständen mehrere Threads weiterlaufen können. Außerdem gibt es mehrere Wartebedingungen (vgl. Abschnitt 2.6), auch wenn im Code nur eine einzige While-Wait-Schleife vorkommt. Die While-Bedingung ist nämlich über x parametrisiert:

```
value - x < 0
```

Ein Thread wartet also z.B., bis value mindestens den Wert 1 hat (Aufruf der Methode mit Parameterwert 1), während ein anderer Thread wartet, bis value mindestens den Wert 5 hat (Aufruf der Methode mit Parameterwert 5).

Die Methode v erwartet ein positives Argument. Denn sonst könnte man mit der V-Methode z.B. den Wert des Attributs erniedrigen durch Angabe eines negativen Argumentwerts. Der Wert des Value-Attributs könnte somit sogar negative Werte annehmen. Aber auch in p sind nur positive Argumente zugelassen, denn bei Angabe eines negativen Werts würde der Attributwert hochgezählt, ohne dass wartende Threads geweckt würden. Deshalb wird sowohl am Anfang von p als auch am Anfang von v überprüft, ob der Argumentwert positiv ist. Ist er negativ oder null, so wird die aus dem vorhergehenden Kapitel bereits bekannte Ausnahme IllegalArgumentException geworfen.

Die Methode change bietet die Möglichkeit, den Wert des Value-Arguments zu erhöhen oder zu erniedrigen, je nach dem, ob der Argumentwert positiv oder negativ ist. Bei der Angabe eines negativen Argumentwerts (z.B. -3) wird der Auftrag erteilt, den Wert des Attributs um 3 zu erniedrigen. Also wird die P-Methode mit dem positiven Argumentwert +3 aufgerufen. Wird change mit positivem Argumentwert versorgt, so wird dieser an die Methode v durchgereicht.

Die While-Bedingung in der P-Methode prüft ab, ob nach Subtraktion des Arguments x das Value-Attribut negativ werden würde. Wenn dem so ist, so wird der Wert zunächst nicht verändert, sondern es wird stattdessen gewartet, bis die Subtraktion möglich ist, ohne dass der Wert negativ wird. Sie sehen anhand dieser Erläuterungen des Programmcodes, dass der Wert des Attributs nicht schrittweise erniedrigt wird, sondern „auf einen Schlag“. Dies

bedeutet, dass es einen großen Unterschied ausmachen kann, ob ein Semaphorwert durch einen Aufruf „`p(3)`“ oder durch 3 aufeinanderfolgende Aufrufe „`p(1)`“ bzw. „`p()`“ erniedrigt wird.

Betrachten wir dazu ein Beispiel: Angenommen, das Attribut eines additiven Semaphors, der von zwei Threads gemeinsam benutzt wird, habe den Wert 4. Nehmen wir weiter an, die beiden Threads möchten den Semaphor jeweils um 3 erniedrigen. Wird dies schrittweise durch drei aufeinanderfolgende Aufrufe „`p(1)`“ realisiert, so kann es zu einer Verklemmung kommen, wenn z. B. der erste Thread den Wert des Semaphors zweimal um jeweils 1 erniedrigt hat und dann auf den zweiten Thread umgeschaltet wird. Der zweite Thread kann den Semaphorwert dann ebenfalls zwei Mal um jeweils 1 erniedrigen, er wird aber beim dritten Versuch blockiert. Wenn dann auf den ersten Thread zurückgeschaltet wird, so wird auch dieser versuchen, den Wert des Semaphors ein drittes Mal um 1 zu erniedrigen, was ebenfalls nicht möglich ist und zum Blockieren des Aufrufers führt. Beide Threads warten dann gegenseitig darauf, dass der jeweils andere Thread den Semaphorwert wieder erhöht. Dies ist eine *Verklemmungssituation* (siehe Abschnitt 3.10). Wenn dagegen die beiden Threads den Semaphorwert durch „`p(3)`“ erniedrigen, so kann es zu keiner Verklemmung kommen, denn einer der beiden Threads kann den Semaphor in jedem Fall erniedrigen, der andere muss warten. Nach einer gewissen Zeit wird der eine Thread den Semaphor wieder erhöhen, so dass der andere Thread ebenfalls weiterlaufen kann – eine Verklemmung tritt in diesem Fall nicht ein.

### 3.1.5 Semaphorgruppen

Diese grundlegende Eigenschaft des Durchführens einer Änderung „auf einen Schlag“ („alles oder nichts“) ist auch die wesentliche Motivation für die Einführung so genannter *Semaphorgruppen*. In Unix bzw. Linux werden diese Gruppen einfach Semaphore genannt. In diesem Buch sprechen wir aber der Deutlichkeit halber von Semaphorgruppen. Semaphorgruppen stellen eine Verallgemeinerung der additiven Semaphore dar. Es ist möglich, mit dem Aufruf der Change-Methode mehrere Semaphore der Gruppe „auf einen Schlag“ zu verändern (zu erhöhen oder zu erniedrigen). Alle Änderungen werden nur durchgeführt, wenn nach der Änderung die Werte aller Semaphore der Gruppe nicht negativ sind. Andernfalls wird gewartet, ohne dass irgendeine Änderung vorgenommen wird. Das heißt, dass in diesem Fall zunächst keine Änderungen an den Semaphorwerten durchgeführt werden, ganz unabhängig davon, ob dies für einzelne Semaphore der Gruppe möglich wäre oder nicht (es werden also auch keine Erhöhungen vorgenommen, die für einen einzelnen Semaphore ja immer möglich sind). Wie bei den additiven Semaphoren kann es bezüglich der Verklemmungsproblematik einen großen Unterschied ausmachen, ob eine Semaphorgruppe oder eine Reihe einzelner Semaphore verwendet wird. Wir werden darauf bei der Besprechung des so genannten Philosophenproblems in Abschnitt 3.4 näher eingehen.

Im Folgenden ist der Programmcode der Klasse `SemaphoreGroup` dargestellt (Listing 3.5).

### Listing 3.5

```
public class SemaphoreGroup
{
    private int[] values;

    public SemaphoreGroup(int numberOfMembers)
    {
        if(numberOfMembers <= 0)
        {
            throw new IllegalArgumentException("Parameter <= 0");
        }
        values = new int[numberOfMembers];
    }

    public int getNumberOfMembers()
    {
        return values.length;
    }

    public synchronized void changeValues(int[] deltas)
    {
        if(deltas.length != values.length)
        {
            throw new IllegalArgumentException("falsche Feldlänge");
        }
        while(!canChange(deltas))
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        doChange(deltas);
        notifyAll();
    }

    private boolean canChange(int[] deltas)
    {
        for(int i = 0; i < values.length; i++)
        {
            if(values[i] + deltas[i] < 0)
            {
                return false;
            }
        }
        return true;
    }

    private void doChange(int[] deltas)
    {
        for(int i = 0; i < values.length; i++)
        {
            values[i] = values[i] + deltas[i];
        }
    }
}
```

Wie am Programmcode zu erkennen ist, basiert die Implementierung der Klasse SemaphoreGroup nicht auf einzelnen Semaphoren, denn – wie oben erläutert – die Wirkung der Methode changeValues ist nicht gleichbedeutend mit einer Hintereinanderausführung mehrerer P- und V-Methodenaufrufe, die auf einzelne Semaphore angewendet werden. Die Werte der Semaphore werden in einem Int-Feld namens values gehalten. Die Größe des Feldes (und damit die Anzahl der Mitglieder der Semaphorgruppe) wird im Konstruktor über einen Parameter bestimmt. Die Klasse SemaphoreGroup besitzt keine P- und V-Methoden mehr, sondern neben der Methode getNumberOfMembers zum Abfragen der Anzahl der enthaltenen Semaphore nur eine einzige weitere öffentliche Methode changeValues mit einem Int-Feld als Parameter, das genau so lang sein muss wie das Values-Feld (eine tolerantere Implementierung wäre leicht möglich). Der Parameter der Methode changeValues gibt an, um welchen Wert die Semaphore jeweils verändert werden sollen (d. h. deltas[i] gibt an, um wie viel values[i] verändert werden soll). Die Werte der Feldelemente des Parameters können dabei positiv, negativ oder null sein. Im letzten Fall bleibt der Wert des entsprechenden Semaphors unverändert. Die private Methode canChange liefert true zurück, falls alle Änderungen durchgeführt werden können, sonst false. Die private Methode doChange führt alle Änderungen durch. In changeValues wird mit canChange geprüft, ob alle Änderungen durchführbar sind. Falls dies nicht der Fall ist, wird gewartet, ohne irgend etwas verändert zu haben. Nach der Durchführung der Änderungen werden alle an diesem Objekt wartenden Threads durch notifyAll geweckt. Es muss notifyAll statt notify verwendet werden, da zum einen die durchgeführten Änderungen mehreren Threads das Weiterlaufen ermöglichen können, und da zum anderen die wartenden Threads auf unterschiedliche Bedingungen warten. So kann es vorkommen, dass z. B. ein Thread darauf wartet, den Wert des ersten Semaphors der Gruppe zu erniedrigen, während ein anderer Thread den Wert des zweiten Semaphors verringern will. Wenn nun ein Thread den Wert des zweiten Semaphors erhöht, so kann es bei der Nutzung von notify vorkommen, dass der falsche Thread, im Beispiel der erste, geweckt wird. Dieser kann aber nicht weiterlaufen und blockiert sich erneut. Der zweite Thread könnte weiterlaufen, wurde aber nicht geweckt.

In unserer Merkregel, wann notifyAll verwendet werden muss (s. Abschnitt 2.6), bedeutet also die Bedingung, dass mehrere Wartebedingungen vorhanden sein müssen, nicht, dass in der betreffenden Klasse nur an einer Stelle eine While-Wait-Schleife vorkommen darf. Wenn wie im Fall der Klasse SemaphoreGroup die While-Bedingung parametrisiert ist, dann gibt es durch den Aufruf der umgebenden Methode mit unterschiedlichen Parametern unterschiedliche Wartebedingungen (dasselbe haben wir auch schon im Zusammenhang mit additiven Semaphoren gesehen).

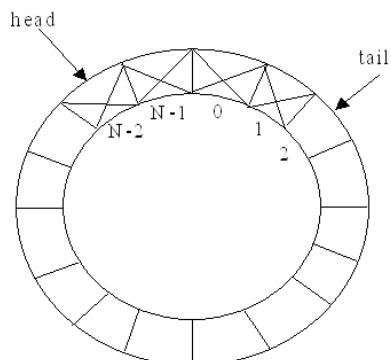
Wenn in changeValues alle Semaphorwerte nur erniedrigt werden, ist kein notifyAll nötig. In der Implementierung der Klasse SemaphoreGroup wurde dieser Fall aber nicht gesondert behandelt. Es wird stattdessen nach jeder Änderung notifyAll aufgerufen, wodurch die Implementierung einfacher gehalten werden konnte.

Semaphore dienen in ihren verschiedenen Ausprägungen der Synchronisation von Threads. Wir wenden uns nun Kommunikationskonzepten zu. Neben der ebenfalls vorhandenen Synchronisation findet hier eine zusätzliche Übertragung von Daten statt. Oder anders formuliert: Semaphore dienen zum Senden (v) und Empfangen (p) von Nachrichten ohne Inhalt (das Attribut value zählt die Anzahl der gesendeten und noch nicht empfangenen Nachrichten).

## ■ 3.2 Message Queues

### 3.2.1 Verallgemeinerung des Erzeuger-Verbraucher-Problems

Zur Vorbereitung unserer Implementierung von *Message Queues* (Nachrichtenwarteschlangen) erweitern wir die Funktionalität der Klasse Buffer aus Abschnitt 2.6. Die Klasse besitzt die Methoden put und get, um einen Zahlenwert in einem Puffer abzulegen bzw. einen Zahlenwert aus einem Puffer auszulesen. Wenn beim Ablegen eines Werts der Puffer voll ist (d. h. der vorhergehende Wert wurde noch nicht abgeholt), dann wird der Aufrufer so lange blockiert, bis der Puffer frei wird. Umgekehrt wird ein Thread blockiert, falls er einen Wert aus einem leeren Puffer lesen möchte (d. h. aus einem Puffer, der seit dem letzten Lesen nicht mehr neu gefüllt wurde). Die in Abschnitt 2.6 besprochene Klasse stellt nur einen einzigen Pufferplatz bereit. Eine Verallgemeinerung der Klasse Buffer erhält man, wenn man Pufferplatz nicht nur für einen Zahlenwert, sondern für N Werte bereitstellt. Das bedeutet, dass ein *Erzeuger* einen Wert ablegen kann und anschließend auch noch einen zweiten, selbst dann, wenn der erste Wert in der Zwischenzeit noch nicht von einem *Verbraucher* abgeholt wurde. Die Verbraucher wollen die Werte in der Reihenfolge abholen, in der sie in den Puffer gelegt wurden. Bei einer Realisierung des Puffers durch ein Feld könnte man z. B. den abzuholenden Wert immer dem Feldelement mit dem Index 0 entnehmen und die abzulegenden Werte immer auf das erste freie Feldelement (das freie Feldelement mit dem kleinsten Index) speichern. Dann müssten aber beim Entnehmen eines Werts alle anderen Werte um eins nachgeschoben werden. Dies kann man vermeiden, wenn man das Feld zyklisch benutzt (d. h. nach Benutzung des letzten Feldelements verwendet man wieder das erste). Zum besseren Verständnis kann man sich das Feld in diesem Fall zu einem Kreis gebogen denken, wie dies in Bild 3.3 zu sehen ist.



**Bild 3.3**

Nutzung eines Feldes der Größe N zur Pufferung

Das Attribut head gibt den Index des ersten (d. h. vorderen) Elements an (d. h. das Element, das als nächstes abgeholt wird). Entsprechend gibt das Attribut tail den Index an, auf den das nächste Element abgelegt werden kann. Nach jedem Abholen bzw. Ablegen wird head bzw. tail um eins erhöht, wobei dabei darauf geachtet werden muss, dass man beim Anstoßen am Feldende wieder bei null beginnt.

Mit diesen Ausführungen dürfte die Klasse BufferN (Listing 3.6) nun einfach zu verstehen sein.

**Listing 3.6**

```
public class BufferN
{
    private int head;
    private int tail;
    private int numberOfElements;
    private int[] data;

    public BufferN(int n)
    {
        // Abprüfen von n <= 0 ausgelassen ...
        data = new int[n];

        //nicht nötig, da automatisch mit 0 initialisiert:
        head = 0;
        tail = 0;
        numberOfElements = 0;
    }

    public synchronized void put(int x)
    {
        while(numberOfElements == data.length)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        data[tail++] = x;
        if(tail == data.length)
        {
            tail = 0;
        }
        numberOfElements++;
        notifyAll();
    }

    public synchronized int get()
    {
        while(numberOfElements == 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        int result = data[head++];
        if(head == data.length)
```

```
{  
    head = 0;  
}  
numberOfElements--;  
notifyAll();  
return result;  
}  
}
```

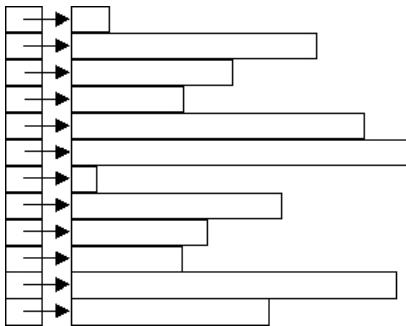
Die Größe des Felds muss im Konstruktor als Parameter angegeben werden. Beim Ablegen eines Werts muss gewartet werden, falls alle Elemente belegt sind. Entsprechend muss beim Abholen gewartet werden, falls kein neues, noch nicht ausgelesenes Element vorhanden ist. Die Notwendigkeit von `notifyAll` statt `notify` wurde schon im Abschnitt 2.6 ausführlich besprochen.

Auf das Attribut `tail` könnte übrigens verzichtet werden, denn dieses ließe sich auch aus `head` und `numberOfElements` berechnen. Gleiches gilt für das Attribut `head`, das man aus `tail` und `numberOfElements` bestimmen könnte. Auf `numberOfElements` kann allerdings nicht verzichtet werden, denn sowohl bei vollem als auch bei leerem Puffer sind `head` und `tail` jeweils gleich und folglich kann der Wert von `numberOfElements` aus `head` und `tail` nicht eindeutig bestimmt werden.

### 3.2.2 Übertragung des erweiterten Erzeuger-Verbraucher-Problems auf Message Queues

Die Synchronisation bei *Message Queues* und Pipes entspricht derjenigen des verallgemeinerten *Erzeuger-Verbraucher-Problems*, wie wir es soeben in Gestalt der Klasse `BufferN` kennen gelernt haben. Der wesentliche Unterschied zwischen `BufferN` einerseits und `MessageQueues` und Pipes andererseits ist die Struktur der übertragenen Daten. Während es bei `BufferN` um Int-Zahlen geht, handelt es sich bei den `MessageQueues` und Pipes um beliebige Daten variabler Länge. Wir werden diese Daten hier als Byte-Felder repräsentieren.

Bei den `Message Queues` bleibt ein gesendetes Byte-Feld als Einheit erhalten, so dass beim Empfangen genau dieses Feld zurückgeliefert wird. Jedes in der `MessageQueue` zwischengepufferte Byte-Feld wird wiederum in einem Feld abgelegt. Dieses Feld entspricht dem Attribut aus der Klasse `BufferN`. Allerdings handelt es sich hierbei nicht um ein Int-Feld, sondern um ein Feld von Byte-Feldern. Das Attribut der Klasse `MessageQueue` ist also ein zweidimensionales Byte-Feld. In Java sind zweidimensionale Felder immer Felder von Feldern, die nicht rechteckig sein müssen, sondern die am rechten Rand ausgefranst sein können, wie dies in Bild 3.4 zu sehen ist.

**Bild 3.4**

Feld von Feldern in Java

Nicht rechteckige, zweidimensionale Felder können in Java z. B. so angelegt werden:

```
byte[][] array2dim = new byte[10][];
for(int i = 0; i < array2dim.length; i++)
{
    array2dim[i] = new byte[10+i];
}
```

Die erste Zeile in obigem Programmfragment erzeugt ein Feld der Größe 10, wobei in jedes Feldelement eine Referenz auf ein Byte-Feld abgespeichert werden kann. In der For-Schleife werden eindimensionale Byte-Felder der Länge 10, 11, 12 usw. erzeugt und die Referenzen auf diese Felder in die Feldelemente von array2dim abgespeichert.

Die Klasse MessageQueue besitzt eine Send-Methode, mit der ein Byte-Feld in einer Message-Queue zur Zwischenspeicherung abgelegt werden kann. Da der Sender-Thread nach dem Senden den Inhalt des gesendeten Byte-Felds manipulieren kann, ist es ratsam, eine Kopie des Byte-Felds beim Senden anzulegen und diese Kopie zu puffern. Die Methode receive der Klasse MessageQueue liefert dann diese Kopie zurück.

Eine MessageQueue im Betriebssystem Unix bzw. Linux ist voll, wenn die Summe der Größe der gepufferten Nachrichten eine bestimmte Schwelle überschreitet. Das heißt, dass eine MessageQueue viele kleine oder wenige große Nachrichten speichern kann. In der Implementierung der MessageQueues in diesem Buch wird der Einfachheit halber nur auf die Anzahl der Nachrichten, nicht aber auf deren Größe, geachtet.

Damit dürfte der Programmcode der Klasse MessageQueue (Listing 3.7) ohne weitere Erläuterungen verständlich sein.

### **Listing 3.7**

```
public class MessageQueue
{
    private byte[][] msgQueue = null;
    private int qsize = 0;
    private int head = 0;
    private int tail = 0;

    public MessageQueue(int capacity)
    {
        // Abprüfen von capacity <= 0 ausgelassen ...
        msgQueue = new byte[capacity][];
    }
}
```

```
public int getCapacity()
{
    return msgQueue.length;
}
public synchronized void send(byte[] msg)
{
    while(qsize == msgQueue.length)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    msgQueue[tail] = new byte[msg.length];
    for(int i = 0; i < msg.length; i++)
    {
        msgQueue[tail][i] = msg[i];
    }
    qsize++;
    tail++;
    if(tail == msgQueue.length)
    {
        tail = 0;
    }
    notifyAll(); // nicht notify
}
public synchronized byte[] receive()
{
    while(qsize == 0)
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    byte[] result = msgQueue[head];
    msgQueue[head] = null;
    qsize--;
    head++;
    if(head == msgQueue.length)
    {
        head = 0;
    }
    notifyAll(); // nicht notify
    return result;
}
```

## ■ 3.3 Pipes

Bei den MessageQueues bleiben die gesendeten Nachrichten, die durch Byte-Felder repräsentiert werden, als Einheiten erhalten. Man spricht in diesem Fall von *nachrichtenorientierter* Kommunikation. Im Gegensatz dazu bieten *Pipes* ein *datenstromorientiertes* Kommunikationsmodell an. Damit ist gemeint, dass der Empfänger einen Datenstrom entgegennimmt, dem er nicht ansieht, in welchen Portionen die Daten vom Sender gesendet wurden. Zum besseren Verständnis dieses Sachverhalts kann man sich vorstellen, dass die beiden Kommunikationspartner durch einen Wasserschlauch verbunden sind. Auf der Senderseite des Schlauchs sei ein Trichter aufgesteckt. Das Senden von Daten entspricht dem Eingießen einer gewissen Menge von Wasser aus einem Becher oder Eimer in den Trichter. Am anderen Ende des Schlauchs sei ein Wasserhahn angebracht, den der Empfänger aufdreht, wenn er Wasser braucht. Dabei ist dem entnommenen Wasser nicht anzusehen, aus wie viel Schüttungen es stammt (d. h. ob es nur ein Teil des Wassers ist, das auf einmal eingegossen wurde oder ob es Wasser ist, das aus mehreren Eingießaktionen stammt).

Einem ähnlichen Unterschied wie hier werden wir im Kapitel 5 bei der Kommunikation von Threads über Rechnergrenzen hinweg mit Hilfe von Sockets begegnen. Die nachrichtenorientierte Kommunikation, die wir in Form von Message Queues kennen gelernt haben, werden wir bei der Kommunikation über das UDP-Protokoll haben, während das TCP-Protokoll datenstromorientiert ist wie die Pipes.

Zur Speicherung der gesendeten Daten verwenden wir bei den Pipes statt eines zweidimensionalen ein eindimensionales Feld. Die gesendeten Daten werden einfach hintereinander in das Feld geschrieben (wieder in zyklischer Weise), so dass die Nachrichtengrenzen nicht mehr zu erkennen sind. Das Senden erfolgt in der Regel als eine unteilbare Aktion. Dies bedeutet: Wenn beim Senden die Länge der zu sendenden Daten größer ist als der im Puffer verfügbare Platz, so wird der Sender-Thread blockiert, ohne auch nur ein Byte gesendet zu haben. Erst wenn für alle zu sendenden Daten genügend Platz im Puffer vorhanden ist, werden die Daten „auf einen Schlag“ in den Puffer kopiert.

Wenn aber die Länge der zu sendenden Daten größer ist als der insgesamt zur Verfügung stehende Speicherplatz, dann würde das beschriebene Verhalten dazu führen, dass der Sender-Thread endlos blockiert bleibt, denn das „Senden auf einen Schlag“ ist in diesem Fall nie möglich. Aus diesem Grund wird in diesem Fall auf ein anderes Sendeverhalten umgeschaltet, nämlich auf das Senden in Portionen. Die Methode `send` wird dadurch komplizierter. So lange nur Nachrichten mit einer nicht die Pufferlänge überschreitenden Größe gesendet werden, befinden diese sich alle an einem Stück im Puffer. Andernfalls kann es vorkommen, dass in einer gesendeten Nachricht Stücke anderer Nachrichten eingestreut sind.

Auch das Empfangen weist einige Besonderheiten auf. Als Parameter wird die Anzahl der zu empfangenden Bytes angegeben. Ist der Puffer ganz leer, so wird der Thread so lange blockiert, bis Daten im Puffer vorhanden sind. Es werden dann so viele Bytes zurückgeliefert wie verlangt wurde. Wenn allerdings so viele Daten nicht im Speicher stehen, dann werden nur alle vorhandenen Bytes zurückgegeben. Es wird also beim Empfangen nur gewartet, wenn der Puffer ganz leer ist. Es wird aber nicht so lange gewartet, bis so viele Daten wie verlangt vorhanden sind. Die Daten werden aus dem Puffer in ein neu erzeugtes Feld passender Länge kopiert.

Die Klasse Pipe kann damit wie in Listing 3.8 angegeben realisiert werden:

**Listing 3.8**

```
public class Pipe
{
    private byte[] buffer = null;
    private int bsize = 0;
    private int head = 0;
    private int tail = 0;

    public Pipe(int capacity)
    {
        // Abprüfen von capacity <= 0 ausgelassen ...
        buffer = new byte[capacity];
    }

    public int getCapacity()
    {
        return buffer.length;
    }

    public synchronized void send(byte[] msg)
    {
        if(msg.length <= buffer.length)
        {
            /* Senden "auf einen Schlag" */
            while(msg.length > buffer.length - bsize)
            {
                try
                {
                    wait();
                }
                catch(InterruptedException e)
                {
                }
            }
            /* Kopieren der Nachricht */
            for(int i = 0; i < msg.length; i++)
            {
                buffer[tail] = msg[i];
                tail++;
                if(tail == buffer.length)
                {
                    tail = 0;
                }
            }
            bsize += msg.length;
            notifyAll();
        }
        else
        {
            /* Senden in Portionen */
            int offset = 0;
            int stillToSend = msg.length;
            while(stillToSend > 0)
            {
                while(bsize == buffer.length)
```

```
{  
    try  
    {  
        wait();  
    }  
    catch(InterruptedException e)  
    {  
    }  
}  
int sendNow = buffer.length - bsize;  
if(stillToSend < sendNow)  
{  
    sendNow = stillToSend;  
}  
for(int i = 0; i < sendNow; i++)  
{  
    buffer[tail] = msg[offset];  
    tail++;  
    if(tail == buffer.length)  
    {  
        tail = 0;  
    }  
    offset++;  
}  
bsize += sendNow;  
stillToSend -= sendNow;  
notifyAll();  
}  
}  
}  
  
public synchronized byte[] receive(int noBytes)  
// noBytes: number of bytes  
{  
    while(bsize == 0)  
    {  
        try  
        {  
            wait();  
        }  
        catch(InterruptedException e)  
        {  
        }  
    }  
    if(noBytes > bsize)  
    {  
        noBytes = bsize;  
    }  
    byte[] result = new byte[noBytes];  
    for(int i = 0; i < noBytes; i++)  
    {  
        result[i] = buffer[head];  
        head++;  
        if(head == buffer.length)  
        {  
            head = 0;  
        }  
    }  
}
```

```

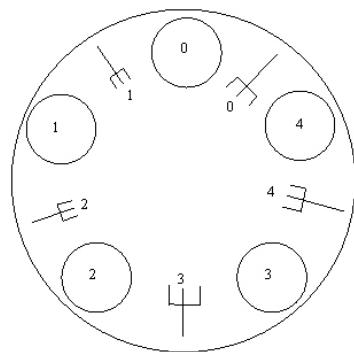
        bsize -= noBytes;
        notifyAll();
        return result;
    }
}

```

## ■ 3.4 Philosophen-Problem

Im Folgenden wenden wir uns nun bekannten Synchronisationsproblemen der Betriebssystemliteratur zu, nämlich dem *Philosophen-Problem* und dem Leser-Schreiber-Problem. Es werden verschiedene Lösungen für diese Probleme entwickelt. Einige Lösungen basieren auf den Java-Synchronisationsmechanismen, andere auf Semaphoren. Durch die Semaphor-Varianten soll gezeigt werden, wie diese Probleme auch in einem anderen Umfeld als dem Java-Umfeld gelöst werden können. Die Tatsache, dass alle Programme in diesem Buch in Java geschrieben sind und die Semaphore wiederum auf den Java-Synchronisationskonzepten basieren, sollte nicht den Blick verstellen, dass die Semaphore-Lösungen dennoch zeigen, wie diese Probleme unter Nutzung der Synchronisationskonzepte eines Betriebssystems wie z.B. Linux angegangen werden können. Dieses Ziel sollte man in Erinnerung behalten, wenn man die unterschiedlichen Lösungsvarianten betrachtet.

Zunächst betrachten wir das Philosophen-Problem. Es sitzen N Philosophen an einem runden Tisch. Vor jedem Philosoph steht ein Teller, der ausschließlich von diesem Philosophen benutzt wird. Die Philosophen und entsprechend deren Teller sind von 0 bis N-1 gegen den Uhrzeigersinn nummeriert. Zwischen je zwei Tellern befindet sich eine Gabel, die ebenfalls nummeriert ist, wobei die linke Gabel des Tellers i ebenfalls die Nummer i trägt. Diese Situation ist in Bild 3.5 für N=5 dargestellt.



**Bild 3.5**  
Das Philosophen-Problem

Die Philosophen tun nichts anderes als abwechselnd essen und denken. Zum Essen brauchen sie die beiden Gabeln links und rechts ihres Tellers. Dies bedeutet, dass keine zwei benachbarten Philosophen gleichzeitig essen können. Das Problem besteht nun darin, das Verhalten der Philosophen als Threads zu programmieren. Falls Ihnen das Problem vorkommt, als hätte es keinen Bezug zu einer praktischen Anwendung, so stellen Sie sich

anstatt der Gabeln Objekte vor, die von einem Thread immer wieder gleichzeitig und ausschließlich benutzt werden müssen.

### 3.4.1 Lösung mit synchronized – wait – notifyAll

Im Folgenden ist eine Lösung des Philosophen-Problems dargestellt, die auf den Java-Synchronisationskonzepten synchronized – wait – notifyAll basiert. Die zentrale Klasse ist die Klasse Table, die einen realen Tisch so abstrahiert, dass nur noch repräsentiert wird, welche Gabeln momentan benutzt werden und welche nicht. Zu diesem Zweck dient ein boolesches Feld forkUsed; das Feldelement i hat den Wert true, wenn die Gabel i benutzt wird, sonst false. Die Methode takeFork wird vom Philosoph i mit dem Argument i vor dem Essen aufgerufen. Dabei wird der Thread blockiert, falls die linke oder die rechte Gabel von einem anderen Philosophen benutzt wird. Werden dann beide Gabeln nicht mehr verwendet, werden sie von dem hungrigen Philiosopen in takeFork als benutzt gekennzeichnet. Die Methode putFork wird vom Philosophen i mit dem Argument i nach dem Essen aufgerufen. Diese Methode kennzeichnet die linke und die rechte Gabel wieder als unbenutzt und ruft notifyAll auf. Damit kann eventuell der linke oder rechte oder beide Nachbarn essen. Aus diesem Grund ist notifyAll statt notify nötig. Ein weiterer Grund für notifyAll ist, dass die Wartebedingung auch wieder parametrisiert ist, so dass es mehrere Wartebedingungen gibt. Die privaten Methoden links und rechts liefern die Nummer der linken bzw. rechten Gabel zu einer als Argument angegebenen Philosophen-Nummer. Damit dürfte das folgende Programm (Listing 3.9) ohne weitere Erläuterungen verständlich sein.

**Listing 3.9**

```
class Table
{
    private boolean[] forkUsed;

    private int left(int i)
    {
        return i;
    }

    private int right(int i)
    {
        if(i + 1 < forkUsed.length)
        {
            return i + 1;
        }
        else
        {
            return 0;
        }
    }

    public Table(int number)
    {
        forkUsed = new boolean[number];
        for(int i = 0; i < forkUsed.length; i++)
        {

```

```
        forkUsed[i] = false;
    }
}

public synchronized void takeFork(int number)
{
    while(forkUsed[left(number)] || forkUsed[right(number)])
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    forkUsed[left(number)] = true;
    forkUsed[right(number)] = true;
}

public synchronized void putFork(int number)
{
    forkUsed[left(number)] = false;
    forkUsed[right(number)] = false;
    notifyAll();
}
}

class Philosopher extends Thread
{
    private Table table;
    private int number;

    public Philosopher(Table table, int number)
    {
        this.table = table;
        this.number = number;
        start();
    }

    public void run()
    {
        while(true)
        {
            think(number);
            table.takeFork(number);
            eat(number);
            table.putFork(number);
        }
    }

    private void think(int number)
    {
        System.out.println("Philosoph " + number + " denkt.");
        try
        {
            sleep((int) (Math.random() * 20000));
        }
    }
}
```

```
        catch(InterruptedException e)
        {
        }
    }

    private void eat(int number)
    {
        System.out.println("Philosoph " + number
                           + " fängt zu essen an.");
        try
        {
            sleep((int) (Math.random() * 20000));
        }
        catch(InterruptedException e)
        {
        }
        System.out.println("Philosoph " + number
                           + " hört auf zu essen.");
    }
}

public class Philosophers
{
    private static final int NUMBER = 5;

    public static void main(String[] args)
    {
        Table table = new Table(NUMBER);
        for(int i = 0; i < NUMBER; i++)
        {
            new Philosopher(table, i);
        }
    }
}
```

### 3.4.2 Naive Lösung mit einfachen Semaphoren

Wie schon oben erwähnt wurde, sollen nicht nur Lösungen, die auf den Java-Synchronisationskonzepten beruhen, entwickelt werden, sondern auch Lösungen mit Semaphoren. Eine erste naheliegende Lösung des Philosophen-Problems mit Semaphoren besteht darin, jede Gabel durch einen Semaphor zu repräsentieren und alle diese Semaphore mit 1 zu initialisieren. Vor dem Essen muss ein Philosoph die P-Methode auf die Semaphore, die der linken und rechten Gabel entsprechen, anwenden. Entsprechend muss nach dem Essen v auf diese Semaphore ausgeführt werden. Das Programm in Listing 3.10 zeigt diese Lösung. Die Methoden eat und think entsprechen denen aus Listing 3.9 und werden deshalb nicht mehr wiederholt. Zur Abwechslung werden die Klassen für die Philosophen in den folgenden Beispielen nicht aus Thread abgeleitet, sondern es wird die Schnittstelle Runnable verwendet.

**Listing 3.10**

```
public class PhilosopherWithSemaphoresNaive implements Runnable
{
    private Semaphore[] sems;
    private int number;
    private int left, right;

    public PhilosopherWithSemaphoresNaive(Semaphore[] sems,
                                           int number)
    {
        this.sems = sems;
        this.number = number;
        left = number;
        right = number + 1;
        if(right == sems.length)
        {
            right = 0;
        }
    }

    public void run()
    {
        while(true)
        {
            think(number);
            sems[left].p();
            sems[right].p();
            eat(number);
            sems[left].v();
            sems[right].v();
        }
    }

    private void think(int number)
    {
        ...
    }

    private void eat(int number)
    {
        ...
    }

    private static final int NUMBER = 5;

    public static void main(String[] args)
    {
        Semaphore[] sems = new Semaphore[NUMBER];
        for(int i = 0; i < NUMBER; i++)
        {
            sems[i] = new Semaphore(1);
        }
        for(int i = 0; i < NUMBER; i++)
        {
            new Thread(new PhilosopherWithSemaphoresNaive(sems, i)).
                start();
        }
    }
}
```

Wie der Klassenname schon andeutet, ist diese Lösung nicht ganz durchdacht, denn es kann bei dieser Lösung zu einer *Verklemmung* kommen. Wenn nämlich – anschaulich gesprochen – alle Philosophen gleichzeitig ihre linke Gabel nehmen, dann kann kein Philosoph seine rechte Gabel nehmen. Da aber damit kein Philosoph mit dem Essen beginnen und später mit dem Essen wieder aufhören kann, wird die jeweilige linke Gabel nie wieder zurückgelegt. Der Zustand ändert sich damit nicht mehr; es ist zu einer Verklemmung gekommen (s. Abschnitte 3.11 und 3.12).

Auf das Programm übertragen kann diese Situation eintreten, falls nach der Anweisung „sems[left].p();“ vom ersten Thread auf den zweiten Thread umgeschaltet wird, dieser zweite Thread ebenfalls diese Anweisung ausführt und dann auf den dritten Thread umgeschaltet wird usw. Auch wenn ein solcher Ablauf eher unwahrscheinlich ist, darf dies für uns kein Grund zur Beruhigung sein, wie dies an anderer Stelle schon erläutert wurde.

### 3.4.3 Einschränkende Lösung mit gegenseitigem Ausschluss

Eine Verklemmungssituation kann nicht vorkommen, wenn man statt mehreren Semaphoren nur einen einzigen Semaphor verwendet, der eine Art von Essenserlaubnis repräsentiert. Das folgende Programm (Listing 3.11) setzt diese Idee um.

**Listing 3.11**

```
public class PhilosopherWithMutexSemaphore implements Runnable
{
    private Semaphore mutex;
    private int number;

    public PhilosopherWithMutexSemaphore(Semaphore mutex,
                                         int number)
    {
        this.mutex = mutex;
        this.number = number;
    }

    public void run()
    {
        while(true)
        {
            think(number);
            mutex.p();
            eat(number);
            mutex.v();
        }
    }

    private void think(int number)
    {
        ...
    }

    private void eat(int number)
    {
        ...
    }
}
```

```

    }

    private static final int NUMBER = 5;

    public static void main(String[] args)
    {
        Semaphore mutex = new Semaphore(1);
        for(int i = 0; i < NUMBER; i++)
        {
            new Thread(new PhilosopherWithMutexSemaphore(mutex, i)).
                start();
        }
    }
}

```

Diese Lösung ist zwar korrekt und verklemmungsfrei. Allerdings ist es bei dieser Lösung nicht möglich, dass mehr als ein Philosoph zu einem Zeitpunkt isst. Dies ist eine unnötig starke Beschränkung der möglichen Parallelität, denn bei N Philosophen können  $(N-1)/2$  Philosophen für ungerades N bzw.  $N/2$  Philosophen für gerades N gleichzeitig essen.

### 3.4.4 Gute Lösung mit einfachen Semaphoren

Eine korrekte und verklemmungsfreie Lösung, die die mögliche Parallelität nicht unnötig einschränkt, kann man erhalten, wenn man mit den Semaphoren keine Gabeln oder eine globale Essenserlaubnis repräsentiert, sondern wenn pro Essenserlaubnis eines Philosophen ein Semaphor verwendet wird. Vor dem Essen führt der Philosoph entsprechend die P-Methode auf seinem Essenserlaubnis-Semaphor aus. Die dazu passende V-Methode wird vom Philosophen selber zuvor ausgeführt, wenn er nämlich festgestellt hat, dass er selber essen kann. Andernfalls führt er die V-Methode nicht aus. Dies kann aber nur daran liegen, dass mindestens einer seiner Nachbarn momentan isst. Entsprechend muss nach dem Essen einer der Nachbarn die V-Methode für den betrachteten Philosophen auslösen. Das Aufrufen der V-Methode auf dem Essenserlaubnis-Semaphor des Nachbarn darf aber nicht in jedem Fall erfolgen, sondern nur dann, wenn der Nachbar auch essen möchte und dies jetzt tatsächlich möglich ist. Deshalb müssen auch Zustände über die Philosophen vermerkt werden. Da auf diese Zustände alle Philosophen-Threads lesend und schreibend zugreifen, muss der Zugriff darauf wiederum synchronisiert werden. Es sei nochmals daran erinnert, dass wir zu diesem Zweck nicht synchronized verwenden, da wir uns selber die Aufgabe gestellt haben, das Philosophen-Problem mit Hilfe von Semaphoren zu lösen, aber ohne direkte Nutzung der Java-Synchronisationsmechanismen synchronized, wait, notify und notifyAll. Aus diesem Grund wird der gegenseitige Ausschluss ebenfalls über einen Semaphor realisiert (vgl. Abschnitt 3.1.2).

Im folgenden Programm (Listing 3.12) ist die soeben beschriebene Idee in Programmcode umgesetzt worden.

#### Listing 3.12

```

class State
{
    public static final int THINKING = 0;
    public static final int HUNGRY = 1;
}

```

```
public static final int EATING = 2;

private int[] state;

public State(int number)
{
    state = new int[number];
    for(int i = 0; i < number; i++)
    {
        state[i] = THINKING;
    }
}
public int leftNeighbor(int i)
{
    if(i - 1 >= 0)
    {
        return i - 1;
    }
    else
    {
        return state.length - 1;
    }
}
public int rightNeighbor(int i)
{
    if(i + 1 < state.length)
    {
        return i + 1;
    }
    else
    {
        return 0;
    }
}
public void set(int index, int value)
{
    state[index] = value;
}
public boolean isEatingPossible(int i)
{
    if(state[i] == HUNGRY &&
       state[leftNeighbor(i)] != EATING &&
       state[rightNeighbor(i)] != EATING)
    {
        return true;
    }
    return false;
}
}

public class PhilosopherWithSemaphoresGood implements Runnable
{
    private Semaphore mutex;
    private Semaphore[] eatingAllowed;
    private State state;
    private int number, leftNeighbor, rightNeighbor;

    public PhilosopherWithSemaphoresGood(Semaphore mutex,
```

```
Semaphore[] eatingAllowed,
State state, int number)
{
    this.mutex = mutex;
    this.eatingAllowed = eatingAllowed;
    this.state = state;
    this.number = number;
    this.leftNeighbor = state.leftNeighbor(number);
    this.rightNeighbor = state.rightNeighbor(number);
}

public void run()
{
    while(true)
    {
        think(number);
        /* zum Essen anmelden; prüfen, ob E. möglich */
        mutex.p();
        state.set(number, State.HUNGRY);
        if(state.isEatingPossible(number))
        {
            state.set(number, State.EATING);
            eatingAllowed[number].v();
        }
        mutex.v();
        /* auf Essenserlaubnis warten und essen */
        eatingAllowed[number].p();
        eat(number);
        /* vom Essen abmelden; für Nachbarn prüfen */
        mutex.p();
        state.set(number, State.THINKING);
        if(state.isEatingPossible(leftNeighbor))
        {
            state.set(leftNeighbor, State.EATING);
            eatingAllowed[leftNeighbor].v();
        }
        if(state.isEatingPossible(rightNeighbor))
        {
            state.set(rightNeighbor, State.EATING);
            eatingAllowed[rightNeighbor].v();
        }
        mutex.v();
    }
}
private void think(int number)
{
    ...
}
private void eat(int number)
{
    ...
}

private static final int NUMBER = 5;

public static void main(String[] args)
{
    Semaphore mutex = new Semaphore(1);
```

Die Klasse State speichert für alle Philosophen einen der Zustände THINKING, HUNGRY oder EATING. Sie besitzt Methoden zum Abfragen der linken bzw. rechten Nummer eines Philosophen zu einer gegebenen Philosophen-Nummer, eine Methode zum Setzen des Zustands eines Philosophen sowie eine Methode zur Abfrage, ob der Philosoph mit der Nummer i essen kann. Dies ist genau dann der Fall, wenn er hungrig ist und weder sein rechter noch sein linker Philosoph gerade essen. Die Klasse PhilosopherWithSemaphores-Good repräsentiert einen Philosophen. Im Konstruktor dieser Klasse werden eine Referenz auf einen Semaphor für den gegenseitigen Ausschluss zum Zugriff auf das Zustands-Objekt, ein Feld mit Referenzen auf die Essenserlaubnis-Semaphore, eine Referenz auf das Zustands-Objekt und wie üblich die Nummer des Philosophen übergeben. In der Run-Methode wird vor dem Essen der eigene Zustand auf HUNGRY gesetzt. Dann wird geprüft, ob der Philosoph selber essen kann. Falls ja, so setzt er seinen eigenen Zustand auf EATING und erteilt sich durch Aufruf der Methode v selbst die Essenserlaubnis. Diese Abfragen und Änderungen der Philosophenzustände finden unter gegenseitigem Ausschluss statt. Anschließend prüft der Philosoph seine Essenserlaubnis. Dabei wird er unter Umständen blockiert. Nach dem Essen setzt der Philosoph seinen eigenen Zustand auf THINKING und prüft, ob der rechte und der linke Nachbar jetzt essen können und wollen. Gegebenenfalls wird deren Zustand auf EATING gesetzt und es wird ihnen die Essenserlaubnis erteilt. Auch der komplette Zugriff auf die Philosophenzustände erfolgt wieder unter gegenseitigem Ausschluss.

### 3.4.5 Lösung mit Semaphorgruppen

Eine einfacher zu programmierende Lösung mit Semaphoren erhält man, wenn man statt einzelner Semaphore die in Abschnitt 3.1.5 eingeführten Semaphorgruppen verwendet. Bei einer solchen Lösung repräsentieren die Semaphore jetzt wieder die Gabeln. Wenn sich alle Semaphore in einer einzigen Gruppe befinden, dann bedeutet dies, dass jeder Philosoph seine linke und rechte Gabel auf einmal ergreift und nicht nacheinander. Damit erhält man eine korrekte Lösung, bei der keine Verklemmungen vorkommen können. Man kann an diesem Beispiel deutlich den Vorteil einer Lösung mit einer Semaphorgruppe gegenüber der verklemmungsgefährdeten Lösung mit Einzelsemaphoren erkennen; die Situation, dass

alle Philosophen quasi gleichzeitig ihre linke Gabel ergreifen, ist hier nicht möglich, da die Änderungen an den Semaphoren „auf einen Schlag“ durchgeführt werden. Das folgende Programm (Listing 3.13) zeigt eine Lösung des Philosophen-Problems mit einer Semaphoregruppe.

### Listing 3.13

```
public class PhilosopherWithSemaphoreGroup extends Thread
{
    private SemaphoreGroup sems;
    private int leftFork;
    private int rightFork;

    public PhilosopherWithSemaphoreGroup(SemaphoreGroup sems,
                                         int number)
    {
        this.sems = sems;
        leftFork = number;
        if(number + 1 < sems.getNumberOfMembers())
        {
            rightFork = number + 1;
        }
        else
        {
            rightFork = 0;
        }
        start();
    }

    public void run()
    {
        int[] deltas = new int[sems.getNumberOfMembers()];
        for(int i = 0; i < deltas.length; i++)
        {
            deltas[i] = 0;
        }
        int number = leftFork;
        while(true)
        {
            think(number);
            deltas[leftFork] = -1;
            deltas[rightFork] = -1;
            sems.changeValues(deltas);
            eat(number);
            deltas[leftFork] = 1;
            deltas[rightFork] = 1;
            sems.changeValues(deltas);
        }
    }

    private void think(int number)
    {
        ...
    }

    private void eat(int number)
    {
```

```

    ...
}

private static final int NUMBER = 5;

public static void main(String[] args)
{
    SemaphoreGroup forks = new SemaphoreGroup(NUMBER);
    int[] init = new int[NUMBER];
    for(int i = 0; i < NUMBER; i++)
    {
        init[i] = 1;
    }
    forks.changeValues(init);
    for(int i = 0; i < NUMBER; i++)
    {
        new PhilosopherWithSemaphoreGroup(forks, i);
    }
}
}

```

Auch diese Lösung ist wieder korrekt und verklemmungsfrei. Auch wird die mögliche Parallelität nicht eingeschränkt. Dies ist die bislang einfachste derartige Lösung mit Semaphoren. Im Abschnitt 3.12 wird im Rahmen der Betrachtung über Verklemmungen noch eine weitere einfache Lösung mit Einzelsemaphoren vorgestellt, die auch korrekt, verklemmungsfrei und ohne Einschränkung möglicher Parallelität ist.

## ■ 3.5 Leser-Schreiber-Problem

Nach dem Philosophen-Problem betrachten wir eine weitere bekannte Synchronisationsaufgabe, nämlich das so genannte *Leser-Schreiber-Problem*. Man geht dabei von einer Situation aus, in der mehrere Threads lesend und schreibend auf gemeinsam benutzte Daten zugreifen. Wie schon in Kapitel 2 beschrieben wurde, müssen dabei alle lesenden und schreibenden Zugriffe synchronisiert werden. Wir haben dies bisher durch gegenseitigen Ausschluss mit Hilfe von `synchronized` realisiert. Der gegenseitige Ausschluss verhindert zwar alle unliebsamen Effekte, schränkt die mögliche Parallelität aber unnötig stark ein (wie z. B. bei der Klasse `PhilosopherWithSemaphoresMutex` in Listing 3.11), denn es ist völlig unkritisch, wenn mehrere Threads gleichzeitig lesen. Allerdings darf zu einem Zeitpunkt höchstens ein Thread die Daten verändern. Die unnötige Einschränkung der Parallelität für die lesenden Zugriffe spielt vor allem dann eine Rolle, falls diese Zugriffe auf die Daten länger andauern. Das Leser-Schreiber-Problem besteht nun darin, eine Lösung für den Zugriff auf die Daten zu finden, so dass zu einem Zeitpunkt mehrere Threads die Daten lesen, aber höchstens ein Thread die Daten schreiben kann.

In diesem Abschnitt werden manche Threads die Daten nur lesen und andere Threads die Daten nur schreiben. Wir sprechen deshalb auch von Lesern und Schreibern. Mögliche Lösungen unterscheiden sich darin, ob jemand bevorzugt wird (die Leser oder die Schreiber), oder ob eine faire Bedienstrategie nach dem Motto „Wer zuerst kommt, mahlt zuerst“ realisiert wird:

- Die einfachste Lösung besteht darin, mehrere Leser gleichzeitig zugreifen zu lassen, Schreibern aber exklusiven Zugriff zu gewähren. Wer wann zum Zug kommt, hängt nur von der Vergabestrategie des Betriebssystems ab. Im Programmcode werden keine besonderen Vorkehrungen dafür getroffen.
- Eine Bevorzugung der Schreiber ist z. B. dann günstig, wenn man erreichen möchte, dass die Leser immer möglichst aktuelle Daten sehen. Sobald ein Thread schreiben möchte, werden keine neuen Leser mehr zugelassen. Allerdings muss der Thread mit gewünschtem Schreibzugriff so lange warten, bis alle momentan laufenden Lesevorgänge beendet sind.
- Die Bevorzugung der Leser ist z. B. dann von Vorteil, wenn es vorwiegend darauf ankommt, den Lesern einen sehr schnellen Zugriff zu gewähren, wenn aber die Aktualität der Daten weniger wichtig ist. In diesem Fall kann ein Schreiber nur dann zugreifen, falls keine Leser mit den Daten arbeitet. Leser werden dagegen nur aufgehalten, falls ein Schreiber aktiv ist.
- Die beiden oben genannten Strategien können nur dann sinnvoll eingesetzt werden, falls kein Thread unerwünscht lang auf seinen Zugriff warten muss. Das heißt, dass bei der Bevorzugung der Leser garantiert sein muss, dass jeder Schreiber nach einer akzeptablen Wartezeit die Änderungen an den Daten durchführen kann. Entsprechendes gilt für die Leser bei einer Bevorzugung der Schreiber. Wenn dies aber nicht garantiert werden kann, dann bietet sich eine strenge Reihenfolgestrategie an. Jeder Zugriffswunsch wird in eine Warteliste eingereiht und immer der erste Wunsch wird erfüllt. Ist der erste Wunsch ein Lesewunsch, so können direkt nachfolgende Lesewünsche ebenfalls erfüllt werden.

### 3.5.1 Lösung mit synchronized – wait – notifyAll

Wie zuvor geben wir wieder zuerst eine Lösung mit direkter Nutzung der Java-Synchronisationsmechanismen und danach eine Lösung mit Semaphoren an. In der ersten Lösung werden die Schreiber bevorzugt, während in der zweiten Lösung den Lesern Vorrang eingeräumt wird. Die Entwicklung einer Lösung ohne besondere Bevorzugung oder einer Lösung mit strenger Reihenfolgestrategie überlassen wir den Leserinnen und Lesern.

Bei der ersten Lösung (Listing 3.14) wird die Zugriffskontrolle in der abstrakten Klasse AccessControl gekapselt. Diese Klasse besitzt die öffentlichen Methoden read und write. Jede dieser Methoden enthält einen Aufruf einer privaten Methode zum eventuellen Warten, bis der Zugriff möglich ist (beforeRead bzw. beforeWrite), einen Aufruf einer Methode zum eigentlichen Lesen und Schreiben (reallyRead bzw. reallyWrite) sowie einen Aufruf einer privaten Methode zum Benachrichtigen von Threads, die auf den Zugriff warten (afterRead bzw. afterWrite). Die Methoden reallyRead und reallyWrite sind abstrakt und protected. In ihnen muss in einer abgeleiteten Klasse der eigentliche Datenzugriff implementiert werden. Die privaten Methoden beforeRead, beforeWrite, afterRead und afterWrite realisieren die Zugriffskontrolle. Damit dies möglich ist, muss die Klasse Attribute besitzen, die die Anzahl der aktiven und wartenden Leser und Schreiber widerspiegeln. In beforeRead bzw. beforeWrite wird die Anzahl der wartenden Leser bzw. Schreiber zunächst erhöht. Anschließend wird geprüft, ob ein Zugriff möglich ist. Falls dies nicht möglich ist, wird gewartet. Nach dem Warten wird die Anzahl der wartenden Leser bzw. Schreiber wieder

heruntergezählt, die Zahl der aktiven Leser bzw. Schreiber dagegen um eins erhöht. In afterRead bzw. afterWrite wird die Anzahl der aktiven Leser bzw. Schreiber wieder gesenkt und alle wartenden Threads werden zu einer Überprüfung ihrer Wartebedingung geweckt.

Je nach Realisierung der privaten Methoden beforeRead, beforeWrite, afterRead und afterWrite können unterschiedliche Bedienstrategien realisiert werden. Hier wird als Beispiel eine Bevorzugung der Schreiber realisiert. Deshalb wird in der Methode beforeRead so lange gewartet, bis es weder wartende Schreiber noch aktive Schreiber gibt, während in der Methode beforeWrite nur gewartet wird, bis keine Leser und keine Schreiber mehr aktiv sind – wartende Leser werden aber nicht berücksichtigt.

### Listing 3.14

```
abstract class AccessControl
{
    private int activeReaders = 0;
    private int activeWriters = 0;
    private int waitingReaders = 0;
    private int waitingWriters = 0;

    protected abstract Object reallyRead();
    protected abstract void reallyWrite(Object obj);

    public Object read()
    {
        beforeRead();
        Object obj = reallyRead();
        afterRead();
        return obj;
    }

    public void write(Object obj)
    {
        beforeWrite();
        reallyWrite(obj);
        afterWrite();
    }

    private synchronized void beforeRead()
    {
        waitingReaders++;
        while(waitingWriters != 0 || activeWriters != 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        waitingReaders--;
        activeReaders++;
    }

    private synchronized void afterRead()
    {
```

```
        activeReaders--;
        notifyAll();
    }

    private synchronized void beforeWrite()
    {
        waitingWriters++;
        while(activeReaders != 0 || activeWriters != 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        waitingWriters--;
        activeWriters++;
    }

    private synchronized void afterWrite()
    {
        activeWriters--;
        notifyAll();
    }
}

class IntData extends AccessControl
{
    private int data;

    protected Object reallyRead()
    {
        return new Integer(data);
        //return data;
    }

    protected void reallyWrite(Object obj)
    {
        data = ((Integer) obj).intValue();
        //data = (Integer) obj;
    }
}

class Reader extends Thread
{
    private IntData data;

    public Reader(IntData data)
    {
        this.data = data;
        start();
    }

    public void run()
    {
        for(int i = 0; i < 10000; i++)
    }
```

```

    {
        Integer integer = (Integer) data.read();
        int iValue = integer.intValue();
        // mit iValue arbeiten, z.B.
        System.out.println("iValue = " + iValue);
    }
}

class Writer extends Thread
{
    private IntData data;

    public Writer(IntData data)
    {
        this.data = data;
        start();
    }

    public void run()
    {
        for(int i = 0; i < 10000; i++)
        {
            data.write(new Integer(i));
        }
    }
}

public class ReaderWriter
{
    public static void main(String[] args)
    {
        IntData data = new IntData();
        for(int i = 0; i < 100; i++)
        {
            new Writer(data);
            new Reader(data);
        }
    }
}

```

Die Methoden `read` und `write` der Klasse `AccessControl` sind möglichst allgemein gehalten. Beim Schreiben wird eine Referenz auf ein Objekt der Klasse `Object`, der Basisklasse aller Klassen, als Parameter übergeben, während beim Lesen eine Referenz auf ein solches Objekt als Rückgabewert zurückgeliefert wird. Als einfaches Beispiel für eine konkrete Klasse, die aus `AccessControl` abgeleitet wird, dient die Klasse `IntData`, die den lesenden und schreibenden Zugriff auf eine Int-Variable realisiert. Da `int` ein primitiver Datentyp ist und nicht aus der Klasse `Object` abgeleitet ist, wird zur Parameterübergabe bzw. zur Rückgabe des gelesenen Werts die Klasse `Integer` verwendet. Aufgrund der so genannten Auto-boxing-Autounboxing-Fähigkeit ist ab Java 5 diese Kapselung im eigenen Programmcode nicht mehr notwendig, da der Compiler die Wandlung zwischen primitiven Datentypen und den entsprechenden Wrapper-Typen automatisch vornimmt (siehe auskommentierte Zeilen in der Klasse `IntData`). `Reader` und `Writer` sind die Klassen für die Leser- bzw. Schreiber-Threads.

### 3.5.2 Lösung mit additiven Semaphoren

Im Gegensatz zu der eben besprochenen Lösung des Leser-Schreiber-Problems, in der relativ einfach andere Bedienstrategien durch entsprechende Änderung der Methoden beforeRead, beforeWrite, afterRead und afterWrite realisiert werden können, ergibt sich bei einer einfachen Lösung des Leser-Schreiber-Problems mit additiven Semaphoren automatisch eine Lösung mit einer Bevorzugung der Leser. Andere Bedienstrategien lassen sich ebenfalls mit Semaphoren realisieren, verlangen aber einen anderen Ansatz. Bei der einfachen Lösung mit additiven Semaphoren sollte man zunächst eine Zahl MAX bestimmen, die größer oder gleich der maximalen Zahl von Lesern ist. Der Semaphor wird mit MAX initialisiert. Vor jedem Lesen wird der Semaphor um eins erniedrigt und nach dem Lesen um eins erhöht, während vor jedem Schreiben der Semaphor um MAX erniedrigt und nach dem Schreiben um MAX erhöht wird. Damit können alle Leser gleichzeitig zugreifen. Wenn aber ein Schreiber zugreift, kann kein anderer Leser und kein anderer Schreiber zugreifen. Ein Schreiber kann nur zugreifen, falls kein Leser aktiv ist. Solange ein Leser aktiv ist, können unbegrenzt lang weitere Leser ihren Datenzugriff beginnen und wartende Schreiber überholen. Die nachfolgend dargestellte Klasse AccessControlSem (Listing 3.15) zeigt eine Lösung des Leser-Schreiber-Problems mit additiven Semaphoren. Ein vollständiges Beispielprogramm ergibt sich durch Änderung von „extends AccessControl“ durch „extends AccessControlSem“ in der Klasse IntData des vorigen Programms in Listing 3.14.

**Listing 3.15**

```
abstract class AccessControlSem
{
    private static final int MAX = 1000;

    private AdditiveSemaphore sem = new AdditiveSemaphore(MAX);

    protected abstract Object reallyRead();
    protected abstract void reallyWrite(Object obj);

    public Object read()
    {
        beforeRead();
        Object obj = reallyRead();
        afterRead();
        return obj;
    }

    public void write(Object obj)
    {
        beforeWrite();
        reallyWrite(obj);
        afterWrite();
    }

    private void beforeRead()
    {
        sem.p(1);
    }

    private void afterRead()
```

```
{  
    sem.v(1);  
}  
  
private void beforeWrite()  
{  
    sem.p(MAX);  
}  
  
private void afterWrite()  
{  
    sem.v(MAX);  
}  
}
```

Sollte die Konstante MAX zu klein gewählt worden sein (d.h. es gibt mehr Leser-Threads als die Konstante MAX), so ist die vorgestellte Lösung nicht falsch. Es können dann einfach lediglich nur MAX Leser gleichzeitig lesen statt alle.

## ■ 3.6 Schablonen zur Nutzung der Synchronisationsprimitive und Konsistenzbetrachtungen

Nachdem wir nun eine ganze Reihe von Beispielen für die Nutzung der Java-Synchronisationsprimitive synchronized, wait und notify bzw. notifyAll betrachtet haben, wird die typische Nutzung dieser Konzepte noch einmal zusammenfassend dargestellt.

Der Zustand eines Objekts wird durch die Werte seiner Attribute beschrieben. In der Regel gibt es gewisse *Konsistenzbedingungen*, die für die Attribute „immer“ gelten (auch *Invarianten* oder *Integritätsbedingungen* genannt). Methoden, die den Zustand eines Objekts ändern, überführen das Objekt von einem konsistenten in einen anderen konsistenten Zustand, wobei dies in der Regel in mehreren Schritten erfolgt und zwischenzeitlich die *Konsistenz* verletzt sein kann (deshalb wurde „immer“ oben in Anführungszeichen geschrieben). Dies ist übrigens auch einer der Gründe dafür, dass Attribute einer Klasse am besten privat sind und deren Werte somit nur durch den Aufruf von Methoden geändert werden können. Sind sie nämlich öffentlich, so kann nicht verhindert werden, dass die Werte der Attribute beliebig verändert und damit die Konsistenzbedingungen verletzt werden. Greifen mehrere Threads auf dasselbe Objekt zu und mindestens einer der Threads auch schreibend, so müssen alle lesenden und schreibenden Zugriffe auf die Attribute in synchronisierter Weise erfolgen (zum Beispiel mit Synchronized-Methoden, mit Locks, mit Semaphoren oder mit Techniken, wie sie im Abschnitt über das Leser-Schreiber-Problem vorgestellt wurden).

In manchen Fällen kommt es vor, dass die Zustandsänderung an einem Objekt nur durchgeführt werden kann, wenn der Zustand eine bestimmte Bedingung erfüllt. Beispiele hierfür sind etwa das Herunterzählen eines Zählers nur dann, falls er nicht negativ wird oder das Entnehmen eines Werts aus einem Puffer, falls der Puffer mindestens einen Wert ent-

hält. In diesem Fall wird vor der Veränderung des Zustands eine Wartebedingung überprüft. Ist diese Bedingung erfüllt, so wird die Methode wait aufgerufen. Wie schon erläutert wurde, muss diese Überprüfung auch nach dem Warten nochmals überprüft werden, so dass hier immer eine While-Schleife zu verwenden ist. Andere Veränderungen des Objektzustands können jederzeit durchgeführt werden wie z.B. das Hochzählen des Zählers bei einem Semaphor. Falls Zustandsänderungen dazu führen können, dass wartende Threads weiterlaufen können, muss nach solchen Zustandsänderungen notify oder notifyAll aufgerufen werden. Dass dies nicht für jede Zustandsänderung der Fall sein muss, ist am Beispiel des Semaphors zu sehen: Threads warten lediglich darauf, dass der Wert des Semaphors größer wird als ein bestimmter Wert. Wenn der Zustand des Semaphors verändert wird, indem der Wert heruntergezählt wird, so kann diese Zustandsänderung nicht dazu führen, dass ein wartender Thread weiterlaufen kann. Entsprechend wird beim Herunterzählen kein wartender Thread geweckt. Nach rein lesenden Zugriffen muss grundsätzlich kein wartender Thread geweckt werden. Allerdings kann das Lesen in manchen Fällen ebenfalls an Wartebedingungen geknüpft werden.

Das heißt also:

- Rein lesende Methoden können mit oder ohne wait ausgestattet sein. Ein Aufruf von notify bzw. notifyAll ist in keinem Fall notwendig, da der Zustand des Objekts nicht verändert wird. Also gibt es nur zwei Schablonen für lesende Methoden.
- Schreibende Methoden gibt es mit und ohne wait, aber auch mit und ohne notify bzw. notifyAll. Dementsprechend gibt es vier Schablonen für schreibende Methoden.

Das folgende Programmgerüst (Listing 3.16) fasst diese Überlegungen noch einmal zusammen:

### **Listing 3.16**

```
public class Template
{
    /* Attribute für den Zustand eines Objekts */
    private ...

    /* Konstruktor: Initialisierung der Attribute,
       dabei Herstellen eines konsistenten Zustands
    */
    public Template(...)
    {
        ...
    }

    /* Methode, bei der der Objektzustand keine Bedingung
       erfüllen muss, bevor er gelesen werden kann
    */
    public synchronized ... m1(...)
    {
        // Attribute lesen:
        ...
        return ...;
    }

    /* Methode, bei der der Objektzustand eine bestimmte Bedingung
       erfüllen muss, bevor er gelesen werden kann
    */
}
```

```
/*
public synchronized ... m2(...)
{
    while(<Wartebedingung>)
        // Wartebedingung abh. vom Zustand des Objekts
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    // Attribute lesen:
    ...
    return ...;
}

/* Methode, bei der der Objektzustand keine Bedingung
erfüllen muss, bevor er geändert werden kann;
die Änderung ist so, dass keine wartenden Threads
danach weiterlaufen können
*/
public synchronized void m3(...)
{
    // Attribute ändern:
    ...
}

/* Methode, bei der der Objektzustand eine bestimmte Bedingung
erfüllen muss, bevor er geändert werden kann;
die Änderung ist aber so, dass keine wartenden Threads
danach weiterlaufen können
*/
public synchronized void m4(...)
{
    // evtl. Attribute schon vor dem Warten ändern:
    ...
    while(<Wartebedingung>)
        // Wartebedingung abhängig vom Zustand des Objekts
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    // Attribute ändern:
    ...
}

/* Methode, bei der der Objektzustand keine Bedingung
erfüllen muss, bevor er geändert werden kann;
die Änderung ist so, dass eventuell wartende Threads
```

```

        danach unter Umständen weiterlaufen können
*/
public synchronized void m5(...)
{
    // Attribute ändern:
    ...
    notifyAll();
    /* oder notify,
       falls höchstens ein Thread weiterlaufen kann
    */
}

/* Methode, bei der der Objektzustand eine bestimmte Bedingung
erfüllen muss, bevor er geändert werden kann;
die Änderung ist so, dass eventuell wartende Threads
danach unter Umständen weiterlaufen können
*/
public synchronized void m6(...)
{
    // evtl. Attribute schon vor dem Warten ändern:
    ...
    while(<Wartebedingung>)
        // Wartebedingung abhängig vom Zustand des Objekts
    {
        try
        {
            wait();
        }
        catch(InterruptedException e)
        {
        }
    }
    // Attribute ändern:
    ...
    notifyAll();
    /* oder notify,
       falls höchstens ein Thread weiterlaufen kann
    */
}
}

```

Falls die Wartebedingung nicht einfach formulierbar ist, sondern etwas komplizierter zu berechnen ist (wie zum Beispiel in der Methode changeValues der Klasse SemaphoreGroup in Listing 3.5), dann sollte zur Berechnung der Bedingung immer eine Methode definiert werden, die in der While-Bedingung aufgerufen wird.

Wie zu Beginn dieses Abschnitts beschrieben hilft bei der Programmierung die Vorstellung, dass zu Beginn eines Methodenaufrufs der Zustand des Objekts (d.h. das sind die Werte der Attribute), auf die die Methode angewendet wird, konsistent ist (d.h. bestimmte Invarianten bzw. Integritätsbedingungen, die anwendungsabhängig sind, gelten für die Attributwerte). Jede Methode der Klasse muss so programmiert sein, dass am Ende des Methodenaufrufs der Zustand des Objekts wieder konsistent ist. Dabei ist insbesondere zu berücksichtigen, dass während der Ausführung der Methoden auch Ausnahmen ausgelöst werden können. Eine Ausnahme darf nicht zu einem vorzeitigen Beenden einer Methode führen, falls der Zustand des Objekts inkonsistent sein könnte. Man sollte also in jedem Fall dafür sorgen,

dass vor dem Verlassen der Methode wieder ein konsistenter Zustand hergestellt wird (z.B. mit try und finally).

Eine weitere Problematik entsteht durch den Aufruf von wait. Da bei wait die Sperre auf das Objekt freigegeben wird, muss vor jedem Aufruf von wait der Objektzustand konsistent sein, denn sonst könnte ein anderer Thread eine Methode auf das Objekt anwenden, wobei die Konsistenz zu Beginn des Methodenaufrufs dann nicht gelten würde. Die Schablonenmethoden m4 und m6 oben weisen darauf hin, dass eventuell schon vor dem Aufruf von wait Attribute des Objekts geändert werden können. Wichtig ist dann aber, dass durch diese Änderungen keine Inkonsistenz entsteht.

Diese Aussagen können in der folgenden Regel zusammengefasst werden:



Bei jedem Setzen der Sperre auf ein Objekt ist der Zustand dieses Objekts konsistent, falls dies nach der Initialisierung und bei jeder Freigabe der Sperre der Fall ist. Das Setzen der Sperre erfolgt beim Betreten eines Synchronized-Blocks sowie kurz vor der Rückkehr aus der Wait-Methode. Das Freigeben der Sperre erfolgt, wenn die Methode beendet (regulär oder durch eine nicht gefangene Ausnahme) oder die Wait-Methode aufgerufen wird. Es ist also darauf zu achten, dass nach der Initialisierung und bei jeder Freigabe einer Sperre der Objektzustand konsistent ist, denn dann kann man davon ausgehen, dass der Zustand auch bei jedem Setzen einer Sperre konsistent ist. Was unter Konsistenz zu verstehen ist, ist anwendungsabhängig.

Wir betrachten zur Illustration einen einfachen Semaphor, der zusätzliche Attribute für den Anfangswert des Semaphors sowie zum Zählen der Anzahl der durchgeföhrten P- und V-Operationen besitzt. Eine sinnvolle Invariante ist:

$$\text{Anfangswert} + \text{Anzahl der V-Operationen} - \text{Anzahl der P-Operationen} = \text{aktueller Wert}$$

Ein Problem entsteht nun, wenn in der Implementierung der Methode p der P-Operationszähler gleich zu Beginn erhöht wird. Damit ist die Invariante verletzt, was zunächst kein Problem darstellt. Wenn dann aber anschließend gewartet werden muss, dann wird die Sperre auf das Objekt freigegeben und der Zustand des Objekts ist bei Aufruf einer Methode durch einen anderen Thread nicht mehr konsistent. Im konkreten Fall ist die Lösung natürlich einfach: Der Zähler für die durchgeföhrten P-Operationen darf in der Methode p erst nach der While-Wait-Schleife erhöht werden, nämlich dann, wenn der Semaphorwert erniedrigt wird.

## ■ 3.7 Concurrent-Klassenbibliothek aus Java 5

Vor Java 5 gab es zur Realisierung von Parallelität im Wesentlichen nur die Klasse Thread und zur Realisierung der Synchronisation im Wesentlichen nur synchronized, wait und notify bzw. notifyAll. Weitere nützliche Klassen wie Semaphore und Message Queues musste man sich selbst schreiben, wie dies in den ersten Abschnitten dieses Kapitels ge-

zeigt wird. Ab Java 5 gibt es in der Klassenbibliothek drei weitere Packages namens `java.util.concurrent`, `java.util.concurrent.atomic` und `java.util.concurrent.locks`, in denen sich eine Reihe von Klassen und Schnittstellen befinden, die mit Parallelität und Synchronisation zu tun haben.

Wir geben in diesem Abschnitt einen Überblick über den Inhalt dieser drei Packages. Einige Aspekte werden dabei etwas ausführlicher behandelt als andere.

### 3.7.1 Executors

Statt Threads selbst zu erzeugen und gegebenenfalls auf deren Ende zu warten, bietet Java 5 mehrere Executor-Schnittstellen und implementierende Klassen an, deren Methoden Aufträge zur Ausführung entgegennehmen. Ob dabei für jeden Auftrag ein eigener Thread erzeugt wird oder ob es einen Pool mit einer beschränkten Anzahl von Threads gibt, ist dabei eine Angelegenheit der implementierenden Klasse. Dieser Gesichtspunkt spielt u. a. bei der Programmierung von Servern eine wichtige Rolle. Wir werden deshalb in Kapitel 5 darauf zurückkommen.

Grundlage aller Executors ist die Schnittstelle *Executor*, welche eine Methode namens *execute* besitzt, um einen Auftrag zur Ausführung eines Runnable-Objekts zu erteilen (die Schnittstelle *Runnable* besitzt eine Void-Methode namens *run* ohne Argumente, s. Abschnitt 2.1):

```
public interface Executor
{
    public void execute(Runnable r);
}
```

Nun hängt der Grad der Parallelität von der Implementierung der Klasse ab, welche die Executor-Schnittstelle implementiert. Eine mögliche Implementierung könnte so sein, dass der Thread, der die Execute-Methode aufruft, den Auftrag selbst ausführt (s. Listing 3.17):

#### **Listing 3.17**

```
public class SelfExecutor implements Executor
{
    public void execute(Runnable r)
    {
        r.run();
    }
}
```

Eine andere Variante, die in vielen Servern verwendet wird, ist die Erzeugung eines neuen Threads für jeden neuen Auftrag (s. Listing 3.18).

#### **Listing 3.18**

```
public class ThreadPerTaskExecutor implements Executor
{
    public void execute(Runnable r)
    {
        Thread t = new Thread(r);
    }
}
```

```
        t.start();
    }
}
```

Wenn für jeden Auftrag ein neuer Thread erzeugt wird, kann damit aber ein Server z. B. bei einem so genannten Denial-Of-Service-Angriff, bei dem ein Server gezielt überlastet werden soll, zusammenbrechen. Aus diesem Grund wird oft ein *Thread-Pool* eingesetzt. Die Vorteile eines Thread-Pools sind, dass nicht für jeden Auftrag ein neuer Thread erzeugt werden muss, und dass die Anzahl der Threads nach oben beschränkt ist (die Anzahl der Threads ist dabei nicht zwingend konstant, sondern die Zahl kann in gewissen Grenzen schwanken). Die vorhandenen Threads teilen sich die Bearbeitung der Aufträge. Das heißt, dass ein Thread im Laufe seines Lebens mehrere Aufträge hintereinander bearbeitet. Eine einfache Implementierung könnten wir selber schreiben, indem wir eine ähnliche Klasse wie Message-Queue (s. Abschnitt 3.2) benutzen (nur mit Runnable-Objekten statt mit Byte-Feldern). Die Methode execute würden wir dann einfach so realisieren, dass diese mit Hilfe der send-Methode das Runnable-Objekt in die MessageQueue stellt. Die Bearbeiter-Threads des Pools würden dann alle so realisiert, dass sie in einer Endlosschleife zuerst mit receive warten, bis sie einen Auftrag aus der MessageQueue entnehmen können, um anschließend die run-Methode auf das entnommene Runnable-Objekt anzuwenden. Es ist somit nicht nötig, darüber Buch zu führen, welche der Bearbeiter-Threads gerade beschäftigt und welche frei sind, um einen neuen Auftrag gezielt einem freien Thread zuzuteilen.

In der Regel will man aber nicht nur Aufträge erteilen und davon zukünftig nichts mehr wissen. Viel typischer ist, dass man an dem Ergebnis, das durch die Ausführung des Auftrags berechnet wird, interessiert ist. Deshalb gibt es eine zweite Kategorie von Aufträgen, die nicht die Void-Methode run implementieren, sondern die Methode *call*, welche ein Objekt als Ergebnis zurückliefert. Man hätte als Rückgabetyp Object verwenden können. Eleganter ist es aber, hier die ebenfalls seit Java 5 vorhandenen Generics einzusetzen und die Schnittstelle *Callable* über den Rückgabetyp zu parametrisieren:

```
public interface Callable<V>
{
    public V call() throws Exception;
}
```

Beim Erteilen eines solchen Callable-Auftrags bekommt man ein Future-Objekt zurückgeliefert. Mit Hilfe dieses Future-Objekts kann man auf das vom Auftrag errechnete Ergebnis warten (mit oder ohne Angabe einer Befristung) oder den Auftrag abbrechen. Future ist ebenfalls eine Schnittstelle, die wie Callable über den Rückgabetyp des Ergebnisses parametrisiert ist:

```
public interface Future<V>
{
    public boolean cancel(boolean mayInterruptIfRunning);
    public V get() throws InterruptedException, ExecutionException;
    public V get(long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
               TimeoutException;
    public boolean isCancelled();
    public boolean isDone();
}
```

Mit dem Argument des Typs boolean der Methode *cancel* kann man angeben, dass der Auftrag nur abgebrochen werden soll, falls mit seiner Bearbeitung noch nicht begonnen wurde (Argumentwert false), oder dass der Abbruch in jedem Fall, auch wenn die Bearbeitung schon begonnen hat, erfolgen soll (Argumentwert true). Mit den zwei Varianten der Get-Methode kann das Ergebnis des Auftrags abgeholt werden, wobei auf das Ergebnis (unbefristet oder befristet) gewartet wird, falls es noch nicht vorliegt. Die Angabe der Frist erfolgt durch einen Zahlenwert und eine Zeiteinheit vom Typ *TimeUnit*. *TimeUnit* ist eine so genannte Enum (Aufzählung), mit der die Zeiteinheit spezifiziert wird (Nanosekunden, Mikrosekunden, Millisekunden, Sekunden usw.). Mit *isCancelled* und *isDone* kann abgefragt werden, ob der Auftrag abgebrochen wurde bzw. seine Bearbeitung beendet ist.

Die Erweiterung der Schnittstelle *Executor* namens *ExecutorService* berücksichtigt das Erteilen von Callable-Aufträgen. Bei der Auftragserteilung wird ein Future-Objekt zurückgeliefert:

```
public interface ExecutorService extends Executor
{
    public boolean awaitTermination(long timeout, TimeUnit unit)
        throws InterruptedException;
    public <T> List<Future<T>> invokeAll(Collection<Callable<T>>
        tasks)
        throws InterruptedException;
    public <T> List<Future<T>> invokeAll(Collection<Callable<T>>
        tasks,
        long timeout,
        TimeUnit unit)
        throws InterruptedException;
    public <T> T invokeAny(Collection<Callable<T>> tasks)
        throws InterruptedException, ExecutionException;
    public <T> T invokeAny(Collection<Callable<T>> tasks,
        long timeout, TimeUnit unit)
        throws InterruptedException, ExecutionException,
        TimeoutException;
    public boolean isShutdown();
    public boolean isTerminated();
    public void shutdown();
    public List<Runnable> shutdownNow();
    public <T> Future<T> submit(Callable<T> task);
    public Future<?> submit(Runnable task);
    public <T> Future<T> submit(Runnable task, T result);
}
```

Mit Hilfe der *Submit-Methoden* kann ein einziger Auftrag (Callable oder Runnable) erteilt werden, mit *invokeAll* und *invokeAny* eine ganze Sammlung von Callable-Aufträgen (Collection ist eine Schnittstelle aus dem Package `java.util`, welche die Schnittstelle zu einer Menge von Elementen eines parametrisierten Typs vorgibt; sie enthält u. a. Methoden zum Hinzufügen und Entfernen von Elementen sowie zum Durchlaufen aller Elemente der Menge). Ein weiterer Unterschied zwischen *submit* und *invoke* ist, dass *submit* eine asynchrone Beauftragung darstellt (d. h., die Methode kehrt sofort zurück, nachdem der Auftrag erteilt wurde, in der Regel ist der Auftrag dann noch nicht abgearbeitet) und *invokeAll* bzw. *invokeAny* eine synchrone Auftragserteilung (d. h. bei allen zurückgegebenen Future-Objekten liefert die Methode *isDone* true zurück, da die Aufträge bei der Rückkehr alle bearbeitet sind). Die Methode *invokeAll* kehrt zurück, wenn alle Aufträge bearbeitet sind (deshalb wird auch

eine Liste von Future-Objekten zurückgegeben), während invokeAny zurückkehrt, wenn ein Auftrag erfolgreich zu Ende gelaufen ist (für diesen Auftrag wird das berechnete Ergebnis zurückgeliefert). Nachdem bei invokeAny ein Auftrag erfolgreich terminiert ist, werden alle anderen Aufträge automatisch abgebrochen. Beide Invoke-Methoden besitzen Varianten mit und ohne Angabe einer Zeit, welche die Frist darstellt, wie lange auf das Ende aller bzw. des ersten Auftrags höchstens gewartet werden soll.

Die Schnittstelle ExecutorService wird in der Regel durch einen *Thread-Pool* implementiert. Dieser Thread-Pool kann heruntergefahren werden mit der Methode *shutdown*. Es werden danach keine neuen Aufträge mehr angenommen, aber alle wartenden und sich gerade in Bearbeitung befindlichen Aufträge werden noch zu Ende ausgeführt. Die Variante *shutdownNow* beendet den Thread-Pool sofort, wobei die Aufträge, die noch in der Warteschlange gewartet haben, von shutdownNow als Rückgabewert zurückgeliefert werden. Auf das Ende des Thread-Pools kann man mit der Methode *awaitTermination* befristet warten. Bitte unterscheiden Sie das Warten bei invokeAll bzw. invokeAny und das Warten auf das Ende des Thread-Pools: Bei invokeAll bzw. invokeAny wird nur auf das Ende der Aufträge gewartet, die in der Methode als Parameter übergeben wurden, während bei awaitTermination die Methode shutdown zuvor aufgerufen worden sein muss und man dann auf das Ende aller Aufträge, die vor dem Aufruf von shutdown erteilt worden sind, wartet.

Eine Klasse, welche die ExecutorService-Schnittstelle implementiert, ist *ThreadPoolExecutor*. Sie besitzt mehrere Konstruktoren, wobei wir uns auf die Beschreibung des einfachsten mit fünf Parametern beschränken wollen:

```
public class ThreadPoolExecutor extends ...
{
    public ThreadPoolExecutor(int corePoolSize, int maximumPoolSize,
                             long keepAliveTime, TimeUnit unit,
                             BlockingQueue<Runnable> workQueue)
    {...}
    ...
}
```

Mit dem Argument *corePoolSize* wird die Anzahl der Threads angegeben, die im Thread-Pool immer mindestens existieren sollen, selbst dann, wenn es nichts zu tun gibt. Der zweite Parameter gibt an, auf welche Zahl die Anzahl der Threads höchstens wachsen soll. Der letzte Parameter gibt die Warteschlange an, in die neue Aufträge zur Bearbeitung eingestellt werden. *BlockingQueue* ist eine Schnittstelle u. a. mit den Methoden *put* und *take* zum Hineinstellen bzw. Herausnehmen von Elementen in die Warteschlange. Es gibt unterschiedliche Implementierungen dieser Schnittstelle (mehr dazu in Abschnitt 3.7.5). Präziser ausgedrückt verhält sich ein Thread-Pool wie folgt:

- Wenn weniger als *corePoolSize* Threads vorhanden sind (dies ist in jedem Fall zu Beginn der Fall), so wird für einen neuen Auftrag ein neuer Thread erzeugt, auch wenn schon Threads vorhanden sind, die nichts zu tun haben.
- Wenn die Anzahl der Threads schon *corePoolSize* oder mehr beträgt, aber noch weniger als *maximumPoolSize*, dann passiert Folgendes: Ist beim Einstellen eines neuen Auftrags die Warteschlange *workQueue* nicht voll, so wird der Auftrag eingereiht. Ist die Warteschlange dagegen voll, so wird ein neuer Thread gestartet.

- Wenn die Anzahl der Threads maximumPoolSize ist, verhält sich der Thread-Pool so: Wie oben wird ein neuer Auftrag in die Warteschlange eingereiht, falls noch Platz vorhanden ist. Andernfalls wird der Auftrag durch das Werfen einer Ausnahme abgelehnt.

Durch die Wahl der BlockingQueue kann man das Verhalten eines Thread-Pools stark beeinflussen (bei einer ArrayBlockingQueue mit vielen Speicherplätzen verhält sich der Thread-Pool beispielsweise anders als bei einer SynchronousQueue, die keinen Speicherplatz bietet). Dies erklärt zum Beispiel auch, dass man die Anzahl der Threads über corePoolSize hinaus nicht erhöhen kann, wenn man eine Warteschlange mit sehr vielen Pufferplätzen benutzt, die die eventuell vielen Aufträge alle aufnehmen kann.

Der dritte Parameter keepAliveTime des Konstruktors oben (mit der Angabe der Zeiteinheit wie z.B. Sekunden oder Minuten als viertem Parameter) legt die Zeit fest, die ein Thread unbeschäftigt ist, bevor er gelöscht wird, falls die Zahl der Threads die Anzahl corePoolSize übersteigt. Wenn corePoolSize und maximumPoolSize gleich groß sind, dann bedeutet das, dass die Anzahl der Threads im Pool in der Regel konstant bleibt, sobald die Anzahl der Threads auf corePoolSize bzw. maximumPoolSize gestiegen ist (Ausnahmen folgen unten).

Außer den Methoden der Schnittstelle ExecutorService hat die Klasse ThreadPoolExecutor noch einige weitere Methoden wie z.B. *getCompletedTaskCount* zum Erfragen der Anzahl der bislang ausgeführten Aufträge und *getLargestPoolSize* zum Erfragen der größten Anzahl der Threads, die jemals gleichzeitig im Pool existiert haben. Mit *preStartCoreThread* bzw. *preStartAllCoreThreads* wird ein Thread bzw. werden corePoolSize Threads gestartet, ohne dass ein Auftrag an den Pool übergeben wurde. Durch *allowCoreThreadTimeOut* mit Parameter true wird dem Thread-Pool erlaubt, Threads zu beenden, auch wenn die Anzahl der Threads dann weniger als corePoolSize beträgt.

Zur Illustration der Klasse ThreadPoolExecutor schreiben wir das Programm aus Abschnitt 2.4.1 um, welches die Anzahl der True-Elemente in einem booleschen Feld zählt, wobei unterschiedliche Bereiche des Felds durch unterschiedliche Threads bearbeitet werden (Listing 3.19). Wir benutzen einen Thread-Pool mit einer konstanten Anzahl von Threads. Zur Auftragserteilung wird die Methode *invokeAll* verwendet.

### **Listing 3.19**

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

class PooledService implements Callable<Integer>
{
    private boolean[] array;
    private int start;
    private int end;

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

    public Integer call() throws Exception
    {
        int result = 0;
```

```

        for(int i = start; i <= end; i++)
    {
        if(array[i])
        {
            result++;
        }
    }
    return result; //entspricht: return new Integer(result);
}

public class AsynchRequestThreadPool
{
    private static final int ARRAY_SIZE = 100000;
    private static final int NUMBER_OF_SERVERS = 1000;

    public static void main(String[] args)
    {
        long startTime = System.currentTimeMillis();

        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;
            }
        }

        LinkedList<Callable<Integer>> serviceList =
            new LinkedList<Callable<Integer>>();

        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;
            }
            serviceList.add(new PooledService(array, start, end));
            start = end + 1;
        }

        ThreadPoolExecutor pool =
            new ThreadPoolExecutor(NUMBER_OF_SERVERS,
                NUMBER_OF_SERVERS,
                0L, TimeUnit.SECONDS,

```

```
new LinkedBlockingQueue<Runnable>());  
  
try  
{  
    List<Future<Integer>> futureList;  
    futureList = pool.invokeAll(serviceList);  
  
    int result = 0;  
    for(Future<Integer> future: futureList)  
    {  
        result += future.get();  
    }  
  
    long endTime = System.currentTimeMillis();  
    float time = (endTime - startTime) / 1000.0f;  
    System.out.println("Rechenzeit: " + time);  
  
    System.out.println("Ergebnis: " + result);  
  
    pool.shutdown();  
}  
catch(Exception e)  
{  
}  
}  
}
```

Statt der Void-Methode run und einer selbst geschriebenen GetResult-Methode im Original-Beispiel (s. Listing 3.14) haben wir nun eine Call-Methode mit einem Rückgabewert des Typs Integer. Somit bekommen wir bei der Auftragserteilung Future-Objekte, deren Get-Methoden eben diese Integer-Objekte zurückliefern. Damit wir den Prozess nach Berechnung des Resultats nicht abbrechen müssen, fahren wir am Ende die Threads des Thread-Pools durch Aufruf der Methode shutdown herunter. Der Prozess endet dann von selbst.

Übrigens existiert eine Erweiterung der Schnittstelle `ExecutorService` namens `ScheduledExecutorService` zur Erteilung von Aufträgen, deren Ausführung sich periodisch wiederholen soll. Die Klasse `ScheduledThreadPoolExecutor` ist eine Implementierung dieser Schnittstelle. Weitere Details mögen sich die Leserinnen und Leser selber erschließen.

### 3.7.2 Locks und Conditions

Locks und Conditions sind allgemeine Synchronisationskonzepte, die in ihrer Wirkung mit synchronized, wait und notify bzw. notifyAll vergleichbar sind. Wer diese in Kapitel 2 eingeführten Mittel beherrscht, wird auch keine Probleme im Umgang mit Locks und Conditions haben. Locks sind in ihrer Wirkung mit synchronized vergleichbar. Es gibt vier wesentliche Unterschiede:

- *Locks* sind im Gegensatz zu `synchronized` nicht als Schlüsselwort in die Sprache Java eingebaut, sondern zum Setzen einer *Sperre* muss die Methode `lock` und zum Freigeben einer Sperre muss die Methode `unlock` aufgerufen werden. Damit ist man natürlich nicht an eine strikte Blockstruktur gebunden. So kann z. B. in einer Methode die Sperre gesetzt und in einer anderen Methode die Sperre freigegeben werden (allerdings ist es nicht

möglich, dass ein Thread eine Sperre setzt und ein anderer Thread diese Sperre freigibt). Abhängig von der Perspektive kann dies als Vor- oder als Nachteil gesehen werden.

- Da man zum Sperren Methoden aufrufen muss, kann man auf das Setzen einer Sperre bei Wunsch auch befristet warten. Auch ist es möglich, das Warten eines Threads auf das Setzen einer Sperre durch *interrupt* zu unterbrechen.
- Es ist möglich, Lese- und Schreibsperren zu unterscheiden.
- Man kann sich Implementierungen vorstellen, die beim Setzen von Sperren überprüfen, ob bei Gewährung der Sperre eine Verklemmungssituation entstehen würde. Falls ja, kann die Sperranforderung zurückgewiesen werden (Verklemmungen s. Abschnitte 3.11 und 3.12).

*Lock* ist eine Schnittstelle mit folgenden Methoden:

```
public interface Lock
{
    public void lock();
    public void lockInterruptibly()
        throws InterruptedException;
    public Condition newCondition();
    public boolean tryLock();
    public boolean tryLock(long time, TimeUnit unit)
        throws InterruptedException;
    public void unlock();
}
```

Mit *lock* wird eine Sperre gesetzt. Falls die Sperre momentan von einem anderen Thread gesetzt ist, wartet der Thread in nicht unterbrechbarer Weise. Dies kann man sich so vorstellen wie das Warten z.B. in der von uns selbst programmierten Methode *p* der Klasse *Semaphore* (s. Abschnitt 3.1). Hier wird *wait* in einem Try-Catch-Block innerhalb einer While-Schleife aufgerufen. Eine Unterbrechung verursacht nur ein kurzzeitiges Verlassen der *Wait*-Methode, da diese anschließend wieder aufgerufen wird. In *lockInterruptibly* kann man sich einen *Wait*-Aufruf in einer While-Schleife ohne Try-Catch-Block vorstellen. Wenn der Thread in diesem Fall in *wait* unterbrochen wird, wird der Methodenaufruf mit einer Ausnahme beendet. Mit *tryLock* kann man versuchen, eine Sperre zu setzen. Wenn die Sperre gesetzt wurde, liefern die TryLock-Methoden *true* zurück, sonst *false*. In der einen Variante kann eine Zeit angegeben werden, wie lange auf das Setzen der Sperre höchstens gewartet werden soll. Die TryLock-Methode ohne Parameter entspricht der TryLock-Methode mit dem Parameterwert 0 (d.h. es wird in keinem Fall gewartet, die Sperre wird gesetzt, wenn dies möglich ist, im anderen Fall kehrt man sofort ohne Wirkung aus der Methode zurück). Auf die Methode *newCondition* kommen wir unten zu sprechen.

Da jetzt keine Blockstruktur für das Setzen und Freigeben von Sperren mehr vorgegeben ist, kann es bei einem unsorgfältigen Einsatz der Locks zu Problemen kommen:

```
Lock l = ...;
l.lock();
...
l.unlock();
```

Wenn zwischen dem Aufruf von *lock* und *unlock* eine Ausnahme ausgelöst wird, dann wird die *Unlock*-Anweisung nicht mehr ausgeführt, die Sperre bleibt gesetzt und ist für keinen

Thread mehr zu setzen. Deshalb wird empfohlen, das Freigeben der Sperre immer in einem Finally-Block auszuführen, der auch dann ausgeführt wird, wenn eine Ausnahme geworfen oder eine Return-Anweisung in dem Try-Block ausgeführt wird:

```
Lock l = ...;
l.lock();
try
{
    ...
}
finally
{
    l.unlock();
}
```

Eine Klasse, welche die Schnittstelle Lock implementiert, ist *ReentrantLock*. Sie besitzt einen Konstruktor ohne Argumente und einen Konstruktor mit einem Argument des Typs boolean, über den angegeben werden kann, ob eine *faire Bedienreihenfolge* der auf die Sperre wartenden Threads gewünscht wird oder nicht (vgl. das faire Parkhaus aus Abschnitt 2.6.3). Neben den Methoden der Schnittstelle Lock bietet ReentrantLock einige Methoden zum Abfragen des Lock-Status an (z.B. mit *getQueueLength* kann abgefragt werden, wie viele Threads auf das Setzen der Sperre warten). Der Begriff Reentrant weist auf eine Eigenschaft der Implementierung hin, die auch synchronized besitzt und auf die in Abschnitt 2.3.3 (s. Listing 2.11) hingewiesen wurde: Wenn ein Thread eine Sperre setzen will und diese ist schon gesetzt, dann wird zusätzlich geprüft, von welchem Thread die Sperre im Augenblick gehalten wird. Wenn dies derselbe Thread wie der anfordernde Thread ist, dann muss nicht gewartet werden, sondern stattdessen wird ein Sperrenzähler erhöht. Entsprechend wird bei unlock die Sperre nicht in jedem Fall freigegeben, sondern nur dann, wenn der Sperrenzähler nach dem Herunterzählen 0 ist. Der aktuelle Stand des Sperrenzählers kann mit der Methode *getHoldCount* der Klasse ReentrantLock abgefragt werden.

Zur Unterstützung von *Lese- und Schreibsperren* wird eine neue Schnittstelle namens *ReadWriteLock* definiert. Diese Schnittstelle besitzt aber keine Methoden zum Setzen und Freigeben von Lese- und Schreibsperren, sondern mit dieser Schnittstelle kann man sich zwei Lock-Objekte, eines für das Lesen und eines für das Schreiben, geben lassen:

```
public interface ReadWriteLock
{
    public Lock readLock();
    public Lock writeLock();
}
```

Vor dem Lesen muss man dann die Lock-Methode auf die *Lesesperre* anwenden. Entsprechendes gilt für das Schreiben. Die Lesesperre kann von mehreren Threads gleichzeitig gesetzt sein, die *Schreibsperre* maximal von einem Thread zu einem Zeitpunkt. Natürlich sind die beiden Sperren miteinander gekoppelt; sobald eine Lesesperre gesetzt ist, kann die Schreibsperre nicht mehr gesetzt werden, und umgekehrt. Die Klasse *ReentrantReadWriteLock* ist eine Implementierung der Schnittstelle ReadWriteLock.

Soweit zum Thema Locks, welche dieselbe Grundfunktionalität wie synchronized bereitstellen, aber offensichtlich in flexiblerer und vielfältigerer Weise. Zur Lösung allgemeiner Synchronisationsprobleme fehlen nun natürlich noch Methoden, welche wait, notify und

`notifyAll` entsprechen. Solche Methoden befinden sich in der Schnittstelle *Condition*; statt `wait` gibt es *await* (in mehreren Varianten), statt `notify` *signal* und statt `notifyAll` *signalAll*:

```
public interface Condition
{
    public void await()
        throws InterruptedException;
    public boolean await(long time, TimeUnit unit)
        throws InterruptedException;
    public long awaitNanos(long nanosTimeout)
        throws InterruptedException;
    public void awaitUninterruptibly();
    public boolean awaitUntil(Date deadline)
        throws InterruptedException;
    public void signal();
    public void signalAll();
}
```

Ohne Erläuterung dürften die unterschiedlichen Varianten von `await` aufgrund der Parameternamen verständlich sein. Wir erinnern uns daran, dass beim Aufruf von `wait`, `notify` und `notifyAll` das Objekt, auf das diese Methoden angewendet werden, zu diesem Zeitpunkt vom aufrufenden Thread gesperrt sein muss, dass ferner bei `wait` die Sperre freigegeben wird, und dass nach dem Wecken mit `notify` oder `notifyAll` die Sperre vom geweckten Thread wieder angefordert werden muss, bevor der Aufruf von `wait` zurückkehrt. Zusammengefasst heißt dies, dass `wait`, `notify` und `notifyAll` in engem Bezug zu einem gesperrten Objekt stehen. Bei den Conditions besteht entsprechend eine solche enge Beziehung zu einem Lock. Aus diesem Grund werden Condition-Objekte nicht mit `new` erzeugt, sondern man lässt sich mit der Methode `newCondition` (beachten Sie das nicht vorhandene Leerzeichen zwischen `new` und `Condition`) ein Condition-Objekt von einem Lock-Objekt generieren (siehe Schnittstelle Lock oben). Jedes Condition-Objekt ist somit mit seinem erzeugenden Lock-Objekt assoziiert. Es gilt nun dasselbe wie zuvor bei `wait`, `notify` und `notifyAll`: Beim Aufruf von `await` wird die Sperre des mit dem Condition-Objekt assoziierten Lock-Objekts freigegeben, wobei der aufrufende Thread diese Sperre natürlich gesetzt haben sollte. Nach dem Aufwachen durch `signal`, `signalAll` oder `interrupt` muss die Sperre wieder gesetzt werden, bevor man aus der Methode `await` zurückkehrt.

Ein Thread, der `await` aufruft, wartet am Condition-Objekt. Entsprechend wird durch `signal` ein an diesem Condition-Objekt wartender Thread bzw. werden durch `signalAll` alle an diesem Condition-Objekt wartenden Threads aufgeweckt. Der entscheidende Unterschied zu `wait`, `notify` und `notifyAll` ist nun, dass es zu jedem Lock mehrere Condition-Objekte geben kann. Damit erhält man die Möglichkeit, Threads gezielter zu wecken, als dies zuvor möglich war. Als Beispiel betrachten wir hierzu nochmals das *Erzeuger-Verbraucher-Problem* aus den Abschnitten 2.6.1 (Listing 2.27) und 2.6.2 (Listing 2.30). Da es unterschiedliche Wartebedingungen gab (die einen warten, bis der Puffer etwas enthält, die anderen warten, bis der Puffer Platz zum Ablegen eines neuen Werts bietet), mussten wir `notifyAll` statt `notify` benutzen, sofern Objekte der Klasse Buffer auch von mehreren Erzeuger- und Verbraucher-Threads ohne Probleme nutzbar sein sollten. Dies ist hier nun anders, wenn wir für die unterschiedlichen Wartebedingungen unterschiedliche Condition-Objekte heranziehen (Listing 3.20):

**Listing 3.20**

```
import java.util.concurrent.locks.*;  
  
class BufferLock  
{  
    private int head;  
    private int tail;  
    private int count;  
    private int[] data;  
    private ReentrantLock lock;  
    private Condition notFull;  
    private Condition notEmpty;  
  
    public BufferLock(int n)  
    {  
        head = 0;  
        tail = 0;  
        count = 0;  
        data = new int[n];  
        lock = new ReentrantLock();  
        notFull = lock.newCondition();  
        notEmpty = lock.newCondition();  
    }  
  
    private void dump()  
    {  
        System.out.print("\t\t\tPufferinhalt: [ ");  
        int index = head;  
        for(int i = 0; i < count; i++)  
        {  
            System.out.print(data[index] + " ");  
            index++;  
            if(index == data.length)  
            {  
                index = 0;  
            }  
        }  
        System.out.println("]");  
    }  
  
    public void put(int x)  
    {  
        lock.lock();  
        try  
        {  
            while(count == data.length)  
            {  
                notFull.awaitUninterruptibly();  
            }  
            data[tail++] = x;  
            if(tail == data.length)  
            {  
                tail = 0;  
            }  
            count++;  
            dump();  
            notEmpty.signal();  
        }  
        finally  
        {  
            lock.unlock();  
        }  
    }  
}
```

```
        }
        finally
        {
            lock.unlock();
        }
    }

    public int get()
    {
        lock.lock();
        try
        {
            while(count == 0)
            {
                notEmpty.awaitUninterruptibly();
            }
            int result = data[head++];
            if(head == data.length)
            {
                head = 0;
            }
            count--;
            dump();
            notFull.signal();
            return result;
        }
        finally
        {
            lock.unlock();
        }
    }
}

class ProducerLock extends Thread
{
    private BufferLock buffer;
    private int start;

    public ProducerLock(BufferLock b, int s, String name)
    {
        super(name);
        buffer = b;
        start = s;
    }

    public void run()
    {
        for(int i = start; i < start + 100; i++)
        {
            buffer.put(i);
        }
    }
}

class ConsumerLock extends Thread
{
    private BufferLock buffer;
```

```

public ConsumerLock(BufferLock b, String name)
{
    super(name);
    buffer = b;
}

public void run()
{
    for(int i = 0; i < 100; i++)
    {
        int x = buffer.get();
        System.out.println("verbraucht " + x);
    }
}

public class ProducerConsumerLock
{
    public static void main(String[] args)
    {
        BufferLock p = new BufferLock(5);
        ConsumerLock v1 = new ConsumerLock(p, "V1");
        ConsumerLock v2 = new ConsumerLock(p, "V2");
        ConsumerLock v3 = new ConsumerLock(p, "V3");
        ProducerLock e1 = new ProducerLock(p, 1, "E1");
        ProducerLock e2 = new ProducerLock(p, 101, "E2");
        ProducerLock e3 = new ProducerLock(p, 201, "E3");
        v1.start();
        v2.start();
        v3.start();
        e1.start();
        e2.start();
        e3.start();
    }
}

```

Im Gegensatz zu der Buffer-Klasse aus Kapitel 2 bietet die Klasse BufferLock Platz für mehr als einen Int-Wert. Wie bei den Message Queues aus Abschnitt 3.2 benutzen wir ein Feld, das wir zyklisch nutzen. Das Entscheidende an dem Beispielprogramm ist nun die Tatsache, dass zwei Condition-Objekte notFull und notEmpty im Konstruktor vom Lock-Objekt erzeugt werden. In der Methode put wird bei vollem Puffer an dem Condition-Objekt notFull gewartet und nach dem Ablegen des Werts das andere Condition-Objekt notEmpty signalisiert. In der Methode get ist es umgekehrt: man wartet bei leerem Puffer an dem Condition-Objekt notEmpty und signalisiert nach dem Entnehmen eines Objekts das Condition-Objekt notFull. Man signalisiert damit also immer den „richtigen“ Thread. Da durch das Einstellen eines Werts höchstens ein wartender Verbraucher bzw. durch das Entnehmen eines Werts höchstens ein wartender Erzeuger weiterlaufen kann, genügt in beiden Fällen eine Signialisierung mit signal (statt signalAll). Wie für notifyAll gilt jetzt, dass signalAll verwendet werden muss, wenn an einem Condition-Objekt Threads mit unterschiedlichen Wartebedingungen warten, oder wenn die vorgenommene Veränderung der Datenstruktur so ist, dass mehrere Threads weiterlaufen können. Beide Bedingungen treffen z.B. bei einer Implementierung von additiven Semaphoren zu (s. Listing 3.21):

**Listing 3.21**

```
import java.util.concurrent.locks.*;  
  
public class AdditiveSemaphoreLock  
{  
    private int value;  
    private ReentrantLock lock;  
    private Condition condition;  
  
    public AdditiveSemaphoreLock(int init)  
    {  
        if(init < 0)  
        {  
            throw new IllegalArgumentException("Parameter < 0");  
        }  
        this.value = init;  
        lock = new ReentrantLock();  
        condition = lock.newCondition();  
    }  
  
    public void p(int x)  
    {  
        if(x <= 0)  
        {  
            throw new IllegalArgumentException("Parameter < 0");  
        }  
        lock.lock();  
        try  
        {  
            while(value - x < 0)  
            {  
                condition.awaitUninterruptibly();  
            }  
            value -= x;  
        }  
        finally  
        {  
            lock.unlock();  
        }  
    }  
  
    public void v(int x)  
    {  
        if(x <= 0)  
        {  
            throw new IllegalArgumentException("Parameter < 0");  
        }  
        lock.lock();  
        try  
        {  
            value += x;  
            condition.signalAll();  
        }  
        finally  
        {  
            lock.unlock();  
        }  
    }  
}
```

### 3.7.3 Atomic-Klassen

In Abschnitt 2.3.6 wurde klargestellt, dass parallele lesende und schreibende Zugriffe auf Attribute eines Objekts synchronisiert werden müssen, selbst dann, wenn es sich um Attribute von Basisdatentypen wie int oder boolean handelt. Im Atomic-Package werden nun vordefinierte Klassen mit einem einzigen Attribut des Typs Boolean (*AtomicBoolean*), Integer (*AtomicInteger*), einem Feld von Integer (*AtomicIntegerArray*), Long (*AtomicLong*), einem Feld von Long (*AtomicLongArray*), einer Referenz (*AtomicReference*), einem Feld von Referenzen (*AtomicReferenceArray*) usw. bereitgestellt, die u. a. Methoden zum Lesen und Schreiben dieser einfachen Attribute besitzen, wobei diese Methoden für die parallele Nutzung geeignet (also „*thread-sicher*“ bzw. „*thread-safe*“) sind. Neben einfachen Get- und Set-Methoden besitzen alle diese Klassen auch Methoden zum Vergleichen und Ändern „auf einen Schlag“ (d. h. in *atomarer [unteilbarer]* Weise). Um besser zu verstehen, was damit gemeint ist, wird im Folgenden Java-Code für die Methode compareAndSet der Klasse AtomicInteger angegeben, wobei aber betont werden soll, dass damit nur die Funktion dieser Methode erklärt wird, dass diese Methode aber eben nicht so wie hier gezeigt implementiert ist:

```
public class AtomicInteger
{
    private int i;

    public synchronized boolean compareAndSet(int expect,
                                              int update)
    {
        if(i == expect)
        {
            i = update;
            return true;
        }
        return false;
    }
}
```

Wenn also der Wert der ganzen Zahl gleich dem ersten Parameter *expect* ist, wird der Wert auf *update* geändert, andernfalls passiert nichts. Über den Rückgabewert wird angezeigt, ob eine Änderung stattgefunden hat oder nicht. Das Besondere an dieser Methode ist nun, dass sie sehr effizient programmiert werden kann, weil die Befehlssätze von Prozessoren üblicherweise Instruktionen haben, die cas (compare and set), tas (test and set) oder so ähnlich heißen und in ihrer Wirkung genau der oben beschriebenen Methode compareAndSet entsprechen. Das heißt, wir gehen im Folgenden davon aus, dass die Methode compareAndSet nicht wie oben gezeigt implementiert ist, sondern irgendwie anders, aber in jedem Fall sehr effizient (im Idealfall durch Ausführung eines einzigen Maschinenbefehls und insbesondere ohne die Verwendung von synchronized).

Mit Hilfe der Methode compareAndSet lassen sich nun auch andere Methoden der Atomic-Klassen effizient realisieren. Als Beispiel wird im Folgenden eine mögliche Implementierung der Methode incrementAndGet der Klasse AtomicInteger gezeigt. Diese Methode soll den Attributwert des Objekts in thread-safer Weise um eins erhöhen und den neuen Wert zurückgeben. Der nächste Wert wird ausgehend vom aktuellen Wert berechnet. Wenn danach der Wert, von dem man bei der Berechnung ausgegangen ist, noch aktuell ist und

nicht verändert wurde, dann wird der berechnete Wert übernommen. Da die Methoden der Klasse AtomicInteger jetzt nicht mehr synchronized sind, machen wir das Attribut der Klasse dafür aber volatile.

```
public class AtomicInteger
{
    private volatile int i;

    public boolean compareAndSet(int expect, int update)
    {
        //effiziente Implementierung (nicht in Java)
        ...
    }

    public int incrementAndGet()
    {
        int current, next;
        do
        {
            current = i;
            next = current + 1;
        } while(!compareAndSet(current, next));
        return next;
    }
}
```

Der Programmcode der Methode incrementAndGet zeigt, dass eventuell mehrere Versuche unternommen werden, um die Erhöhung durchzuführen. Im Prinzip ist dies auch eine Form des aktiven Wartens, das im vorausgehenden Text (insbesondere dem vorigen Kapitel 2) verteuft wurde. An dieser Stelle müssen wir ein klein wenig zurückrudern: Unter besonderen Bedingungen kann aktives Warten sogar die bessere Lösung sein, nämlich zum Beispiel dann, wenn das Warten von sehr kurzer Dauer ist oder wenn die Anzahl der Threads die Anzahl der Prozessoren nicht übersteigt. So geht man im obigen Beispiel bei der Methode incrementAndGet von der Annahme aus, dass sich die Anzahl der Versuche zum Erhöhen des Werts in engen Grenzen halten wird, so dass nur ganz kurz aktiv gewartet wird, bis das Erhöhen geklappt hat. In einem solchen Fall erspart das wiederholte Versuchen eine Thread-Umschaltung und ist somit wesentlich effizienter als eine Lösung mit synchronized, Semaphoren oder vergleichbaren Mechanismen. Denn die Thread-Umschaltung benötigt die Ausführung von viel mehr Anweisungen als die wenigen zu erwartenden Schleifendurchläufe in der Methode incrementAndGet.

Die Klasse AtomicInteger hat außer compareAndSet, incrementAndGet und natürlich get und set weitere Methoden wie decrementAndGet oder addAndGet. Daneben gibt es auch Methoden, die eine Änderung durchführen und den alten Wert (den Wert vor der Veränderung) zurückliefern wie getAndSet, getAndIncrement, getAndDecrement oder getAndAdd. Mit Java 8 sind akkumulierende Methoden dazugekommen, deren Bedeutung anhand der folgenden Aufgabenstellung erklärt werden soll: Angenommen, mehrere Threads errechnen jeweils einen Wert und das Maximum dieser Werte soll bestimmt und gespeichert werden. Unter Nutzung von AtomicInteger könnte man im ersten Versuch eine Klasse Max programmieren. Ein Objekt dieser Klasse würde dann von allen Threads gemeinsam benutzt; jeder Thread würde auf diesem Objekt die Methode update aufrufen mit seinem eigenen Wert als Argument:

```
public class Max
{
    private AtomicInteger max = new AtomicInteger();
    public void update(int newValue)
    {
        max.set(Math.max(max.get(), newValue));
    }
}
```

Dass dies nicht korrekt funktioniert, sollte allen, die dieses Buch bis hierhin durchgearbeitet haben, klar sein und muss sicher nicht erläutert werden. Eine korrekte Lösung unter Verwendung der zuvor besprochenen Methode compareAndSet könnte so aussehen:

```
public class Max
{
    private AtomicInteger max = new AtomicInteger();

    public void update(int newValue)
    {
        int o, n;
        do
        {
            o = max.get();
            n = Math.max(o, newValue);
        } while(!max.compareAndSet(o, n));
    }
}
```

Wie bei incrementAndGet wird ein neuer Wert basierend auf dem alten Wert berechnet und nur, wenn der alte Wert noch gültig ist, wird der neue Wert „auf einen Schlag“ übernommen. In Java 8 wurde für derartige Aufgabenstellungen eine Methode namens accumulateAndGet hinzugefügt, die folgende Signatur hat:

```
public int accumulateAndGet(int x, IntBinaryOperator op);
```

IntBinaryOperator ist eine funktionale Schnittstelle mit einer einzigen Methode, die zwei Argumente des Typs int erwartet und einen Wert des Typs int zurückgibt. Beim Aufruf von accumulateAndGet wird der zweistellige Operator auf den aktuellen Wert des AtomicInteger-Objekts und den Parameter x angewendet. Anschließend wird der Wert des AtomicInteger-Objekts auf das Ergebnis dieser Operation gesetzt. Diese beiden Schritte erfolgen in atomarer Weise (also thread-sicher). Der gesetzte Wert wird dann auch zurückgegeben (es gibt auch in diesem Fall die Variante getAndAccumulate, die den alten Wert zurückgibt). Unsere Methode update der Klasse Max lässt sich nun mit accumulateAndGet und einem Lambda-Ausdruck wesentlich kompakter schreiben:

```
public void update(int newValue)
{
    max.accumulateAndGet(newValue, (x1, x2) -> Math.max(x1, x2));
}
```

AtomicReference ist AtomicInteger sehr ähnlich. AtomicReference ist jedoch eine generische Klasse und kapselt eine Referenz auf ein Objekt des Typs T, wobei T der Typparameter ist. Da man mit Referenzen nicht rechnen kann, existieren die Increment-, Decrement und Add-Methoden nicht, wohl aber z. B. compareAndSet sowie die akkumulierenden Methoden.

Als Beispiel für die Nutzung von AtomicReference sehen wir uns in Listing 3.22 eine Implementierung eines parallel nutzbaren Kellers an. Die Besonderheit dabei ist, dass die Synchronisation nicht über synchronized erfolgt, sondern mit Hilfe einer AtomicReference auf das erste und damit oberste Element einer einfach verketteten Liste. Die Liste enthält Objekte der Klasse Element. Element ist eine generische Klasse. Der Typparameter gibt den Typ der Objekte an, die auf den Keller gelegt werden sollen. Neben dem eigentlichen Objekt, das sich im Keller befindet, enthält jedes Element-Objekt eine Referenz auf seinen Nachfolger in der Liste. Da wir hier ganz bewusst ohne synchronized auskommen wollen, sind alle Attribute mit volatile gekennzeichnet.

**Listing 3.22**

```
import java.util.concurrent.atomic.*;  
  
class Element<T>  
{  
    private volatile T value;  
    private volatile Element<T> next;  
  
    public Element(T value, Element<T> next)  
    {  
        this.value = value;  
        this.next = next;  
    }  
  
    public T getValue()  
    {  
        return value;  
    }  
  
    public Element<T> getNext()  
    {  
        return next;  
    }  
}  
  
public class SyncStack<T>  
{  
    private volatile AtomicReference<Element<T>> ar;  
  
    public SyncStack()  
    {  
        ar = new AtomicReference<>();  
    }  
  
    public void push(T t)  
    {  
        Element<T> oldHead, newHead;  
        do  
        {  
            oldHead = ar.get();  
            newHead = new Element<>(t, oldHead);  
        } while(!ar.compareAndSet(oldHead, newHead));  
    }  
  
    public T pop()  
}
```

```

{
    Element<T> oldHead, newHead;
    do
    {
        oldHead = ar.get();
        if(oldHead == null)
        {
            return null;
        }
        newHead = oldHead.getNext();
    } while(!ar.compareAndSet(oldHead, newHead));
    return oldHead.getValue();
}
}

```

Beim Hinzufügen (push) und beim Wegnehmen eines Elements (pop) wird ein neues oberstes Element basierend auf dem bisherigen obersten Element bestimmt. Wenn sich danach am obersten Element nichts geändert hat, wird das neue oberste Element mit compareAndSet gesetzt. Diese Realisierung ohne Sperren (*lockfree*) ist einer Realisierung z.B. mit synchronized überlegen, wenn es höchsten so viele Threads wie Prozessoren bzw. Prozessorkerne gibt. In einem solchen Fall macht es keinen Sinn, einem Thread durch synchronized den Prozessor zu entziehen. Da ein Thread keinem anderen den Prozessor wegnimmt, ist es viel effizienter, wenn ein Thread eventuell mehrere Versuche unternimmt um die Datenstruktur zu aktualisieren. Außerdem erwarten wir auch in diesem Fall wieder, dass immer nur sehr wenige Versuche gebraucht werden, um den Keller zu aktualisieren. Ein zweiter oder dritter Versuch ist immer noch weniger aufwändig als eine Thread-Umschaltung, die z.B. durch synchronized bewirkt wird.

Zwei Varianten von AtomicReference sind AtomicMarkableReference und AtomicStampedReference. AtomicMarkableReference hat neben der Referenz zusätzlich ein Boolean-Attribut, das atomar in compareAndSet mit der Referenz gelesen und gesetzt werden kann. AtomicStampedReference ist ähnlich, nur gibt es statt des zusätzlichen Boolean-Attributs ein zusätzliches Attribut des Typs int, das z.B. als Versionszähler verwendet werden kann. Mit einer Referenz liest man auch ihre dazugehörige Versionsnummer.

### 3.7.4 Synchronisationsklassen

Das „klassische“ Synchronisationskonzept ist der *Semaphor*, der zu Beginn dieses Kapitels (Abschnitt 3.1) in mehreren Varianten vorgestellt wurde. In der Concurrent-Klassenbibliothek gibt es auch eine Semaphorklasse, die in seiner Funktionalität dem additiven Semaphor (Abschnitt 3.1.4) entspricht. Wie bei unserer selbst geschriebenen Semaphorklasse gibt es einen Konstruktor mit einem Int-Parameter, der den Anfangswert des Semaphors vorgibt. Ein zweiter Konstruktor hat neben dem Int-Argument einen zweiten Parameter vom Typ boolean, mit dem eingestellt werden kann, ob der Semaphor eine *faire Bedienstrategie* realisieren muss oder nicht (vgl. das faire Parkhaus aus Abschnitt 2.6.3). Die zuvor als p bezeichnete Methode heißt nun *acquire*, v wird *release* genannt. Es existieren mehrere Varianten von acquire: solche ohne Angabe, um wie viel der Zählerwert reduziert werden soll, was eine Reduktion um 1 bedeutet und solche, bei denen als Parameter angegeben werden

kann, um wie viel der Wert heruntergezählt werden soll; solche, die unterbrochen werden können und solche, die nicht unterbrochen werden können; solche mit befristeter Wartezeit und solche ohne Befristung. Die Methode `release` hat nur zwei Varianten: eine ohne Parameter (Erhöhung des Zählers um 1) und eine mit einem Int-Parameter, der angibt, um wie viel der Zähler erhöht werden soll. Da wir Semaphore ausführlich in Abschnitt 3.1 besprochen haben, erübrigt sich eine weitere Diskussion hier.

Die Klasse `CountDownLatch` repräsentiert einen Zähler, dessen positiver Anfangswert im Konstruktor durch ein Int-Argument festgelegt werden muss. Mit der Methode `countDown` wird der Wert des Zählers um eins erniedrigt. Mit der Methode `await` können ein oder mehrere Threads darauf warten, bis der Zähler 0 wird. Es gibt keine Methoden zum Erhöhen des Zählers.

Eine Art Treffpunkt wird durch die Klasse `CyclicBarrier` nachgeahmt. Im Konstruktor gibt man als Int-Wert an, wie viele Teilnehmerinnen und Teilnehmer am Treffpunkt erwartet werden. Durch Aufruf der Methode `await` wartet man am Treffpunkt auf alle anderen. Das heißt, wenn N Threads erwartet werden, dann werden die ersten N-1 Threads blockiert, die `await` aufrufen. Der Thread Nr. N wird beim Aufruf von `await` nicht blockiert, befreit aber die N-1 anderen Threads aus ihrer Blockade.

Die letzte hier behandelte Klasse ist `Exchanger`. Sie ist zwar keine reine Synchronisationsklasse wie die anderen drei, denn es werden hier auch Daten ausgetauscht. Wir behandeln sie aber dennoch in diesem Abschnitt, da sie Ähnlichkeiten mit `CyclicBarrier` besitzt. Auch bei `Exchanger` warten wie bei `CyclicBarrier` Threads aufeinander. Da bei `Exchanger` aber noch Daten ausgetauscht werden, können sich zu einem Zeitpunkt immer nur zwei Threads treffen. Man kann sich das Ganze vorstellen, als ob sich zwei Personen an einem vereinbarten Treffpunkt treffen, um Gegenstände auszutauschen, ohne die Gegenstände dabei aber irgendwo abzustellen. Das heißt, die erste Person muss warten, bis die zweite da ist, um ihren Gegenstand zu übergeben. Dabei übernimmt sie gleichzeitig den Gegenstand von der anderen Person. Der Typ der ausgetauschten Gegenstände wird in der Klasse `Exchanger` parametrisiert. Neben einem parameterlosen Konstruktor gibt es zwei Varianten der Methode `exchange` zum Warten und Tauschen der Gegenstände, eine mit und eine ohne Angabe einer Wartefrist:

```
public class Exchanger<V>
{
    public Exchanger() {...}
    public V exchange(V x) {...}
    public V exchange(V x, long timeout, TimeUnit unit) {...}
}
```

Listing 3.23 zeigt ein Beispiel, bei dem zwei Threads Integer-Werte mit Hilfe eines `Exchanger`-Objekts austauschen:

#### Listing 3.23

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

class ExchangeThread extends Thread
{
    private int startValue;
    private int numberOfValues;
```

```

private Exchanger<Integer> exchanger;

public ExchangeThread(String name, int s, int n,
                      Exchanger<Integer> e)
{
    super(name);
    this.startValue = s;
    this.numberOfValues = n;
    this.exchanger = e;
}

public void run()
{
    try
    {
        for(int i = startValue; i < startValue + numberOfValues;
            i++)
        {
            Integer iToSend = new Integer(i);
            Integer iReceived = exchanger.exchange(iToSend);
            System.out.println("Thread "
                               + Thread.currentThread().getName()
                               + ": gegeben " + iToSend
                               + ", genommen " + iReceived);
        }
    }
    catch(InterruptedException ex)
    {
    }
}
}

public class ExchangerDemo
{
    public static void main(String[] args)
    {
        Exchanger<Integer> exchanger = new Exchanger<Integer>();
        ExchangeThread t1 = new ExchangeThread("T1", 101, 2,
                                              exchanger);
        ExchangeThread t2 = new ExchangeThread("T2", 201, 2,
                                              exchanger);
        t1.start();
        t2.start();
    }
}

```

Eine mögliche Ausgabe des Programms sieht so aus:

```

Thread T1: gegeben 101, genommen 201
Thread T2: gegeben 201, genommen 101
Thread T1: gegeben 102, genommen 202
Thread T2: gegeben 202, genommen 102

```

Werden statt zwei vier Threads gestartet, die jeweils zwei Mal die Exchange-Methode wie in obigem Beispiel aufrufen, so ist übrigens nicht garantiert, dass der Prozess zu Ende läuft, was auf den ersten Blick überraschend sein mag. Es könnte nämlich z. B. sein, dass T1 und

T2, T1 und T3 und T2 und T3 je einmal tauschen. Dann haben T1, T2 und T3 je zwei Mal getauscht und terminieren. T4 hat nun keinen Partner mehr zum Tauschen, bleibt für immer in exchange blockiert und läuft somit nicht zu Ende. Am besten ist es daher, immer nur zwei Threads mit einem Exchanger-Objekt arbeiten zu lassen, da sonst eventuell etwas Unvorhergesehenes passiert.

### 3.7.5 Queues

In der Concurrent-Bibliothek gibt es einige Implementierungen von Warteschlangen (Queues), die wie die Message Queue aus Abschnitt 3.2 und der mit Locks und Conditions implementierte Puffer aus Abschnitt 3.7.2 dem *Erzeuger-Verbraucher-Prinzip* entsprechend implementiert sind. Das heißt: Der Versuch, ein Objekt in eine volle Warteschlange abzulegen, blockiert den aufrufenden Thread so lange, bis die Warteschlange nicht mehr voll ist. Entsprechend wird ein Thread blockiert, wenn er versucht, ein Objekt aus einer leeren Warteschlange zu entnehmen. Und zwar so lange, bis sich mindestens ein Element in der Warteschlange befindet.

Die grundlegende Schnittstelle aller Queues ist *BlockingQueue*. Die Schnittstelle ist parametrisiert mit dem Datentyp der Elemente, die in die Warteschlange eingereiht werden können. Die grundlegenden Methoden sind *put* zum Einfügen und *take* zum Entnehmen eines Elements. Beide Methoden warten gegebenenfalls.

```
public interface BlockingQueue<E>
{
    public void put(E o) throws InterruptedException;
    public E take() throws InterruptedException;
    ...
}
```

Daneben gibt es weitere Methoden, die das Einfügen bzw. Entnehmen von Elementen versuchen, aber nicht oder nur eine begrenzte Zeit warten, falls die Operation nicht sofort möglich ist (*offer* und *add* zum Einfügen, *poll* zum Entnehmen). Es gibt fünf Klassen, die die Schnittstelle *BlockingQueue* implementieren: *ArrayBlockingQueue*, *LinkedBlockingQueue*, *SynchronousQueue*, *DelayQueue* und *PriorityBlockingQueue*.

Die „klassische“ Implementierung der Schnittstelle *BlockingQueue* wird durch die Klasse *ArrayBlockingQueue* repräsentiert. Wie in unserer Message Queue wird zur Realisierung der Warteschlange ein Feld verwendet, dessen Größe zu Beginn (beim Aufruf des Konstruktors) durch ein Argument festgelegt wird.

Die Klasse *LinkedBlockingQueue* bietet eine Implementierung von *BlockingQueue* mit ähnlichem Verhalten an. Die Warteschlange wird in diesem Fall durch eine verkettete Liste realisiert. Im Konstruktor kann wie bei *ArrayBlockingQueue* die Größe der Warteschlange vorgegeben werden. Es gibt aber auch einen Konstruktor ohne Argumente, der bewirkt, dass die Warteschlange theoretisch unendlich groß ist (praktisch ist die Größe der Warteschlange natürlich aufgrund des beschränkten Speicherplatzes begrenzt). Dadurch blockiert die Methode *put* zum Einfügen eines Elements theoretisch nie.

*SynchronousQueue* ist eine Implementierung einer Warteschlange, bei der keine Warteschlange vorhanden ist. Damit ist gemeint, dass die Warteschlange 0 Pufferplätze besitzt. Bildlich gesprochen müssen sich der gebende und der nehmende Thread an der Warteschlange wie bei einem Exchanger treffen; die Objekte werden ohne Pufferung direkt vom gebenden an den nehmenden Thread geleitet. Im Unterschied zum Exchanger ist der Datenfluss hier asymmetrisch (d. h. unidirektional).

Mit der Klasse *DelayQueue* können nur solche Elemente in die Warteschlange eingefügt werden, welche die Schnittstelle *Delayed* implementieren. Die Schnittstelle *Delayed* hat eine einzige Methode namens *getDelay*, mit der die aktuelle Verzögerungszeit abgefragt werden kann. Wird ein solches *Delayed*-Element in die Warteschlange eingefügt, so kann es erst dann entnommen werden, wenn die Verzögerungszeit abgelaufen ist. Dies ist dann der Fall, wenn die Methode *getDelay* 0 oder einen negativen Wert zurückgibt. Am Kopf der Warteschlange steht das Element, dessen Verzögerungszeit zuerst abgelaufen ist.

Die Klasse *PriorityBlockingQueue* ist schließlich die letzte hier vorgestellte Implementierung der Schnittstelle *BlockingQueue*. Bei einer *PriorityBlockingQueue* ist die Größe der Warteschlange theoretisch unbegrenzt. Ein Element wird beim Einfügen mit den bereits in der Warteschlange sich befindlichen Elementen verglichen; dazu müssen entweder die Elemente vergleichbar sein (d. h. die Schnittstelle *Comparable* implementieren), oder im Konstruktor von *PriorityBlockingQueue* wird ein *Comparator* übergeben, der je zwei Elemente vergleichen kann. Die Elemente werden aufsteigend sortiert in die Warteschlange einge-reiht. Das heißt: Das gemäß diesem Vergleich „kleinste“ Element befindet sich am Kopf der Warteschlange und wird als erstes entnommen.

## ■ 3.8 Das Fork-Join-Framework von Java 7

Die im letzten Abschnitt beschriebene Concurrent-Bibliothek wurde in Java 7 um das *Fork-Join-Framework* erweitert. Eine der zentralen Klassen dieses Frameworks ist *ForkJoinPool*, eine Klasse, die wie *ThreadPoolExecutor* (s. Abschnitt 3.7.1) einen Thread-Pool realisiert und die Schnittstelle *ExecutorService* implementiert. Dieser Thread-Pool ist speziell für baumartige Berechnungen gedacht, so wie wir sie in Abschnitt 2.4.5 kennen gelernt haben. Dort haben wir für jeden Knoten des Berechnungsbaums einen eigenen Thread erzeugt. Bei größeren Berechnungen, die große Bäume mit vielen Knoten zur Folge haben, werden dadurch jedoch unnötig viele Threads erzeugt, was einen großen Overhead verursacht, aber keinen Gewinn durch zusätzliche Parallelität bringt. Deshalb ist eine Realisierung mit Hilfe eines Thread-Pools zu bevorzugen.

### 3.8.1 Grenzen von *ThreadPoolExecutor*

Um zu verstehen, warum die schon existierende Realisierung eines Thread-Pools in *ThreadPoolExecutor* für rekursive Berechnungen ungeeignet ist, betrachten wir das folgende Beispiel, in dem die Fakultätsfunktion nach der rekursiven Vorschrift berechnet wird:

fakultät(n) = n \* fakultät(n-1) für n > 0 (fakultät(0) = 1). Selbstverständlich ist eine solch rekursive Berechnung durch unterschiedliche Threads für diesen Fall nicht sinnvoll, aber die Grenzen von ThreadPoolExecutor werden dadurch deutlich sichtbar (s. Listing 3.24).

### Listing 3.24

```

import java.util.*;
import java.util.concurrent.*;

class RecursiveFacCallable implements Callable<Integer>
{
    private int n;
    private ThreadPoolExecutor pool;

    public RecursiveFacCallable(int n, ThreadPoolExecutor pool)
    {
        this.n = n;
        this.pool = pool;
    }

    public Integer call()
    {
        System.out.println("ausgeführt von " +
                           Thread.currentThread().getName());
        if(n == 0)
        {
            return 1;
        }
        RecursiveFacCallable rcs = new RecursiveFacCallable(n-1,
                                                              pool);
        List<Callable<Integer>> list = new ArrayList<>();
        list.add(rcs);
        try
        {
            int r = pool.invokeAny(list);
            return n * r;
        }
        catch(Exception e)
        {
            System.out.println(e.getMessage());
            return 0;
        }
    }
}

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

        ThreadPoolExecutor pool = new ThreadPoolExecutor(5, 5,
                                                          0L, TimeUnit.SECONDS,
                                                          new SynchronousQueue<Runnable>());

        RecursiveFacCallable c = new RecursiveFacCallable(5, pool);
        ArrayList<RecursiveFacCallable> reqs = new ArrayList<>();
        reqs.add(c);
        try
    }
}

```

```

    {
        int result = pool.invokeAny(reqs);
        System.out.println("Ergebnis: " + result);
    }
    catch(Exception e)
    {
        System.out.println(e.getMessage());
    }
    pool.shutdown();
}
}

```

Das Entscheidende in Listing 3.24 ist, dass in der Methode call, welche von einem Thread des Pools ausgeführt wird, ein weiterer Auftrag an den Thread-Pool abgesetzt und auf dessen Ende gewartet wird (der Thread-Pool ist aus diesem Grund ein Attribut der Klasse, welche Callable implementiert). Da also der auftraggebende Thread seine Auftragsbearbeitung während des Wartens noch nicht abgeschlossen hat, kommt für die Bearbeitung des Unterauftrags nur ein anderer Thread des Pools in Frage. Das zeigt die Ausgabe von Listing 3.24. Die Aufrufe der Methode call werden alle von unterschiedlichen Threads ausgeführt. Solange die Fakultät von 0, 1, 2, 3 oder 4 berechnet wird, geht auch alles gut. Soll aber die Fakultät von 5 oder mehr berechnet werden, erzeugt das obige Programm eine Ausnahme. Ursächlich für dieses Verhalten ist der Pool mit fünf Threads und einer SynchronousQueue ohne Pufferplätze. Nachdem nämlich fünf Aufträge an den Pool geschickt wurden, von dem noch keiner abgeschlossen wurde und somit alle fünf Threads beschäftigt sind, führt der sechste Auftrag zu einer Ausnahme („Task rejected from ThreadPoolExecutor“). Wichtig ist hierbei, dass eine SynchronousQueue keine Aufträge speichern kann und ab der fünften Fakultät mindestens sechs Aufträge (einschließlich der 0) nötig sind.

### 3.8.2 ForkJoinPool und RecursiveTask

Im Folgenden wollen wir die Fakultätsberechnung nun zum Vergleich mit einem ForkJoinPool durchführen. Die Klasse *ForkJoinPool* hat u. a. einen Konstruktor mit einem Argument des Typs int, womit man den Parallelitätsgrad angeben kann (verwendet man den parameterlosen Konstruktor, so ist der Parallelitätsgrad gleich der Anzahl der Prozessoren). Aufträge für einen ForkJoinPool werden durch Objekte der generischen Klasse *ForkJoinTask* repräsentiert, wobei der Typparameter der Typ des Resultats ist. Aus *ForkJoinTask* abgeleitet sind *RecursiveAction* und *RecursiveTask*, die einfacher zu benutzen sind. *Recursive-Action* ist für Aufträge ohne Resultate gedacht (das kommt z. B. beim Sortieren vor, wenn die Elemente im übergebenen Feld getauscht werden und somit am Ende das Ergebnis im zuvor als Auftrag übergebenen Feld zu finden ist; *RecursiveAction* ist aus *ForkJoinTask<Void>* abgeleitet). *RecursiveTask* dagegen ist für Aufträge mit Ergebnis vorgesehen; wie *ForkJoinTask* ist *RecursiveTask* eine generische Klasse, wobei auch in diesem Fall der Typparameter für den Ergebnistyp steht.

Um einen Auftrag mit Ergebnis zu programmieren, leitet man eine eigene Klasse beispielsweise aus *RecursiveTask* ab und überschreibt darin die Methode *compute* (keine Parameter, Rückgabetyp ist der Typparameter der Klasse; dies entspricht der Methode *call* der Schnittstelle *Callable*). In *compute* entscheidet man in der Regel, ob der Auftrag direkt ausgeführt

werden kann, weil er klein genug ist, oder ob er weiter delegiert werden soll. Zum Delegieren kann man ein oder mehrere Objekte seiner aus RecursiveTask abgeleiteten Klasse erzeugen. Durch Anwendung der von ForkJoinTask geerbten Methode fork wird dieser Auftrag dem Thread-Pool zur Auftragsbearbeitung übergeben. Der Thread, welche die Methode compute ausführt, weiß, zu welchem Thread-Pool er gehört, so dass die Angabe des Thread-Pools nicht nötig ist. Im Gegensatz zu Listing 3.24 benötigt deshalb im folgenden Programm (s. Listing 3.25) die Klasse RecursiveFacTask, welche den Auftrag zur Berechnung der Fakultät repräsentiert, keine Referenz auf den Thread-Pool. Neben der Methode fork ist die zweite wichtige Methode, die ebenfalls über RecursiveTask von ForkJoinTask vererbt wird, die Methode join. Mit join wartet man auf das Ende der Berechnung des Auftrags, auf den man die Methode anwendet. Der Rückgabewert von join ist das berechnete Ergebnis. Damit können wir Listing 3.25 betrachten, in dem die Fakultätsfunktion in rekursiver Weise mit einem ForkJoinPool berechnet wird.

**Listing 3.25**

```
import java.util.concurrent.*;  
  
class RecursiveFacTask extends RecursiveTask<Integer>  
{  
    private int n;  
  
    public RecursiveFacTask(int n)  
    {  
        this.n = n;  
    }  
  
    public Integer compute()  
    {  
        System.out.println("compute(" + n + "): ausgeführt von " +  
                           Thread.currentThread().getName());  
        if(n == 0)  
        {  
            return 1;  
        }  
        RecursiveFacTask rat = new RecursiveFacTask(n-1);  
        rat.fork();  
        int r = rat.join();  
        return n*r;  
    }  
}  
  
public class FacForkJoin  
{  
    public static void main(String[] args)  
    {  
        ForkJoinPool pool = new ForkJoinPool(5);  
        RecursiveFacTask task = new RecursiveFacTask(10);  
        int result = pool.invoke(task);  
        System.out.println("Ergebnis: " + result);  
    }  
}
```

Bei der Ausführung des Programms von Listing 3.25 sieht man, dass dieses Mal die Fakultätsberechnung auch für  $n > 4$  (z.B. für  $n = 10$ ) funktioniert, obwohl auch in diesem Fall der Parallelitätsgrad nur fünf ist. Die Ausgabe des Programms zeigt, dass die Anzahl der involvierten Threads zur Bearbeitung der Aufträge bei wiederholter Ausführung unterschiedlich ist: Manchmal kann man sehen, dass ein einziger Thread alle Aufträge bearbeitet. Bei weiteren Probeläufen können auch zwei oder drei Threads beteiligt sein. Offenbar kann also ein Thread eines ForkJoinPools weitere Aufträge bearbeiten, sobald er mit join auf das Ende eines anderen Auftrags wartet. Dies ist einer der wesentlichen Unterschiede zwischen einem ForkJoinPool und einem ThreadPoolExecutor.

Dieses Programm terminiert übrigens auch ohne einen Aufruf von shutdown, da alle Threads des ForkJoinPools Hintergrund-Threads (Daemon Threads, s. Abschnitt 2.9) sind.

### 3.8.3 Beispiel zur Nutzung des Fork-Join-Frameworks

Das vorhergehende Beispiel aus Listing 3.25 war eine reine Demonstrationsanwendung, bei der die Aufträge in einer linearen Kette angeordnet sind und somit keinerlei Parallelität möglich ist, wobei das Fork-Join-Framework manchmal schlau genug ist, dies zu erkennen und alle Aufträge von einem einzigen Thread ausführen lässt. Im folgenden Programm wollen wir nun mit Hilfe des Fork-Join-Frameworks statt einer linearen Kette einen Baum von Aufträgen realisieren, so dass tatsächlich Parallelität möglich wird. Als Beispiel verwenden wir die in Abschnitt 2.4 schon mehrfach strapazierte Aufgabe des Zählens der True-Werte in einem booleschen Feld. Der Programmcode in Listing 3.26 ist dem aus Listing 3.25 sehr ähnlich. Der wesentliche Unterschied besteht darin, dass für den Fall, dass der Auftrag nicht selbst bearbeitet werden soll, mehrere Unteraufträge (SPLIT\_FACTOR viele) statt eines einzigen erzeugt und mit fork dem Pool zur weiteren Bearbeitung übergeben werden. Das Ergebnis ist die Summe der Ergebnisse der Teilaufträge. Insgesamt ergibt sich damit wieder die Struktur aus Bild 2.3.

**Listing 3.26**

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

class TrueCountingTask extends RecursiveTask<Integer>
{
    private static final int SEQUENTIAL_THRESHOLD = 100;
    private static final int SPLIT_FACTOR = 5;

    private boolean[] array;
    private int start;
    private int end;

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

    protected Integer compute()
```

```
{  
    if(end - start + 1 <= SEQUENTIAL_THRESHOLD)  
    {  
        int result = 0;  
        for(int i = start; i <= end; i++)  
        {  
            if(array[i])  
            {  
                result++;  
            }  
        }  
        return result;  
    }  
    else  
    {  
        int s = start;  
        int e;  
        int howMany = (end-start+1) / SPLIT_FACTOR;  
  
        TrueCountingTask[] tasks =  
            new TrueCountingTask[SPLIT_FACTOR];  
        for(int i = 0; i < SPLIT_FACTOR; i++)  
        {  
            if(i < SPLIT_FACTOR-1)  
            {  
                e = s + howMany - 1;  
            }  
            else  
            {  
                e = end;  
            }  
            tasks[i] = new TrueCountingTask(array, s, e);  
            tasks[i].fork();  
            s = e + 1;  
        }  
        int result = 0;  
        for(int i = 0; i < SPLIT_FACTOR; i++)  
        {  
            result += tasks[i].join();  
        }  
        return result;  
    }  
}  
}  
  
public class AsynchRequestForkJoin  
{  
    private static final int ARRAY_SIZE = 100000;  
  
    public static void main(String[] args)  
    {  
        boolean[] array = new boolean[ARRAY_SIZE];  
        for(int i = 0; i < ARRAY_SIZE; i++)  
        {  
            array[i] = true;  
        }  
  
        // Startzeit messen
```

```
long startTime = System.currentTimeMillis();

// Ergebnis berechnen
ForkJoinPool pool = new ForkJoinPool(5);
TrueCountingTask task = new TrueCountingTask(array, 0,
                                              array.length-1);
int result = pool.invoke(task);

// Endzeit messen
long endTime = System.currentTimeMillis();
float time = (endTime - startTime) / 1000.0f;
System.out.println("Rechenzeit: " + time);

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

Wenn man in dem Programm von Listing 3.26 ähnliche Augaben wie im Programm aus Listing 3.25 einbaut, die den Namen des ausführenden Threads ausgeben, dann erkennt man, dass wesentlich mehr Threads an der Ausführung beteiligt sind, als dies im Konstruktor durch den Parallelitätsgrad vorgegeben wurde. Offensichtlich sind die Threads im Fork-JoinPool nicht immer dieselben, sondern werden im Laufe der Zeit ausgetauscht.

In vielen Beispielprogrammen zum Fork-Join-Framework, die Sie an anderer Stelle finden können, wird eine gewisse Optimierung vorgenommen, auf die wir an dieser Stelle der Einfachheit halber verzichtet haben, die aber leicht in Listing 3.26 einzubauen wäre: Wenn in der Methode compute einer Task (im Beispiel in der Klasse TrueCountingTask) N Unteraufträge erzeugt werden (im Beispiel in Form von Objekten der Klasse TrueCountingTask), dann wird nur auf N-1 Aufträge fork und join angewendet. Zwischen den N-1 Fork-Aufrufen und den N-1 Join-Aufrufen wird auf den verbleibenden Auftrag direkt compute angewendet. Diesem Vorgehen liegt folgende Überlegung zugrunde: Der Thread, der den aktuellen Auftrag ausführt, wartet nach den Fork-Aufrufen mit join auf das Ende dieser Aufträge. Während dieses Wartens wird dieser Thread mindestens einen, eventuell sogar mehrere der soeben erstellen Unteraufträge selbst bearbeiten. Somit ist es für mindestens einen Unterauftrag nicht notwendig, diesen mit fork dem ForkJoinPool zu übergeben, sondern der Thread kann diesen Auftrag einfacher durch einen direkten Aufruf der Methode compute bearbeiten.

Es soll abschließend noch erwähnt werden, dass sich die Konzepte des Fork-Join-Frameworks und von Intels *Threading Building Blocks (TBB)* sehr stark ähneln. Wenn Sie eines der Konzepte kennen, werden Sie sich mühelos in das andere einarbeiten können.

## ■ 3.9 Das Data-Streaming-Framework von Java 8

In Java 8 gab es eine wesentliche Neuerung, die aus dem Big-Data-Bereich inspiriert wurde, nämlich ein Framework zur Verarbeitung von Datenströmen (bitte nicht verwechseln mit den Ein- und Ausgabeströmen, die in den Abschnitten 5.5 und 6.4 besprochen werden). Im Wesentlichen geht es dabei darum, einen Datenstrom wie in einem Fließband mit unterschiedlichen Stationen zu verarbeiten. Die Daten werden aus einer Quelle (zum Beispiel einer Liste) in das Fließband eingespeist und in den Zwischenstationen bearbeitet (transformiert, gefiltert usw.). Am Ende wird die Bearbeitung der Daten durch eine finale Operation abgeschlossen (z.B. werden alle Daten ausgegeben oder, falls es sich bei den Daten um Zahlen handelt, könnte auch die Summe aller Daten gebildet werden). Das Thema Parallelität kommt dabei für die Entwicklerin in kaum wahrnehmbarer Weise vor. Bevor wir uns aber mit parallelen Datenströmen beschäftigen, beginnen wir unsere Betrachtung mit der Verarbeitung sequenzieller Datenströme anhand eines einleitenden Beispiels.

### 3.9.1 Einleitendes Beispiel

Wir gehen in einem einleitenden Beispiel zum Data-Streaming (der Java-API-Dokumentation entnommen) von einer Klasse Item aus, die Gegenstände repräsentiert, die ein Gewicht und eine Farbe haben (s. Listing 3.27).

**Listing 3.27**

```
class Item
{
    public enum Color
    {
        RED, GREEN, BLUE, YELLOW, BLACK, WHITE
    }

    private int weight;
    private Color color;

    public Item(int weight, Color color)
    {
        this.weight = weight;
        this.color = color;
    }

    //Getter- und Setter-Methoden für beide Attribute:
    ...
}
```

Wenn sich nun mehrere Item-Objekte in einer Liste befinden, dann sollte es keine große Herausforderung sein, beispielsweise das Gesamtgewicht aller roten Gegenstände in der Liste zu berechnen. Wie Listing 3.28 zeigt, kann man mit dem Data-Streaming-Framework diese Berechnung aber deutlich kompakter formulieren.

**Listing 3.28**

```

import java.util.*;

public class StreamExample
{
    private static List<Item> createSampleList(int size)
    {
        ...
    }

    public static void main(String[] args)
    {
        List<Item> itemsList = createSampleList(1000);
        int sum = itemsList.stream().
                    filter(item -> item.getColor() ==
                               Item.Color.RED).
                    mapToInt(item -> item.getWeight()).
                    sum();
        System.out.println("Sum of weight of all red items: " + sum);
    }
}

```

Zunächst wird eine Item-Liste der Größe 1000 erzeugt (wie das geschieht, soll hier nicht näher interessieren). Dann wird auf die Liste die Methode *stream* angewendet. Auch wenn das nicht so ist, wie später noch erläutert wird, so kann man sich doch vorstellen, dass damit das Einspeisen der Listenelemente in ein Fließband angestoßen wird. Als erste Verarbeitungsstation des Fließbands wird ein Filter definiert, der nur rote Gegenstände durchlässt. Bei der zweiten Station wird jedes Item-Element in einen Int-Wert transformiert, der seinem Gewicht entspricht. Das heißt, ab dieser Station fließen statt Item-Elemente Zahlen durch das restliche Fließband. In unserem Fall wird das Fließband mit der nächsten Station, in der alle ankommenden Zahlen aufsummiert werden, beendet. Diese Summe ist der Rückgabewert der gesamten Streaming-Anweisung. Die Summe wird dann abschließend ausgegeben.

In solchen Streaming-Anweisungen ist die Verwendung von Lambda-Ausdrücken typisch. Dies ist natürlich nur deshalb möglich, weil viele der darin verwendeten Methoden Parameter besitzen, deren Typ eine funktionale Schnittstelle ist. So hat in unserem obigen Beispiel die Methode *filter* einen Parameter des Typs *Predicate*. *Predicate* ist eine generische Schnittstelle mit mehreren Methoden. Diese sind aber alle Default-Methoden bis auf eine. *Predicate* hat also nur eine einzige abstrakte Methode und ist somit eine funktionale Schnittstelle. Die abstrakte Methode von *Predicate* heißt *test* (dieser Name kommt im Lambda-Ausdruck nicht vor), besitzt einen Parameter vom Typ des generischen Typparameters und den Rückgabetyp *boolean*. Im Lambda-Ausdruck des Aufrufs der Methode *filter* lautet der Parameter *item* (vom Typ *Item*). Der Teil hinter dem Pfeil ist ein Ausdruck des Typs *boolean*, der von der durch diesen Lambda-Ausdruck definierten Methode zurückgegeben wird. Ähnlich verhält es sich mit der Methode *mapToInt*. Der Typ des Parameters ist *ToIntFunction*. *ToIntFunction* ist eine generische Schnittstelle mit einer einzigen abstrakten Methode namens *apply* (dieser Name ist wiederum unwichtig für den Lambda-Ausdruck). Auch diese Methode besitzt einen Parameter vom Typ des generischen Typparameters. Der Rückgabetyp ist in diesem Fall aber *int*. Im Lambda-Ausdruck steht folgerichtig rechts vom Pfeil ein Ausdruck des Typs *int*.

Der Aufruf der Methode `stream` auf eine Liste liefert etwas zurück vom Typ `Stream`. Stream ist eine generische Schnittstelle. Der Typparameter von Stream ist derselbe Typ wie der der Liste. In unserem Fall wenden wir `stream` auf `List<Item>` an. Also ist `Stream<Item>` der Rückgabetyp beim Aufruf von `stream`. Die Schnittstelle Stream besitzt u.a. eine Methode `filter`, die so definiert ist (`T` ist dabei der Typparameter der Schnittstelle Stream):

```
public Stream<T> filter(Predicate<? super T> predicate);
```

Da `filter` eine Methode der Schnittstelle `Stream<T>` ist, wird `filter` auf etwas angewendet vom Typ `Stream<T>`, liefert aber wieder etwas vom Typ `Stream<T>` zurück. Dadurch ist die Verkettung der Methodenaufrufe möglich: Auf das, was von `filter` zurückkommt, kann `mapToInt` angewendet werden, eine weitere Methode der Schnittstelle Stream. Man sieht, dass der Parameter von `filter` den Typ `Predicate` besitzt. `Predicate` ist ebenfalls generisch. Es wird nun nicht verlangt, dass man hier ein `Predicate` benötigt, dessen Typparameter genau `T` ist (in unserem Beispiel wäre das `Item`), sondern durch den Super-Ausdruck mit dem Fragezeichen ist es möglich, dass allgemeinere `Predicate`-Ausdrücke verwendet werden, nämlich solche, deren Parameter eine Oberklasse von `T` ist. In unserem `Item`-Beispiel kommt als Oberklasse nur `Object` in Frage. Somit könnte man ein `Predicate` definieren, das jedem beliebigen Objekt zufällig einen Wahrheitswert zuordnet. Dieses `Predicate` ist dann als Filter-Ausdruck in einem `Stream<Item>` auch möglich, was nicht so wäre, wenn als Parameter von `filter` nur `Predicate<T>` erlaubt wäre statt `Predicate<? super T>`. Zur Klarstellung hier das Ganze noch als Java-Code:

```
Predicate<Object> pred = obj -> Math.random() < 0.5 ? true : false;
List<Item> itemsList = createSampleList(1000);
int sum = itemsList.stream().
    filter(pred).
    mapToInt(item -> item.getWeight()).
    sum();
```

Neben dem allgemeinen `Stream<T>` gibt es noch die Schnittstellen `IntStream`, `LongStream` und `DoubleStream`, die Streams der primitiven Datentypen `int`, `long` und `double` repräsentieren. Diese Schnittstellen haben u.a. Methoden wie z.B. `sum` und `average`, die in `Stream<T>` nicht vorhanden sind, da von einem Strom irgendwelcher Objekte keine Summe und kein Durchschnitt berechnet werden kann. Die Methode `mapToInt` der Schnittstelle `Stream<T>` liefert einen `IntStream` zurück:

```
public IntStream mapToInt(ToIntFunction<? super T> mapper);
```

### 3.9.2 Sequenzielles Data-Streaming

Nach diesem einleitenden Beispiel wollen wir uns etwas allgemeiner mit dem sequenziellen Data-Streaming beschäftigen. Grundsätzlich gibt es drei Arten von Methoden: Methoden zur Erzeugung von Streams (im Beispiel `stream`), intermediäre Methoden zur Definition von Verarbeitungsschritten innerhalb des Fließbands (im Beispiel `filter` und `mapToInt`) und terminale Methoden zum Abschließen des Fließbands (im Beispiel `sum`). Im Folgenden schauen wir uns diese drei Arten von Methoden noch etwas genauer an:

## Methoden zur Erzeugung von Streams

Viele Klassen des Collection-Frameworks (u.a. verschiedene Varianten von List, Set und Queue) besitzen die Methode stream, die wir auf unsere Liste angewendet haben. Daneben gibt es noch eine ganze Reihe weiterer Möglichkeiten zur Erzeugung von Streams. So kann man beispielsweise durch unterschiedliche statische Methoden der Klasse Arrays aus Feldern (Arrays) Streams erzeugen:

```
int[] array = {7, 9, 5, 2};
IntStream istream = Arrays.stream(array);
```

Die aus Listen und Feldern generierten Streams sind offensichtlich endlich. Es ist aber auch möglich, unendliche Ströme zu erzeugen. Dafür gibt es in allen vier Stream-Schnittstellen jeweils eine statische Methode namens *generate* (hier für das Beispiel DoubleStream gezeigt):

```
public static DoubleStream generate(DoubleSupplier supplier);
```

DoubleSupplier ist eine funktionale Schnittstelle, die eine parameterlose Methode mit Rückgabetyp double enthält. Somit könnte man also beispielsweise folgenden unendlichen Strom aus Zufallszahlen erzeugen:

```
DoubleStream dstream = DoubleStream.generate(() -> Math.random());
```

Mit Hilfe der statischen Methode *iterate* kann man ebenfalls einen unendlichen Strom erzeugen. Hierzu muss man ein Startelement angeben sowie eine Vorschrift, wie man aus einem Element des Stroms das nächste Element berechnet. Wir wollen das Prinzip wieder anhand von DoubleStream erläutern. Die Methode iterate der Schnittstelle DoubleStream hat diese Signatur:

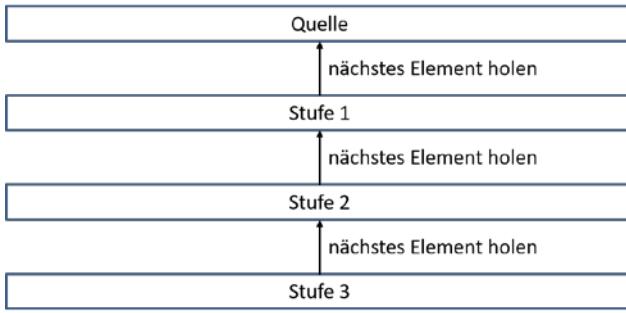
```
public static DoubleStream iterate(double seed, DoubleUnaryOperator op);
```

Auch DoubleUnaryOperator ist wieder eine funktionale Schnittstelle mit einer abstrakten Methode, die einen Parameter des Typs double erwartet und einen Wert des Typs double zurückliefert. Damit könnte auf diese Art ein unendlicher DoubleStream erzeugt werden mit den Elementen 10,0, 10,2, 10,4, 10,6 usw.:

```
DoubleStream dstream = DoubleStream.iterate(10, d -> d+0.2);
```

Wie für generate gibt es auch in den anderen Stream-Schnittstellen Stream<T>, IntStream und LongStream entsprechende Varianten für iterate.

Übrigens bewirken die oben gezeigten Aufrufe von generate und iterate vielleicht wider Erwarten keine unendliche Laufzeit, sondern kehren nach kurzer Zeit zurück. Dies liegt daran, dass alle Stream erzeugenden Operationen nicht aktiv Elemente in ein Fließband einspeisen, sondern dass eine Fließbandverarbeitung erst dann erfolgt, wenn eine terminale Operation aufgerufen wird. Es ist also so, dass über die terminale Operation die Elemente von der jeweils vorhergehenden Station des Fließbands aktiv geholt werden (Pull-Prinzip, s. Bild 3.6) und nicht die vorhergehende Station die bearbeiteten Daten aktiv zur nachfolgenden Station weiterleitet (Push-Prinzip).



**Bild 3.6**  
Pull-Prinzip der Fließbandverarbeitung

Dadurch passiert durch die gezeigten Aufrufe von generate und iterate erst einmal gar nichts. Ein darauffolgender Aufruf von sum (terminale Operation) würde dann allerdings nicht zurückkehren, sondern in einer Endlosschleife hängen bleiben:

```
double sum = dstream.sum();
```

Man kann sich auch für die Dateien in einem Teilbaum des Dateisystems oder einer Zip-Datei entsprechende Streams erzeugen lassen, um diese Dateien dann einer Fließbandverarbeitung zuzuführen.

### Intermediäre Methoden zur Definition von Verarbeitungsschritten

Als Beispiel für eine intermediäre Methode haben wir schon die Methode filter kennengelernt. Eine speziellere Form des Filterns erzielt man mit der Methode *limit* (im Folgenden stellvertretend für Stream<T> gezeigt):

```
public Stream<T> limit(long maxSize);
```

Von einem endlichen oder unendlichen Strom werden höchstens so viele Elemente durchgelassen, wie durch den Parameter vorgegeben wird. Danach gilt der Stream als beendet. Die zuvor über generate oder iterate erzeugten unendlichen Ströme könnte man so auf 1.000 Elemente begrenzen:

```
double sum = dstream.limit(1000).sum();
```

In diesem Fall terminiert diese Anweisung.

Mit mapToInt haben wir eine Transformationsmethode kennengelernt. Auch hier gibt es weitere Mapping-Funktionen, u. a. eine, die einen Stream<X> in einen Stream<Y> umwandelt, wobei X und Y beliebige Klassen sind.

Eine weitere wichtige intermediäre Methode ist *peek* (auch hier wieder für Stream<T> gezeigt):

```
public Stream<T> peek(Consumer<? super T> action);
```

*Consumer* ist eine generische, funktionale Schnittstelle mit einer Void-Methode namens accept, die einen Parameter des generischen Typparameters besitzt. Man könnte mit peek beispielsweise bei den durchlaufenden Item-Objekten deren Gewicht verdoppeln:

```
List<Item> itemsList = createSampleList(1000);
int sum = itemsList.stream().
    filter(item -> item.getColor() == Item.Color.RED).
    peek(item -> item.setWeight(item.getWeight()*2)).
    mapToInt(item -> item.getWeight()).
    sum();
```

Die Methode `peek` kann auch gut zur Fehlersuche verwendet werden. Wenn wir davon ausgehen, dass die Klasse `Item` eine passende Methode `toString` hat, dann könnte man sich durch zwei weitere Aufrufe von `peek` beispielsweise den Strom von `Item`-Elementen am Anfang des Fließbands und den Strom von Int-Werten am Ende des Fließbands ausgeben lassen:

```
List<Item> itemsList = createSampleList(1000);
int sum = itemsList.stream().
    peek(item -> System.out.println(item)).
    filter(item -> item.getColor() == Item.Color.RED).
    peek(item -> item.setWeight(item.getWeight()*2)).
    mapToInt(item -> item.getWeight()).
    peek(i -> System.out.println(i)).
    sum();
```

Die Ausgabe erfolgt elementweise, nicht stationsweise. Das heißt, das erste `Item`-Element wird ausgegeben und im Fall, dass es rot ist, folgt direkt im Anschluss der Wert seines doppelten Gewichts. Dann erfolgt die Ausgabe für das zweite Element usw.

## Terminale Methoden

Wie schon erwähnt wurde, muss ein Streaming durch eine terminale Operation abgeschlossen werden, andernfalls passiert gar nichts. Die terminalen Operationen lassen sich unterteilen in Traversierungsoperationen, Suchoperationen, Reduzieroperationen und Sammelergebnisse. Zu den Traversierungsoperationen gehört zum Beispiel die Methode `forEach`. Sie ist sehr ähnlich wie `peek` (hat auch einen Consumer-Parameter), ist aber im Unterschied zu `peek` terminal und deshalb auch void (das heißt, man kann nach `forEach` die Operationskette nicht weiter fortsetzen). Als Beispiele für Suchoperationen seien die Methoden `allMatch` und `anyMatch` genannt (beide mit einem Predicate-Argument und Rückgabetyp `boolean`). Über das Predicate-Argument (s. Methode `filter`) lässt sich eine Bedingung definieren, die im Fall von `allMatch` alle der am Ende des Fließbands an kommenden Elemente erfüllen müssen, damit `true` zurückgegeben wird. Im Fall von `anyMatch` genügt es, wenn mindestens eines der an kommenden Elemente die vorgegebene Bedingung erfüllt, um `true` zu erhalten. Zu den Reduzieroperationen gehört beispielsweise die Methode `count`, die einfach alle an kommenden Elemente zählt, oder für `IntStream`, `LongStream` und `DoubleStream` auch die Methoden `sum` und `average`. Durch die allgemeine Methode `reduce` ist man in der Lage, sich seine eigene Form der Reduktion selbst zu definieren, indem man einen Startwert für den zu errechneten Reduktionswert angibt sowie eine Vorschrift, wie aus dem bisher errechnenden Wert und dem neu eintreffenden Wert der Reduktionswert aktualisiert wird. Die Methode `sum` könnte man dann auf diese Weise selber so programmieren:

```
...reduce(0, (a,b) -> a+b);
```

Mit Sammeloperationen kann man die ankommenden Elemente in einer Datenstruktur zusammenfassen. So gibt es beispielsweise die terminale Operation `toArray`, die – wie der Name sagt – alle angekommenen Elemente in einem Feld (Array) zurückliefert. Ähnlich wie bei den Reduzieroperationen gibt es auch bei den Sammeloperationen eine allgemeine Form einer Methode. Diese heißt in diesem Fall `collect` und ermöglicht selbst festzulegen, in welcher Weise die ankommenden Elemente in einer Datenstruktur zusammengefasst werden.

### 3.9.3 Paralleles Data-Streaming

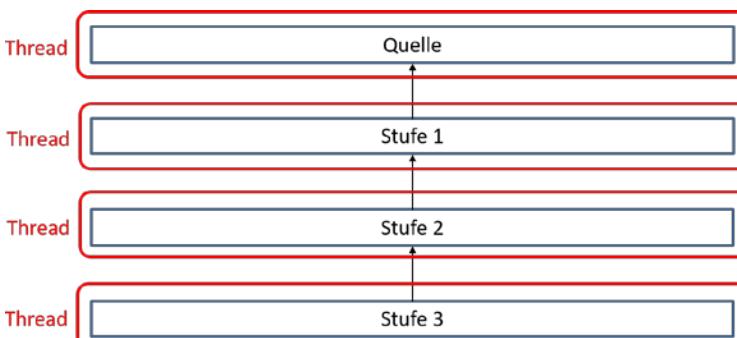
Äußerst interessant und vielleicht verblüffend ist, wie einfach aus Sicht eines Entwicklers der Übergang vom sequenziellen zum parallelen Data-Streaming ist. Hat man einen Strom, so kann man einfach durch Aufruf von `parallel` dafür die Parallelverarbeitung anstoßen:

```
List<Item> itemsList = createSampleList(1000);
int sum = itemsList.stream().parallel().
    mapToInt(item -> item.getWeight()).sum();
```

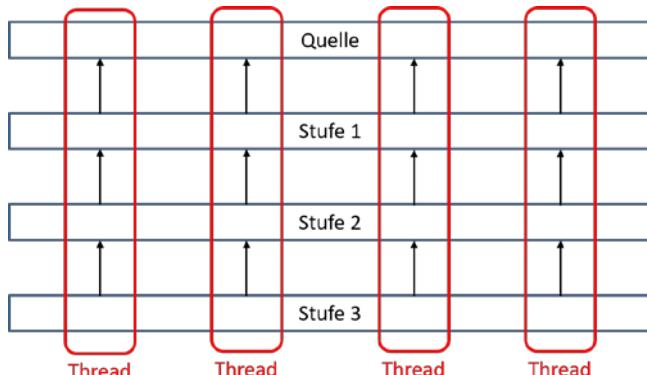
Für den Fall, dass man von einer Liste startet, gibt es alternativ auch die Möglichkeit, statt `stream` die Methode `parallelStream` anzuwenden:

```
List<Item> itemsList = createSampleList(1000);
int sum = itemsList.parallelStream().
    mapToInt(item -> item.getWeight()).sum();
```

Die Programmiererin muss sich (und kann sich) dabei nicht darum kümmern, wie viele Threads gestartet und welche Aufträge auf welche Threads verteilt werden. Auch muss man sich nicht um die Synchronisation sorgen. Das erledigt alles das Framework. Das heißt, wir sehen hier parallele Programmierung auf sehr hoher Abstraktionsebene. Die Parallelisierung erfolgt übrigens nicht stationsweise (s. Bild 3.7), sondern elementweise (s. Bild 3.8).



**Bild 3.7** Horizontale Parallelisierung basierend auf Fließbandstationen



**Bild 3.8** Vertikale Parallelisierung basierend auf den zu bearbeitenden Daten

In der Regel ist die Anzahl der durch das Fließband geschleusten Datenelemente deutlich größer als die Anzahl der Bearbeitungsstationen des Fließbands. Damit ergibt sich ein deutlich größeres Parallelisierungspotential. Allerdings gibt es auch Bearbeitungsschritte im Fließband, die die Parallelisierungsmöglichkeiten deutlich beschränken. So werden beispielsweise durch Einfügen der bislang nicht erwähnten intermediären Operation *sorted* die Elemente von einem unsortierten Strom in einen sortierten Strom transformiert, was bedeutet, dass die Elemente nicht völlig unabhängig voneinander von allen Fließbandstationen bearbeitet werden können. Messungen auf meinem Rechner haben gezeigt, dass in einer Anwendung durch paralleles Streaming die Rechenzeit um ca. 50 % gegenüber sequenziellem Streaming reduziert werden konnte. In einer anderen Anwendung war jedoch die Rechenzeit bei parallelem Streaming bei jeder Ausführung ca. 1,5-mal größer als bei sequenziellem Streaming.

## ■ 3.10 Die Completable Futures von Java 8

Eine weitere Neuerung von Java 8 sind *CompletableFuture*. Im Abschnitt 3.7.1 wurden Thread-Pools behandelt. Dabei haben wir gesehen, dass man einen Auftrag mit *submit* bei einem Thread-Pool abgeben kann und ein sogenanntes Future zurück bekommt, mit dem man den Abschluss der Auftragsbearbeitung abfragen, auf das Ergebnis warten oder auch den eventuell noch laufenden Auftrag abbrechen kann. Future ist eine Schnittstelle. Konkret liefert die Methode *submit* ein Objekt der Klasse *FutureTask* zurück. FutureTask implementiert die Schnittstelle Future, kapselt den dazugehörigen Auftrag und besitzt u.a. die Protected-Methoden *set* und *setException*. Mit beiden Methoden wird angezeigt, dass die Auftragsbearbeitung abgeschlossen wurde, einmal regulär mit der Angabe des Ergebnisses und einmal irregulär mit der Angabe der Ausnahme, die den Abbruch der Bearbeitung bewirkt hat. Beide Methoden werden von den Threads des Thread-Pools benutzt. Da sie protected sind, stehen sie für die Anwendungsentwickler nicht zur Verfügung. CompletableFuture ist nun eine Klasse, die ebenfalls die Schnittstelle Future implementiert und die Methoden *complete* und *completeExceptionally* zum Anzeigen des Endes der Auftragsbear-

beitung besitzt, die nun öffentlich sind. Dies erklärt zwar den Namen CompletableFuture, ist aber nicht das Entscheidende an dieser Klasse, die sehr viele weitere Methoden besitzt. Das Besondere ist vielmehr, dass man mit CompletableFuture eine verkettete Ausführung mehrerer Aufträge in besonders eleganter Weise durchführen kann. Hat man ein erstes CompletableFuture-Objekt für den ersten Auftrag der Verarbeitungskette, dann kann auf dieses eine Methode angewendet werden mit einem zweiten Auftrag als Parameter. Die Ausführung dieses zweiten Auftrags wird begonnen, nachdem der erste Auftrag, der zu dem CompletableFuture-Objekt gehört, auf das man die Methode anwendet, beendet wurde. Dabei ist es möglich, dass der zweite Auftrag das Ergebnis des ersten Auftrags als Eingabeparameter benutzt. Die Methode, mit der man den zweiten Auftrag spezifiziert hat, liefert nun wieder ein CompletableFuture-Objekt zurück, auf das man wieder eine Methode anwenden kann zur Angabe eines weiteren Auftrags. Vom letzten CompletableFuture-Objekt der Auftragskette kann dann das letzte Ergebnis abgeholt werden. Zum Starten der Auftragskette besitzt CompletableFuture eine statische Methode namens *supplyAsync*. Als Parameter übergibt man den ersten Auftrag und erhält ein CompletableFuture-Objekt zurück. CompletableFuture ist wie die Schnittstelle Future generisch. Im Folgenden gehen wir davon aus, dass Class1, Class2 usw. die Ergebnistypen der einzelnen Bearbeitungsschritte sind. Es gibt viele Methoden zum Anstoßen der nächsten Bearbeitung. Wir nutzen im Folgenden die Methode *thenApplyAsync*. Dann kann eine verkettete Auftragsbearbeitung so geschrieben werden:

```
CompletableFuture<Class1> f1 = CompletableFuture.supplyAsync(job1);
CompletableFuture<Class2> f2 = f1.thenApplyAsync(job2);
CompletableFuture<Class3> f3 = f2.thenApplyAsync(job3);
CompletableFuture<Class4> f4 = f3.thenApplyAsync(job4);
CompletableFuture<Class5> f5 = f4.thenApplyAsync(job5);
Class5 result = f5.get();
```

Oder verkürzt:

```
Class5 result = CompletableFuture.supplyAsync(job1).
    thenApplyAsync(job2).
    thenApplyAsync(job3).
    thenApplyAsync(job4).
    thenApplyAsync(job5).
    get();
```

Der Parametertyp der Methode *supplyAsync* ist die generische Schnittstelle *Supplier*. Diese ist eine funktionale Schnittstelle mit der parameterlosen Methode *get*, die ein Objekt des generischen Typs zurückliefert. Mit einem Lambda-Ausdruck könnte man job1 so festlegen, damit das Beispiel oben zumindest syntaktisch korrekt ist:

```
Supplier<Class1> job1 = () -> new Class1();
```

Die Methode *thenApplyAsync* besitzt einen Parameter des Typs *Function*. Selbstverständlich ist *Function* auch eine funktionale Schnittstelle mit zwei generischen Typparametern T und R. Die abstrakte Methode *apply* hat einen Parameter vom Typ des ersten generischen Typparameters T und einen Rückgabetyp vom Typ des zweiten generischen Typparameters R. Somit muss job2 eine Methode haben mit einem Argument des Typs Class1 und einem Rückgabetyp Class2. Nur um den Compiler zufrieden zu stellen, kann job2 so definiert werden:

```
Function<Class1, Class2> job2 = c1 -> new Class2();
```

Die anderen Jobs müssen dann entsprechend definiert werden. Schauen wir uns die Signatur der Methode thenApplyAsync genauer an:

```
public class CompletableFuture<T> implements Future<T>, ...
{
    ...
    public <U> CompletableFuture<U> thenApplyAsync(
        Function<? super T, ? extends U> fn) {...}
}
```

Bitte beachten Sie, dass T der generische Typparameter der Klasse CompletableFuture und U der generische Typparameter der Methode thenApplyAsync ist. Das heißt, die Methode wird auf ein Objekt von CompletableFuture<T> angewendet und liefert ein Objekt von CompletableFuture<U> zurück. Folglich braucht man eine Methode, die ein T-Objekt als Parameter entgegennimmt und ein U-Objekt zurückliefert. Der Typ des Parameters fn ist nun aber so angegeben, dass noch mehr zugelassen ist. Es ist nicht schädlich, wenn fn mit mehr als nur T-Objekten umgehen kann. Deshalb wird der erste generische Typ von Function mit „? super T“ angegeben, was bedeutet, dass auch Elemente von Oberklassen von T zugelassen sind. Genauso wenig schädlich wäre ein Function-Parameter, dessen Rückgabetype spezieller als U ist, der also Objekte eines aus U abgeleiteten Typs zurückgibt. Dies wird durch „? extends U“ als zweitem generischen Typparameter von Function ausgedrückt. Wenn wir davon ausgehen, dass Class2Extended aus Class2 abgeleitet ist, dann könnte deshalb job2 in obigem Beispiel auch so definiert werden und wäre syntaktisch immer noch korrekt:

```
Function<Object, Class2Extended> job2 = o -> new Class2Extended();
```

Die Aufträge werden durch Threads eines Thread-Pools ausgeführt. Da im Beispiel kein Thread-Pool angegeben ist, wird der sogenannte CommonPool verwendet. Das ist der Standard-Thread-Pool des Fork-Join-Frameworks. Die Anzahl der Threads ist im Normalfall gleich der Anzahl der Prozessoren - 1. Da der hier zum Ausprobieren der Programme verwendete Rechner einen 4-Kern-Prozessor besitzt, hat der Pool drei Threads. Alternativ gibt es eine überladene Methode zu thenApplyAsync, die neben dem Auftrag einen weiteren Parameter des Typs Executor zur Angabe eines Thread-Pools besitzt. So kann man bei der Auftragsdefinition also auch andere Thread-Pools einsetzen, für unterschiedliche Aufträge einer Verarbeitungskette sogar unterschiedliche Pools.

Die Aufträge waren bisher so gestaltet, dass sie lediglich syntaktisch korrekt waren. Inhaltlich waren sie wenig sinnvoll und auch nicht für eine Demonstration gut geeignet. Um jetzt zumindest die Demonstrationsfähigkeit zu erhöhen, werden im Folgenden Aufträge mit künstlich durch sleep verlängerten Laufzeiten (zufällig gewählte Werte zwischen 1 und 5 Sekunden) verwendet. Für den generischen Typ wird immer String verwendet. Der String wird durch die Auftragskette gereicht. Jeder Auftrag verlängert den String, indem er sowohl seine Bearbeitungszeit als auch den Namen des ausführenden Threads anhängt. In Listing 3.29 ist das Programm dargestellt.

**Listing 3.29**

```

import java.util.concurrent.*;

public class CompletableFutureDemo
{
    public static String supply()
    {
        return apply("");
    }

    public static String apply(String message)
    {
        int sleepTime = (int)(Math.random() * 4000 + 1000);
        try
        {
            Thread.sleep(sleepTime);
        }
        catch(InterruptedException e)
        {
        }
        String result = message;
        if(result.length() > 0)
        {
            result += " - ";
        }
        result += sleepTime + " [" + Thread.currentThread().getName() + "]";
        return result;
    }

    public static void main(String[] args) throws Exception
    {
        CompletableFuture<String> f1 = CompletableFuture.supplyAsync(
            () -> supply());
        CompletableFuture<String> f2 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f3 = f2.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f4 = f3.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f5 = f4.thenApplyAsync(s -> apply(s));
        System.out.println(f5.get());
    }
}

```

Die Ausgabe könnte (in editierter Form) so aussehen:

```

2869 [ForkJoinPool.commonPool-worker-1]
- 2135 [ForkJoinPool.commonPool-worker-1]
...
- 3347 [ForkJoinPool.commonPool-worker-1]

```

Da `get` eine Ausnahme werfen kann, die der Einfachheit halber nicht gefangen wird, wurde `main` mit „throws Exception“ versehen. Die Klasse `CompletableFuture` bietet mit der Methode `join` eine Alternative zu `get`, die nur Runtime-Ausnahmen werfen kann, die bekanntlich syntaktisch ignoriert werden können.

Da eine lineare Verarbeitungskette definiert wurde, ist bei diesem Beispiel keine Parallelität im Spiel (mit Ausnahme des Main-Threads, der auf das Ende der Bearbeitung aller Aufträge wartet, die von Threads des Thread-Pools ausgeführt werden); die Aufträge werden alle

streng hintereinander ausgeführt. Entsprechend zeigt sich in vielen, wenn auch nicht in allen Beispielläufen, dass alle fünf Aufträge der Verarbeitungskette von demselben Thread des CommonPools verarbeitet werden (wie oben in der Beispieldausgabe zu sehen ist). Es ist nun aber mit CompletableFutures auch möglich, Verzweigungen und Zusammenführungen von Verarbeitungsschritten zu definieren. Die Verzweigung ist mit den bisher besprochenen Konzepten zu realisieren. Auf dasselbe CompletableFuture-Objekt für den ersten Auftrag wird mehrfach die Methode `thenApplyAsync` angewendet, was mehrere CompletableFuture-Objekte liefert. Das Zusammenführen von genau zwei CompletableFutures zu einem CompletableFuture-Objekt ist u. a. durch die Methode `thenCombineAsync` möglich. Auf das erste CompletableFuture-Objekt wird diese Methode angewendet, das zweite CompletableFuture-Objekt wird als erster Parameter übergeben. Der zweite Parameter ist ein weiterer Auftrag, der wieder durch einen Thread des Thread-Pools ausgeführt wird. Dieser Auftrag hat nun aber zwei Eingabeparameter, welches die Ergebnisse der beiden zuvor ausgeführten Aufträge sind. Der Auftrag generiert selbst ein Ergebnis, das über das CompletableFuture-Objekt abgeholt werden kann, das von `thenCombineAsync` zurückgegeben wird. Sollen mehr als zwei Aufträge zusammengeführt werden, dann muss `thenCombineAsync` mehrfach angewendet werden. Diese Methode hat folgende Signatur:

```
public class CompletableFuture<T> implements Future<T>, CompletionStage<T>
{
    ...
    public <U,V> CompletableFuture<V> thenCombineAsync(
        CompletionStage<? extends U> other,
        BiFunction<? super T, ? super U, ? extends V> fn) {...}
}
```

`CompletionStage` ist eine weitere Schnittstelle, die im Wesentlichen die Methoden umfasst, die `CompletableFuture` zusätzlich zu den Methoden der Schnittstelle `Future` implementiert. `BiFunction` ist eine funktionale Schnittstelle für eine zweistellige Funktion. Deshalb hat sie drei generische Typparameter: zwei für die Typen der beiden Argumente und einen für den Rückgabetyp. Die Methode `thenCombineAsync` hat zwei Typparameter `U` und `V`: `U` gehört zum `CompletableFuture`, das als erstes Argument übergeben wird, `V` gehört zum Rückgabetyp der `BiFunction` und damit zum `CompletableFuture`, das die Methode zurückgibt. Die Verwendung von `super` und `extends` sollte nicht erneut erklärt werden müssen.

Im folgenden Beispielprogramm verwenden wir dieselben Aufträge wie zuvor. Der Kombinationsauftrag ist ähnlich wie ein einzelner Auftrag, nur gibt es jetzt zwei Eingabestrings, die mit einer Zusatzinformation miteinander verbunden und als Ergebnis zurückgegeben werden. Als generischer Typparameter kommt auschließlich die Klasse `String` vor. Die Struktur der Auftragsbearbeitung entspricht derjenigen aus [Bild 3.2](#). Nach einem ersten Auftrag werden drei Aufträge parallel ausgeführt. Anschließend werden die drei Ergebnisse kombiniert. Da immer nur jeweils zwei Ergebnisse kombiniert werden können, sind zwei Kombinationsschritte nötig. Das Programm in Listing 3.30 sollte somit ohne weitere Erläuterungen verständlich sein.

### Listing 3.30

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

public class CompletableFutureDemo
```

```

{
    //supply und apply wie zuvor: ...

    public static String combine(String message1, String message2)
    {
        String result = "(" + message1 + "\n" +
                        " | combined " +
                        "[" + Thread.currentThread().getName() + "] | \n" +
                        message2 + ")";
        return result;
    }

    public static void main(String[] args) throws Exception
    {
        CompletableFuture<String> f1 = CompletableFuture.supplyAsync(
            () -> supply());
        CompletableFuture<String> f2 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f3 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f4 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f23 = f2.thenCombineAsync(f3,
            (s2, s3) -> combine(s2, s3));
        CompletableFuture<String> f234 = f23.thenCombineAsync(f4,
            (s23, s4) -> combine(s23, s4));
        CompletableFuture<String> f5 = f234.thenApplyAsync(s -> apply(s));
        System.out.println(f5.get());
    }
}

```

Mit Hilfe der statischen Methode *allOf* der Klasse `CompletableFuture` kann man auf das Vorliegen der Ergebnisse von beliebig vielen `CompletableFuture`-Objekten warten. Eine Kombination der Einzelergebnisse ist in diesem Fall allerdings nicht möglich; die Methode *allOf* liefert ein Objekt des Typs `CompletableFuture<Void>` zurück, was bedeutet, dass die Anwendung von `get` (oder `join`) auf dieses Objekt null zurückgibt, den einzigen möglichen gültigen Wert für `Void`. Man erreicht damit lediglich eine Synchronisation.

Die Zusammenführung in Listing 3.30 war eine UND-Zusammenführung. Das heißt, alle vorhergehenden Aufträge mussten zu Ende sein und alle diese Ergebnisse wurden verwendet. Alternativ gibt es auch ODER-Zusammenführungen, bei denen man nur das Ergebnis verwendet, das als erstes vorliegt. Analog zu `thenCombineAsync` gibt es `applyToEitherAsync`, was das erste Ergebnis von zwei vorausgehenden Aufträgen zur Durchführung des nächsten Auftrags verwendet. Eine ODER-Zusammenführung von mehr als zwei Berechnungen wird aber mühsam. Deshalb verwenden wir in unserem letzten Beispiel das Gegenstück zu `allOf`, welches *anyOf* heißt. Wie `allOf` ist `anyOf` eine statische Methode mit beliebig vielen `CompletableFuture`-Objekten (VarArgs-Parameter). Der Rückgabetyp von `anyOf` ist `CompletableFuture<Object>`. Das zurückgegebene Objekt kapselt das Ergebnis eines der Vorgängeraufträge, in der Regel desjenigen, der am schnellsten zu Ende gelaufen ist. In Listing 3.31 finden Sie das analoge Beispiel zu Listing 3.30. Nur wird dem letzten Auftrag nicht das zusammengefasste Ergebnis der drei Vorgängeraufträge übergeben, sondern lediglich das Ergebnis des am schnellsten gelaufenen Auftrags. Wegen des Rückgabetyps `CompletableFuture<Object>` ist beim letzten Auftrag ein Casting auf `String` sowohl des Arguments als auch des Rückgabewerts nötig.

**Listing 3.31**

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

public class CompletableFutureDemo
{
    //supply() und apply wie zuvor: ...

    public static void main(String[] args) throws Exception
    {
        CompletableFuture<String> f1 = CompletableFuture.supplyAsync(
            () -> supply());
        CompletableFuture<String> f2 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f3 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<String> f4 = f1.thenApplyAsync(s -> apply(s));
        CompletableFuture<Object> f234 = CompletableFuture.anyOf(f2,f3,f4);
        CompletableFuture<String> f5 = f234.thenApplyAsync(
            s -> (String)apply((String)s));
        System.out.println(f5.get());
    }
}
```

CompletableFuture besitzt noch zahlreiche weitere Methoden. Aber auch ohne deren Besprechung sollten durch die wenigen mit Beispielen erläuterten Methoden das Prinzip und die Grundidee von CompletableFuture deutlich geworden sein.

Sowohl bei den CompletableFutures als auch beim Data-Streaming-Framework können Methoden mit Argumenten von Lambda-Ausdrücken verkettet aufgerufen werden. In beiden Fällen wird dadurch eine Abfolge von Verarbeitungsschritten festgelegt. Bei oberflächlicher Betrachtung sieht beides ähnlich aus. Es gibt aber gravierende Unterschiede zwischen CompletableFutures und dem Data-Streaming, die bei der Besprechung deutlich geworden sein sollten und die wir hier noch einmal zusammenfassen wollen:

- Beim Data-Streaming hat man in der Regel eine Quelle mit mehreren Elementen. Damit werden die Verarbeitungsstufen mehrfach durchlaufen, nämlich für jedes Element der Quelle einmal. Bei den CompletableFutures werden die definierten Verarbeitungsschritte einmalig durchlaufen.
- Beim Data-Streaming geht es immer um eine lineare Verarbeitungskette, während CompletableFutures Verzweigungen und sowohl UND- als auch ODER-Zusammenführungen unterstützen.
- Die Parallelität beim Data-Streaming ist vertikal und basiert auf den verarbeiteten Elementen (s. Bild 3.8), während es bei den CompletableFutures um horizontale Parallelität auf der Ebene der Verarbeitungsstufen geht.
- Das Data-Streaming folgt dem Pull-Prinzip (Nachfolgerstufe holt Element bei Vorgängerstufe ab), während CompletableFutures das Push-Prinzip unterstützen (Vorgängerstufe reicht Element an Nachfolgerstufe weiter).

Damit wollen wir die Besprechung der Concurrent-Bibliothek beenden und uns dem Thema Verklemmungen zuwenden.

## ■ 3.11 Ursachen für Verklemmungen

Das Thema *Verklemmung* wurde im Laufe des Buches an verschiedenen Stellen angesprochen. Diese Problematik wird jetzt genauer betrachtet. Zunächst wird erläutert, wie es zu Verklemmungen kommen kann und was typisch an einer Verklemmung ist. Danach werden im nächsten Abschnitt Maßnahmen besprochen, wie Verklemmungen vermieden werden können.

Das Entstehen einer Verklemmung lässt sich anhand unserer Küchen-Köche-Metapher aus Kapitel 1 erläutern. Angenommen, zwei Köche benötigen eine Schüssel und ein Rührgerät. Sowohl von der Schüssel als auch von dem Rührgerät sei nur je ein Exemplar in der Küche vorhanden. Der erste Koch nimmt zuerst die Schüssel aus dem Schrank und will dann das Rührgerät nehmen. Der zweite Koch will in der umgekehrten Reihenfolge vorgehen. Während nun aber der erste Koch die Schüssel nimmt, beschafft sich der zweite Koch das Rührgerät. Der erste Koch will danach das Rührgerät nehmen. Da dieses aber nicht an seinem Platz ist, muss er warten, bis es derjenige, der es momentan benutzt, zurückgibt. Dem zweiten Koch geht es ähnlich mit der Schüssel. Beide Köche warten darauf, dass der jeweils andere Koch die benötigten Utensilien zurückstellt. Dies wird aber nie geschehen; eine Verklemmungssituation liegt vor. Wie in diesem Fall wird es in allen nachfolgenden Beispielen so sein, dass das Eintreten einer Verklemmungssituation nicht zwangsläufig erfolgt: Die beiden Köche können über Wochen und Monate hinweg arbeiten, wobei der eine zuerst die Schüssel und das Rührgerät nimmt und der andere umgekehrt vorgeht. Dabei kommt es zu keiner Verklemmung, wenn die beiden nie dieselben Utensilien brauchen oder wenn einer der beiden Köche sowohl die Schüssel als auch das Rührgerät bereits benutzt, wenn der andere diese Dinge braucht. Der zweite muss dann einfach warten, bis der erste fertig ist und beide Gegenstände zurückstellt. Die Verklemmung tritt nur ein, wenn sich zufällig die oben beschriebenen Abläufe ereignen.

In der Literatur werden Verklemmungen in der Regel im Zusammenhang mit der Nutzung von *Betriebsmitteln* besprochen. Im soeben besprochenen Beispiel waren die Betriebsmittel die Schüssel und das Rührgerät. In der Welt der Informatik denkt man bei Betriebsmitteln zunächst an Ein-/Ausgabegeräte wie Drucker, Scanner, CD-Brenner, Modem, aber auch an Speicherbereiche und Dateien. Wir werden in diesem Buch den Begriff Betriebsmittel etwas weiter fassen:



**Definition:** Ein Betriebsmittel ist ein Objekt, auf das ein Thread unter Umständen warten muss, bevor er es benutzen kann. In diesem Zusammenhang kann ein Betriebsmittel ein Objekt sein, das Synchronized-Methoden hat, so dass eventuell gewartet werden muss, bevor eine solche Synchronized-Methode auf das Objekt angewendet werden kann. Statt Synchronized-Methoden können auch Synchronized-Blöcke eingesetzt werden. Ein Betriebsmittel kann aber auch ein Objekt sein, dessen Nutzung durch höchstens einen Thread zu einem Zeitpunkt über Semaphore oder Locks realisiert wird.

Im Folgenden betrachten wir hierzu einige Beispiele.

### 3.11.1 Beispiele für Verklemmungen mit synchronized

Im Folgenden betrachten wir eine leicht modifizierte Version des Beispiels mit Konten und einer Bank aus Abschnitt 2.2, wobei die Bankangestellten nun nicht mehr nur einen bestimmten Betrag von einem Konto abbuchen oder einem Konto gutschreiben, sondern jedes Mal einen bestimmten Betrag von einem Konto der Bank auf ein anderes Konto der Bank überweisen. Da keine Zwischenzustände nach außen sichtbar sein sollen, bei denen der Betrag von einem Konto schon abgebucht, aber auf dem anderen Konto noch nicht gutgeschrieben ist, werden in der Methode überweisen der Klasse Bank erst beide Konten durch synchronized gesperrt und dann wird die Überweisung durchgeführt. Die Bankangestellten werden durch Threads repräsentiert, wobei diese jeweils 10.000 Mal zufällig zwei Kontennummern und einen Betrag wählen und dann eine entsprechende Überweisung vornehmen. In Listing 3.32 ist dieses Programm, das eine Verklemmungsgefahr enthält, zu sehen:

**Listing 3.32**

```
class Account
{
    private float balance;

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

class Bank
{
    private Account[] account;

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

    public void transferMoney(int fromAccountNumber,
                             int toAccountNumber,
                             float amount)
    {
        synchronized(account[fromAccountNumber])
        {
            synchronized(account[toAccountNumber])
            {
                account[fromAccountNumber].debitOrCredit(-amount);
                account[toAccountNumber].debitOrCredit(amount);
            }
        }
    }
}

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++)
        {
            /* Nummer eines Kontos einlesen, von dem abgebucht wird;
               simuliert durch Wahl einer Zufallszahl
               zwischen 0 und 99
            */
            int fromAccountNumber = (int)(Math.random()*100);

            /* Nummer eines Kontos einlesen, auf das abgebucht wird;
               simuliert durch Wahl einer Zufallszahl
               zwischen 0 und 99
            */
            int toAccountNumber = (int)(Math.random()*100);

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

            bank.transferMoney(fromAccountNumber, toAccountNumber,
                               amount);
        }
    }
}

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

```

Dieses Programm wird sehr oft ohne Verklemmung zu Ende laufen. Es ist aber auch möglich, dass irgendwann einmal Folgendes passiert: Eine Bankangestellte wählt die Nummer 47 als fromAccountNumber und die Nummer 11 als toAccountNumber. Nehmen wir an, nach Ausführung der Anweisung **synchronized(account[47])** wird auf einen anderen Thread umgeschaltet. Wenn in diesem Thread nun zufällig 11 als fromAccountNumber und 47 als toAccountNumber gewählt wird, dann wird eine Verklemmung eintreten, denn dieser Thread wird zwar noch **synchronized(account[11])** ausführen können, wird aber beim Ausführen von **synchronized(account[47])** blockiert. Wenn nun auf den ersten Thread

umgeschaltet wird, so wird dieser bei der Ausführung von `synchronized(account[11])` ebenfalls blockiert. Es liegt eine Verklemmung vor.

Ein ähnliches Beispiel lässt sich natürlich auch mit Synchronized-Methoden konstruieren. Wir nehmen an, eine Klasse X enthält als Attribut eine Referenz auf ein anderes Objekt der Klasse X. Diese Referenz kann durch die Methode setPartner festgelegt werden. Ferner gibt es zwei Synchronized-Methoden m1 und m2, wobei in m1 die Methode m2 auf das andere X-Objekt angewendet wird. Was in m2 geschieht, ist in diesem Zusammenhang nicht weiter interessant. Wenn nun zwei Objekte erzeugt werden, die jeweils eine Referenz auf das andere Objekt besitzen, so kann bei Aufruf der Methode m1 auf die beiden Objekte durch zwei unterschiedliche Threads eine Verklemmung eintreten. In Listing 3.33 ist das Programm dargestellt.

### Listing 3.33

```
class X
{
    private X otherX;

    public void setPartner(X otherX)
    {
        this.otherX = otherX;
    }

    public synchronized void m1()
    {
        otherX.m2();
    }

    public synchronized void m2()
    {
        // nicht weiter wichtig
    }
}

public class UserOfX extends Thread
{
    private X myX;

    public UserOfX(X x)
    {
        myX = x;
    }

    public void run()
    {
        for(int i = 0; i < 10000; i++)
        {
            myX.m1();
        }
    }

    public static void main(String[] args)
    {
        X x1 = new X();
        X x2 = new X();
```

```

        x1.setPartner(x2);
        x2.setPartner(x1);
        UserOfX user1 = new UserOfX(x1);
        user1.start();
        UserOfX user2 = new UserOfX(x2);
        user2.start();
    }
}

```

Eine Verklemmung tritt ein, wenn nach dem Aufruf der Methode m1, aber vor der Ausführung der Anweisung **otherX.m2()** umgeschaltet wird. In diesem Fall ist eines der X-Objekte gesperrt. Wenn auf den anderen Thread umgeschaltet wird, dann wird dort durch den Aufruf von m1 das andere Objekt gesperrt. Weder der eine noch der andere Thread können dann m2 auf das jeweils andere Objekt anwenden.

### 3.11.2 Beispiele für Verklemmungen mit Semaphoren

Verklemmungsbeispiele gibt es auch bei der Nutzung von Semaphoren, wenn diese zum gegenseitigen Ausschluss für die Nutzung von Objekten eingesetzt werden. Ein Beispiel dazu haben wir bereits beim Philosophen-Problem kennen gelernt. Hier ging es um die Nutzung von Gabeln, wobei jede Gabel durch einen Semaphore repräsentiert wurde. Vor der Nutzung einer Gabel muss die P-Methode auf den entsprechenden Semaphore angewendet werden, nach der Nutzung die V-Methode. Zu einer Verklemmung kann es kommen, wenn alle Philosophen ihre linke Gabel ergreifen. Ein einfacheres Beispiel mit zwei Semaphoren, das zu einer Verklemmung führen kann, ist in Listing 3.34 zu sehen:

**Listing 3.34**

```

class UserOfSemaphore extends Thread
{
    /* die beiden Semaphore werden benutzt
       zum gegenseitigen Ausschluss für die Nutzung je eines Objekts
    */
    private Semaphore s1, s2;

    public UserOfSemaphore(Semaphore s1, Semaphore s2)
    {
        this.s1 = s1;
        this.s2 = s2;
    }

    public void run()
    {
        for(int i = 0; i < 10000; i++)
        {
            s1.p();
            s2.p();
            // Nutzung der beiden Objekte
            s2.v();
            s1.v();
        }
    }
}

```

```

public static void main(String[] args)
{
    Semaphore s1 = new Semaphore(1);
    Semaphore s2 = new Semaphore(1);
    UserOfSemaphore user1 = new UserOfSemaphore(s1, s2);
    user1.start();
    UserOfSemaphore user2 = new UserOfSemaphore(s2, s1);
    user2.start();
}
}

```

Entscheidend bei diesem Programm ist die Tatsache, dass bei einem Aufruf des Konstruktors der Klasse UserOfSemaphore s1 als erstes Argument und s2 als zweites Argument übergeben werden, während beim zweiten Aufruf des Konstruktors diese Reihenfolge umgekehrt ist. Eine Verklemmung kann dann bei der Ausführung dieses Programms eintreten, wenn nach Ausführung der Anweisung **s1.p()** von einem Thread auf den anderen Thread umgeschaltet wird.

Selbstverständlich kann es auch bei der Nutzung von Locks zu Verklemmungen kommen. Da aber entsprechende Beispiele den bereits vorgestellten Beispielen sehr ähnlich wären, verzichten wir darauf.

### 3.11.3 Bedingungen für das Eintreten von Verklemmungen

Wie schon oben erwähnt wurde, wird die Verklemmungsproblematik in der Literatur im Zusammenhang mit der Nutzung von Betriebsmitteln behandelt, wobei unter Betriebsmitteln Objekte verstanden werden, die unter gegenseitigem Ausschluss benutzt werden. Der gegenseitige Ausschluss kann z.B. durch synchronized oder durch Semaphore realisiert werden. Folgende Bedingungen müssen erfüllt sein, damit es zu Verklemmungen kommen kann:

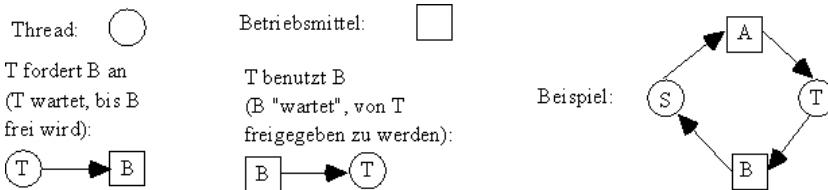
1. Die Betriebsmittel sind nur unter *gegenseitigem Ausschluss* nutzbar (man sagt dazu auch: exklusiv nutzbar).
2. Betriebsmittel, die benutzt werden, können dem benutzenden Thread nicht entzogen werden.
3. Threads besitzen bereits Betriebsmittel und fordern weitere an.

Neben diesen Bedingungen, die erfüllt sein müssen, damit es überhaupt zu Verklemmungen kommen kann, gilt folgende Bedingung genau dann, wenn eine Verklemmung konkret vorliegt:

4. Es gibt eine zyklische Kette von Threads, von denen jeder mindestens ein Betriebsmittel besitzt, das der nächste Thread in der Kette benötigt.

Die Belegungs- und Anforderungssituation von Betriebsmitteln lässt sich grafisch anhand eines *Betriebsmittelgraphen* veranschaulichen. Dabei stellen wir Threads als Kreise und Betriebsmittel als Rechtecke dar. Ferner wird ein Pfeil von einem Betriebsmittel zu einem Thread gezeichnet, falls der Thread dieses Betriebsmittel besitzt. Umgekehrt bedeutet ein Pfeil von einem Thread zu einem Betriebsmittel, dass dieser Thread dieses Betriebsmittel anfordert (d.h. haben möchte). Bild 3.9 fasst diese Punkte zusammen und zeigt ein Beispiel

für einen Betriebsmittelgraph. Die obige Bedingung 4 ist genau dann erfüllt, wenn es in dem Betriebsmittelgraph einen Weg gibt, der von einem Knoten (Kreis oder Rechteck) immer den Pfeilen folgend zu diesem Knoten zurückführt.



**Bild 3.9** Betriebsmittelgraph

In allen geschilderten Verklemmungsbeispielen waren jeweils alle vier Bedingungen erfüllt. Dies soll exemplarisch anhand des einleitenden Beispiels mit den Köchen, der Schüssel und dem Rührgerät veranschaulicht werden:

1. Sowohl die Schüssel als auch das Rührgerät können nur von höchstens einem Koch zu einem Zeitpunkt benutzt werden.
2. Wenn ein Koch die Schüssel oder das Rührgerät zur Benutzung an sich genommen hat, kann kein anderer Koch ihm den Gegenstand wieder wegnehmen, sondern der Gegenstand kann erst dann wieder von anderen benutzt werden, wenn der Koch ihn nicht mehr benötigt und freiwillig an seinen Platz zurückstellt.
3. Ein Koch nimmt zuerst die Schüssel und möchte danach das Rührgerät noch dazu haben. Der andere Koch nimmt zuerst das Rührgerät und möchte danach die Schüssel noch dazu haben.
4. Der Betriebsmittelgraph besitzt einen Zyklus. Der Betriebsmittelgraph für das Köche-Beispiel ist der Gleiche wie derjenige in **Bild 3.9**. Dabei entspricht A der Schüssel und B dem Rührgerät, T dem ersten Koch und S dem zweiten Koch. Der erste Koch T hat die Schüssel A und fordert das Rührgerät B an. Der zweite Koch S hat das Rührgerät B und fordert die Schüssel A an.

Übrigens haben die Betriebsmittelgraphen aller bisher besprochenen Beispiele diese Struktur. Es handelt sich immer um zwei Threads und zwei Betriebsmittel, wobei ein Thread eines der Betriebsmittel besitzt und das jeweils andere anfordert.

## ■ 3.12 Vermeidung von Verklemmungen

Die vier Bedingungen, die genau dann gelten, wenn eine Verklemmung vorliegt, geben Hinweise, wie Verklemmungen vermieden werden können. Offenbar kann keine Verklemmung vorliegen, falls eine der Bedingungen nicht gilt. An dem Zutreffen der Bedingung 1 lässt sich nichts ändern, denn – wie in Kapitel 2 ausführlich beschrieben wurde – ist in vielen Fällen der gegenseitige Ausschluss notwendig. In Bedingung 2 geht es darum, dass Betriebsmittel nicht entzogen werden können. Eine Gegenmaßnahme, auf die wir in diesem Buch nicht näher eingehen wollen, besteht beispielsweise darin, dass von Zeit zu Zeit geprüft

wird, ob eine Verklemmung vorliegt oder nicht. Ist eine Verklemmungssituation eingetreten, dann werden einigen Threads Betriebsmittel entzogen, so dass die Verklemmung beseitigt wird. Die Realisierung des Betriebsmittelentzugs kann z.B. durch Abbrechen der Threads erfolgen und unter Umständen sehr viel Aufwand erfordern. Aus den restlichen Bedingungen 3 und 4 ergeben sich vier Gegenmaßnahmen, wobei die erste Gegenmaßnahme aus der Nichterfüllung der Bedingung 3 und die drei anderen Gegenmaßnahmen aus der Nichterfüllung der Bedingung 4 hervorgehen:

1. Ein Thread darf nur Betriebsmittel anfordern, wenn er keine Betriebsmittel besitzt.
2. Ein Thread fordert Betriebsmittel immer in einer bestimmten Reihenfolge an. Dadurch ist garantiert, dass keine zyklischen Wartebeziehungen entstehen.
3. Ein Thread t<sub>1</sub> darf auf ein Betriebsmittel, das ein anderer Thread t<sub>2</sub> gerade benutzt, nur dann warten, falls bestimmte Beziehungen zwischen t<sub>1</sub> und t<sub>2</sub> gelten. Auch dadurch wird garantiert, dass es zu keinen zyklischen Wartebeziehungen kommen kann.
4. Beim Anfordern von Betriebsmitteln wird anhand einer Bedarfsanalyse untersucht, ob es „im schlimmsten Fall“ bei Erfüllung dieser Forderung zu einer Verklemmung kommen könnte. Nur, falls es selbst „im schlimmsten Fall“ zu keiner Verklemmung kommen kann, wird die Anforderung erfüllt.

Bevor wir auf die drei Gegenmaßnahmen näher eingehen, wollen wir den Kontext beschreiben, in den diese Maßnahmen eingebettet sind. Wir gehen im Folgenden davon aus, dass ein Objekt einer Klasse ResourceManager existiert, auf das alle Threads, die Betriebsmittel benutzen wollen, Zugriff haben. Vor der Benutzung von Betriebsmitteln werden diese beim *Betriebsmittelverwalter-Objekt* durch Aufruf einer entsprechenden Methode angefordert, wobei der aufrufende Thread unter Umständen blockiert werden kann. Nachdem der Methodenaufruf beendet ist, kann der Thread die angeforderten Betriebsmittelobjekte exklusiv benutzen. Die Methoden der Betriebsmittelobjekte müssen nicht synchronized sein, da der gegenseitige Ausschluss durch die vorhergehende Anforderung beim Betriebsmittelverwalter garantiert wird. Die Methoden des Betriebsmittelverwalters werden aber in der Regel synchronized sein müssen. Falls die Methoden der Betriebsmittelobjekte aber auch synchronized sind, so schadet dies nicht, denn eine Verklemmung ist durch die Arbeitsweise des Betriebsmittelverwalters ausgeschlossen. Wenn ein Thread Betriebsmittel nicht mehr benutzen möchte, muss er diese durch Aufruf einer entsprechenden Methode des Betriebsmittelverwalters freigeben, wodurch die Anforderungen wartender Threads eventuell erfüllt werden können.

Im Folgenden gehen wir weiter davon aus, dass die Betriebsmittel durch ihren Typ in Gruppen zusammengefasst sind. Einem anfordernden Thread soll es dabei gleichgültig sein, welche konkreten Betriebsmittelobjekte er nutzen kann, sofern nur der Typ stimmt. Das heißt, ein Thread fordert bestimmte Mengen von *Betriebsmitteltypen* an (z.B. 3 Exemplare des Typs 7 und 1 Exemplar des Typs 11), und ihm werden dann entsprechende konkrete Objekte zur Nutzung zugeteilt. Falls man keine Betriebsmittel in einer Typgruppe zusammenfassen kann, so besitzt jedes Betriebsmittel seinen eigenen Typ; von diesem Typ gibt es dann eben nur ein einziges Exemplar.

Anforderungen und Freigaben von Betriebsmitteln werden durch ein Int-Feld beschrieben, dessen Länge der Anzahl der Betriebsmitteltypen entspricht. Der Wert dieses Feldes an der Stelle i gibt die Anzahl der Exemplare des Typs i an. Sowohl beim Anfordern als auch beim

Freigeben wird eine Referenz auf ein solches Feld als Parameter übergeben, wobei dadurch die Anzahl der anzufordernden bzw. freizugebenden Betriebsmittelexemplare pro Typ beschrieben wird. Falls es pro Typ nur genau ein Exemplar gibt, so kann man sich vorstellen, dass ein Thread nach erfolgreicher Anforderung die Betriebsmittel direkt nutzen kann, da ihm die entsprechenden Referenzen bekannt sind. Existieren dagegen mehrere Exemplare pro Typ, dann muss die Methode zum Anfordern die zugeteilten Exemplare an den anfordernden Thread bekannt geben. Wir wollen in der folgenden Beschreibung allerdings davon absehen und nehmen an, dass die benutzbaren Exemplare den anfordernden Threads auf eine andere Art bekannt gemacht werden. Die Schnittstelle des Betriebsmittelverwalters sieht dann wie folgt aus:

```
public interface ResourceManager
{
    public void acquire(int[] resources);
    public void release(int[] resources);
}
```

Eine naive Implementierung eines Betriebsmittelverwalters ohne besondere Strategie, der jede Anforderung erfüllt, sofern die angeforderten Betriebsmittel vorhanden sind, entspricht derjenigen einer Semaphorgruppe. Der Einfachheit halber gehen wir hier von einer korrekten Nutzung des Betriebsmittelverwalters aus. Das heißt, man kann davon ausgehen, dass alle Werte im Parameterfeld größer oder gleich null sind. Außerdem soll ein Thread nur die Betriebsmittel zurückgeben, die er auch angefordert hat. Ein naiver Betriebsmittelverwalter kann unter Verwendung einer Semaphorgruppe wie in Listing 3.35 implementiert werden:

### Listing 3.35

```
public class ResourceManagerNaive implements ResourceManager
{
    private SemaphoreGroup availableResources;

    public ResourceManagerNaive(int[] initialResources)
    {
        availableResources =
            new SemaphoreGroup(initialResources.length);
        availableResources.changeValues(initialResources);
    }

    public void acquire(int[] resources)
    {
        /* angeforderte Ressourcen müssen von verfügbaren Ressourcen
         * abgezogen werden
        */
        int[] tmp = new int[resources.length];
        for(int i = 0; i < resources.length; i++)
        {
            tmp[i] = -resources[i];
        }
        availableResources.changeValues(tmp);
    }

    public void release(int[] resources)
    {
```

```

    /* zurückgegebene Ressourcen müssen zu
     * verfügbaren Ressourcen addiert werden
     */
    availableResources.changeValues(resources);
}
}

```

Die Methoden acquire und release müssen bei dieser Implementierung eines Betriebsmittelverwalters nicht synchronized sein, da die Methode changeValues der Klasse SemaphoreGroup synchronized ist. Diese Implementierung ist allerdings im Namen als naiv gekennzeichnet, da hier Verklemmungen möglich sind. Denn ein Thread kann mit dieser Implementierung zuerst ein Exemplar des Typs 0 und danach durch ein erneutes Aufrufen der Methode acquire ein Exemplar des Typs 1 anfordern. Ein anderer Thread kann in umgekehrter Reihenfolge vorgehen. Falls es vom Typ 0 und 1 jeweils nur ein einziges Exemplar gibt, kann es wieder zu einer Verklemmung kommen.

### 3.12.1 Anforderung von Betriebsmitteln „auf einen Schlag“

Die erste Gegenmaßnahme gegen Verklemmungen besteht darin, dass ein Thread nur dann Betriebsmittel anfordern darf, falls ihm momentan keine zugewiesen sind. Damit kann es keine Verklemmungen geben. Betrachtet man nämlich den Betriebsmittelgraph, so gilt für jeden Thread zu jedem Zeitpunkt, dass entweder von ihm nur Pfeile ausgehen oder Pfeile auf ihn zulaufen können, aber nicht beides gleichzeitig, denn wenn der Thread Betriebsmittel besitzt, darf er keine anfordern, und wenn er Betriebsmittel anfordert, darf er keine besitzen. Damit es einen Zyklus geben kann, muss es aber mindestens einen Thread geben, von dem sowohl Pfeile ausgehen als auch an ihm enden. Da dies nie vorkommt, kann es nie zu einem zyklischen Warten und damit auch nie zu einer Verklemmung kommen.

Diese Strategie kann auf mehrere Arten umgesetzt werden:

1. Eine Möglichkeit besteht darin, die zuvor angegebene naive Implementierung Resource-ManagerNaive zu nutzen und die Threads, die Betriebsmittel anfordern, so zu programmieren, dass die beschriebene Strategie eingehalten wird.
2. Eine weitere Möglichkeit der Umsetzung besteht darin, dass sich das Betriebsmittelverwalter-Objekt für jeden Thread alle aktuell zugewiesenen Betriebsmittel merkt. Diese Informationen können dann auch beim Freigeben herangezogen werden, um zu überprüfen, ob die freigegebenen Betriebsmittel dem Thread auch tatsächlich zugewiesen waren. Wenn ein Thread, der schon Betriebsmittel besitzt, weitere Betriebsmittel anfordert, dann kann dies aufgrund dieser Informationen erkannt werden. Man kann darauf auf mehrere Arten reagieren: Man lehnt diese Anforderung als fehlerhaft ab und löst z.B. eine Ausnahme aus. Oder man könnte in diesem Fall in der Methode acquire alle Betriebsmittel zurückgeben und anschließend die zuvor freigegebenen und die neu benötigten wieder zusammen anfordern.

Beim Anfordern mehrerer Betriebsmittel ist wichtig, dass diese „auf einen Schlag“ angefordert werden. Dies wird durch die Implementierung mit einer Semaphorgruppe garantiert. Die Bedeutung dieser Änderung der Semaphorwerte „auf einen Schlag“ wurde schon bei der Beschreibung von Semaphorgruppen in Abschnitt 3.1 erläutert. Die Lösung des Philoso-

phen-Problems mit Semaphorgruppen entspricht dieser Strategie. Es werden dort nämlich nur Gabeln angefordert, wenn man momentan keine besitzt. Ferner erfolgt die Anforderung der beiden Gabeln „auf einen Schlag“, so dass es zu keiner Verklemmung kommen kann.

### 3.12.2 Anforderung von Betriebsmitteln gemäß einer vorgegebenen Ordnung

Eine weitere Maßnahme gegen Verklemmungen besteht darin, Betriebsmittel nur gemäß einer vorgegebenen Reihenfolge anzufordern. Zur Definition einer solchen Reihenfolge kann z.B. die Nummerierung der Betriebsmitteltypen in aufsteigender Reihenfolge verwendet werden. Dies bedeutet, dass ein Thread nur Betriebsmittel des Typs s anfordern darf, falls ihm im Moment keine Betriebsmittel des Typs s oder eines Typs  $t > s$  zugeteilt sind. Mit anderen Worten: Einem Thread, der Betriebsmittel des Typs s anfordert, dürfen zu diesem Zeitpunkt nur Betriebsmittel der Typen  $t < s$  zugeteilt sein. Dies bedeutet beispielsweise, dass ein Thread zuerst Betriebsmittel des Typs 2 anfordern kann. Sind ihm diese zugeteilt worden, so darf er nur noch Betriebsmittel des Typs 3, 4 usw. anfordern. Angenommen, er fordert anschließend Betriebsmittel des Typs 7 an, dann kann er anschließend nur noch welche vom Typ 8 oder höher anfordern.

Wie zuvor kann man durch Überlegungen mit Hilfe eines Betriebsmittelgraphen erkennen, dass diese Strategie tatsächlich Verklemmungsfreiheit garantiert. Nehmen wir an, es gäbe einen Zyklus in einem Betriebsmittelgraph. Dann muss es einen Knoten für ein Betriebsmittel des Typs i1 geben, von dem ein Pfeil zu einem Thread führt. Dies bedeutet, dass diesem Thread dieses Betriebsmittel zugeordnet ist. Weiterhin muss von diesem Thread ein Pfeil zu einem weiteren Betriebsmittel führen, dessen Typ i2 ist. Dieser Pfeil bedeutet, dass dieses Betriebsmittel von diesem Thread angefordert wird. Aufgrund der angegebenen Strategie muss gelten:  $i1 < i2$ . Mit derselben Argumentation gibt es ein weiteres Betriebsmittel in diesem Zyklus vom Typ i3, wobei gelten muss:  $i1 < i2 < i3$ . Da wir von einem Zyklus ausgegangen sind, kommt man irgendwann zu dem ursprünglichen Betriebsmittel zurück. Es müsste dann gelten:  $i1 < i2 < i3 < \dots < iN < i1$ , also  $i1 < i1$ . Da dies ein Widerspruch ist, muss die ursprüngliche Annahme, dass es einen Zyklus im Betriebsmittelgraph geben kann, falsch sein.

Für die Umsetzung dieser Strategie gibt es dieselben Möglichkeiten wie zuvor:

1. Die erste Möglichkeit besteht wieder darin, die naive Implementierung ResourceManagerNaive zu nutzen und die Threads, die Betriebsmittel anfordern, so zu programmieren, dass die beschriebene Strategie eingehalten wird.
2. Die zweite Möglichkeit der Umsetzung besteht auch wieder darin, dass sich das Betriebsmittelverwalter-Objekt für jeden Thread alle aktuell zugeteilten Betriebsmittel merkt. Wenn ein Thread Betriebsmittel eines Typs anfordert, der kleiner oder gleich eines Typs ist, von dem ihm schon Betriebsmittel zugeteilt sind, so kann dies erkannt werden. Als Reaktionsmöglichkeiten kommen dieselben wie zuvor in Frage: Die Anforderung wird zurückgewiesen. Oder man gibt zuerst diejenigen Betriebsmittel frei, deren Typ größer oder gleich dem angeforderten Typ ist. Anschließend fordert man alle freigegebenen und die neu gewünschten in der richtigen Reihenfolge wieder an.

Wendet man dieses Prinzip auf das Philosophen-Problem an, so ergibt sich eine einfache, verklemmungsfreie Lösung auch bei Nutzung einfacher Semaphore (vgl. Listing 3.10 in Abschnitt 3.4.2). Die Gabeln entsprechen Betriebsmitteln unterschiedlicher Typen, wobei die Typnummer die Nummer der Gabel ist. Wenn man dann die Gabeln in der Reihenfolge ihrer Typnummer anfordert, kann keine Verklemmung entstehen. Bei der naiven Lösung, die eine Verklemmungsgefahr birgt, ist es ja so, dass alle Philosophen erst ihre linke Gabel und dann ihre rechte Gabel nehmen. Alle Philosophen i außer dem Philosoph mit der Nummer N-1 haben zuerst die linke Gabel mit der Nummer i und dann die rechte Gabel mit der Nummer  $i+1$  ergriffen. Diese fordern die Gabeln also in der richtigen Reihenfolge an. Der Philosoph mit der Nummer N-1 fordert ebenfalls zuerst die linke Gabel, die die Nummer N-1 trägt, und dann die rechte Gabel mit der Nummer 0 an. Dieser Philosoph fordert also die Betriebsmittel nicht in der richtigen Reihenfolge an. Wenn der Philosoph N-1 als einziger nun aber zuerst seine rechte Gabel und dann seine linke Gabel nimmt, dann erhält man auch bei der Nutzung einfacher Semaphore eine verklemmungsfreie Lösung. Es ist übrigens unwichtig, welcher Philosoph seine rechte Gabel zuerst nimmt. Man erhält immer eine verklemmungsfreie Lösung, falls alle Philosophen bis auf einen zuerst die linke und dann die rechte Gabel nehmen und einer in umgekehrter Reihenfolge.

### 3.12.3 Weitere Verfahren

Es gibt noch weitere Verfahren zur Vermeidung von Verklemmungen, die aber im Java-Umfeld keine oder nur eine untergeordnete Rolle spielen dürfen:

Wait-Die-Verfahren: Bei diesen Verfahren bezieht sich die Ordnung nicht auf die Betriebsmittel, sondern auf die Threads. Den Threads werden beim Start eindeutige Nummern zugewiesen. Man erlaubt dann im Betriebsmittelgraph (s. Bild 3.9) Wartebeziehungen zwischen Threads nur in aufsteigender Reihenfolge dieser Thread-Nummern. Wenn man davon ausgeht, dass die Thread-Nummern aufsteigend vergeben werden, heißt dies, dass nur ältere Threads (mit kleinerer Nummer) auf jüngere Threads (mit größerer Nummmer) warten dürfen, umgekehrt aber nicht. Man kann dann mit genau derselben Überlegung wie bei der Anforderung von Betriebsmitteln gemäß einer vorgegebenen Ordnung (s. voriger Abschnitt) folgern, dass es im Betriebsmittelgraph keinen Zyklus geben kann, weil sonst für die Nummer t eines Threads  $t < t$  gelten müsste, was zu einem Widerspruch führt. Wenn also ein Thread Betriebsmittel anfordert, wird für den Fall, dass das Betriebsmittel schon benutzt wird, geprüft, ob die Nummer des Threads, der das Betriebsmittel belegt hat, größer ist als die Nummer des anfordernden Threads. Ist dies der Fall, so kann der anfordernde Thread auf das Freiwerden des Betriebsmittels warten (Wait), andernfalls startet der anfordernde Thread von vorne. Das heißt, dass er sich zuerst beendet (Die), dabei alle schon vorgenommenen Änderungen zurücknimmt (das ist insbesondere in einem Datenbankumfeld, das Transaktionen unterstützt, einfach realisierbar), alle Betriebsmittel freigibt und anschließend wieder ganz von vorne beginnt. Bei diesem Neustart bekommt der Thread dann aber keine neuen, höhere Nummer zugewiesen, denn je älter ein Thread ist (also je kleiner seine Nummer ist), umso geringer ist die Wahrscheinlichkeit, dass er nochmals abgebrochen wird, da andere ältere Threads mit einer gewissen Wahrscheinlichkeit nun schon zu Ende gelaufen sind. Bitte beachten Sie, dass es keine Lösung wäre, wenn der Thread, der auf das

benötigte Betriebsmittel nicht warten darf, einfach später nochmals danach fragt, ohne zuvor die Betriebsmittel, die er schon in Besitz genommen hat, freizugeben. Denn dies läuft letztlich auch auf ein Warten hinaus, so dass es auch in diesem Fall zu einer Situation wie in Abschnitt 3.10 kommen könnte.

Variante des Wait-Die-Verfahrens: Bei einer Variante des Wait-Die-Verfahrens ist nur erlaubt, dass jüngere auf ältere Threads warten, umgekehrt aber nicht (also genau das Gegenteil zum ursprünglichen Verfahren). Wenn dann ein älterer Thread ein Betriebsmittel anfordert, das von einem jüngeren Thread gehalten wird, dann sollte dem älteren Thread beim Neustarten allerdings eine neue Nummer zugewiesen werden, da in diesem Fall die Wahrscheinlichkeit steigt, dass man warten darf, wenn man jünger ist.

Wound-Wait-Verfahren: Als eine anderen Variante des Wait-Die-Verfahrens kann das Wound-Wait-Verfahren betrachtet werden. Bei diesem Verfahren wird in dem Fall, dass man nicht warten darf, nicht der anfordernde Thread, sondern der Thread, der das Betriebsmittel schon reserviert hat, abgebrochen (Wound). Bei all diesen Verfahren können Abbrüche auch unnötig erfolgen, denn wenn man ein Warten, das zum Abbruch führt, zulassen würde, so würde dies nicht in jedem Fall zu einer Verklemmung führen.

Anforderung von Betriebsmitteln mit Bedarfsanalyse (Bankier-Algorithmus): Dieses Verfahren setzt die Kenntnis voraus, wie viele Exemplare jedes Betriebsmitteltyps jeder Thread höchstens brauchen wird. Mit diesen Informationen kann bei jeder Anforderung von Betriebsmitteln geprüft werden, ob es nach Zuteilung der angeforderten Betriebsmittel noch möglich wäre, dass selbst dann, wenn alle Threads ihren Maximalbedarf auch tatsächlich anfordern würden, alle Threads in irgendeiner Reihenfolge bedient werden und zu Ende laufen könnten. Ist dies der Fall, so erfolgt die Zuteilung, andernfalls nicht.

Wir wollen diese Idee anhand eines Beispiels näher erläutern. Angenommen, es gäbe der Einfachheit halber nur einen Typ von Betriebsmitteln und davon zwei Exemplare. Wenn zwei Threads beide Exemplare benötigen, dann kann es zu einer Verklemmung kommen, wenn beide Threads zuerst jeweils nur ein Exemplar anfordern und wenn beide danach, nachdem ihnen jeweils ein Exemplar zugeteilt wurde, ein zweites Exemplar anfordern. Da zu diesem Zeitpunkt nämlich kein Exemplar verfügbar ist, wartet jeder Thread, bis der jeweils andere Thread das benötigte Exemplar freigibt. Wenn allerdings im Voraus bekannt ist, dass beide Threads beide Exemplare benötigen, so kann man dies bei der Zuteilung der Betriebsmittel berücksichtigen. Wenn der erste Thread ein Exemplar anfordert, so kann diese Anforderung ohne Bedenken erfüllt werden, denn sollten nun beide Threads ihren Maximalbedarf an Betriebsmitteln einfordern (der erste Thread noch ein weiteres Exemplar, der zweite Thread zwei Exemplare), dann gibt es eine Reihenfolge, in der die Threads bedient werden und zu Ende laufen können. Man würde nämlich in diesem Fall dem ersten Thread das noch fehlende zweite Exemplar zuteilen, worauf dieser Thread irgendwann zu Ende läuft und alle Betriebsmittel freigibt. Danach kann der zweite Thread bedient werden. Die erste Anforderung ist also unkritisch und somit kann ein Betriebsmittelexemplar dem ersten Thread zugeteilt werden. Man spricht in diesem Fall von einem sicheren Zustand der Betriebsmittelbelegungen. Anschließend könnte der zweite Thread ebenfalls ein Betriebsmittelexemplar anfordern. Wenn nun aber auch diese Anforderung erfüllt würde, dann würde man „im schlimmsten Falle“ (beide Threads fordern jeweils ein weiteres Exemplar an) keine Reihenfolge mehr finden, in der die Threads bedient werden können. Man hätte einen unsicheren Belegzungszustand. Deshalb würde die Forderung des zweiten Threads

nach einem Betriebsmittelexemplar nicht erfüllt werden, obwohl noch eines frei ist. Achtet man darauf, dass man nur solche Anforderungen akzeptiert, die zu sicheren Belegungszuständen führen, dann kann es zu keiner Verklemmung kommen.

## ■ 3.13 Zusammenfassung

In den ersten fünf Abschnitten dieses Kapitels wurde die Nutzung der im vorigen Kapitel eingeführten Synchronisationsprimitive synchronized, wait und notify bzw. notifyAll anhand zahlreicher Beispiele illustriert. In den ersten drei Abschnitten wurden die aus Unix bzw. Linux bekannten Synchronisations- und Kommunikationskonzepte Semaphore, Message Queues und Pipes mit den Java-Synchronisationsprimitiven nachimplementiert. Es sei nochmals darauf hingewiesen, dass bei dieser Implementierung nur Threads, die demselben Prozess angehören, interagieren können, während dies in Unix bzw. Linux auch Prozessgrenzen überschreitend möglich ist. Danach wurden aus der Betriebssystemliteratur bekannte Synchronisationsprobleme gelöst. Es handelte sich dabei um das Philosophenproblem und das Leser-Schreiber-Problem. Die in den ersten fünf Abschnitten gesammelten Erfahrungen bei der Nutzung der Java-Synchronisationsprimitive wurden in Abschnitt 3.6 in Form von Schablonen für Methoden, die synchronized, wait und notify bzw. notifyAll verwenden, verdichtet.

In den Abschnitten 3.7 bis 3.10 wurde dann die seit Java 5 existierende und seit Java 7 und Java 8 erweiterte Concurrent-Klassenbibliothek vorgestellt. Diese lässt sich als Alternative zu den bisher vorgestellten Sachverhalten sehen: So ist es mit Executors und Thread-Pools für manche Aufgaben nicht mehr notwendig, selber in seinem Programm explizit Threads zu starten und auf deren Ende zu warten, sondern es können Aufträge zur parallelen Ausführung über Executor-Schnittstellen übergeben werden, wobei für baumartige Berechnungen die Nutzung des Fork-Join-Frameworks sich anbietet. Locks und Conditions stellen eine Alternative zu den Java-Synchronisationsprimitiven dar. Die Nutzung der neuen Synchronisationsmittel benötigt aber im Wesentlichen dieselben Kenntnisse wie die Nutzung der alten. Außerdem gibt es eine Reihe von Klassen zur Synchronisation (Klassen Semaphore, CyclicBarrier usw.) und zur Kommunikation (Klassen ArrayBlockingQueue, LinkedBlockingQueue usw., die die Schnittstelle BlockingQueue implementieren), die besprochen wurden.

In den Abschnitten 3.9 und 3.10 ging es um zwei Formen von Fließbandverarbeitung, die in Java 8 eingeführt wurden. Der Abschnitt 3.9 präsentierte das durch den Big-Data-Bereich inspirierte Data-Streaming-Framework zur fließbandartigen Verarbeitung größerer Datens Mengen. Das Besondere daran war, dass durch den Entwickler festgelegt wird, ob die Verarbeitung sequenziell oder parallel erfolgen soll, dass man sich aber um die Realisierung der Parallelverarbeitung nicht kümmern muss (und auch nicht darf). Wie das Data-Streaming-Framework erlauben auch die in Abschnitt 3.10 behandelten CompletableFutures von Java 8 eine fließbandartige Verarbeitung mit der Möglichkeit von Verzweigungen und Zusammenführungen von Verarbeitungsschritten sowie weiteren gravierenden Unterschieden zum Data-Streaming-Framework, die im Text herausgearbeitet wurden.

Im Abschnitt 3.11 wurden typische Beispiele vorgestellt, wie es zu Verklemmungen kommen kann. Diese Problematik sollte man bei der Synchronisation nie aus den Augen verlieren, ganz gleichgültig, ob man mit synchronized, Semaphoren, Locks oder anderen Synchronisationsmitteln arbeitet. Im Abschnitt 3.12 wurden aus dem Bereich der Betriebssysteme bekannte Ansätze vorgestellt, wie man Verklemmungen vermeiden kann. Diese Ansätze lassen sich auf die Benutzung von synchronized, Locks, Semaphoren usw. übertragen.

# 4

# Parallelität und grafische Benutzeroberflächen

In diesem Kapitel wenden wir uns der Frage zu, welche Wechselwirkungen es zwischen der Programmierung grafischer Benutzeroberflächen und dem Thema Parallelität mit Threads in Java gibt. Insbesondere werden Sie sehen, dass Parallelität eingesetzt werden sollte, wenn länger dauernde Tätigkeiten z.B. durch das Drücken einer Schaltfläche (eines Buttons), die sich in einem Fenster einer grafischen Benutzeroberfläche befindet, angestoßen werden. Bevor wir uns diesem Thema in Abschnitt 4.5 zuwenden, wird als Grundlage in den Abschnitten 4.1 bis 4.4 zuerst eine stark kondensierte Einführung in die Programmierung grafischer Benutzeroberflächen mit JavaFX gegeben, da nicht davon ausgegangen wird, dass die Leserinnen und Leser JavaFX bereits kennen. In Abschnitt 4.1 folgen zur Einstimmung einige erste Beispiele, darunter das unvermeidliche „Hallo Welt“. In JavaFX spielen Properties, Bindings und JavaFX-Collections eine wichtige Rolle. Da sie intensiv benutzt werden, ist eine tiefergehende Behandlung von JavaFX ohne Kenntnisse dieser Konzepte nicht möglich. Deshalb ist Abschnitt 4.2 diesem Thema gewidmet. Anschließend folgt in Abschnitt 4.3 ein Überblick über die wichtigsten Elemente von JavaFX, insbesondere Container, Interaktionselemente und grafische Elemente. Nachdem die wesentlichen Bausteine bekannt sind, wird in Abschnitt 4.4 das MVP-Architekturmuster vorgestellt (MVP: Model – View – Presenter), welches dabei hilft, den Programmcode von Anwendungen mit grafischen Benutzeroberflächen zu strukturieren. In Abschnitt 4.5 geht es dann schließlich darum, welche Rolle das Thema Parallelität im Kontext grafischer Benutzeroberflächen spielt.

Die Thematik, welche Wechselwirkungen zwischen Threads und grafischen Benutzeroberflächen bestehen, ist für alle Arten von Anwendungen wichtig. Im Kontext dieses Buches kommt dem Thema aber besondere Bedeutung zu, denn bei verteilten Anwendungen wird die Client-Seite in der Regel mit einer grafischen Benutzeroberfläche ausgestattet. Werden durch Benutzeraktionen Vorgänge auf dem Server angestoßen, so können diese aus unterschiedlichen Gründen länger dauern (z.B. wenn bei einer langsamen Verbindung viele Daten zum Server übertragen werden sollen oder wenn der Server wegen eines Ausfalls nicht mehr antwortet). Eine naive Implementierung erzeugt ein Verhalten des Client-Programms, das nicht als gebrauchstauglich bezeichnet werden kann. Im Zusammenhang mit verteilten Anwendungen spielt aber auch das Thema MVP eine wichtige Rolle. Ähnlich wie eine nicht verteilte Anwendung mit (oder ohne) grafische Benutzeroberfläche gemäß diesem Entwurfsmuster realisiert werden kann, kann MVP auch als grundlegende Struktur einer verteilten Anwendung dienen.

In diesem Buch verwenden wir ausschließlich *JavaFX*. In Java gibt es alternativ dazu auch z.B. das ältere *Swing*, das noch ältere *AWT (Abstract Window Toolkit)* oder das aus Eclipse stammende, ebenfalls schon ältere *SWT (Standard Window Toolkit)*.

## ■ 4.1 Einführung in die Programmierung grafischer Benutzeroberflächen mit JavaFX

### 4.1.1 Allgemeines zu grafischen Benutzeroberflächen

Programme mit grafischer Benutzeroberfläche erzeugen ein oder mehrere Fenster auf dem Bildschirm. Die sichtbaren Elemente in einem Fenster sind Labels (Anzeigeflächen für Texte und Icons), Buttons (Schaltflächen zum Anklicken mit der Maus), CheckBoxes (Auswahllemente zum Selektieren), RadioButtons (wie CheckBoxes Auswahlelemente, in der Regel werden mehrere zu einer Gruppe zusammengefasst, so dass nur höchstens ein RadioButton dieser Gruppe ausgewählt sein kann), Sliders (Schieberegler zum Einstellen eines numerischen Werts), Texteingabeelemente (einzeilig, mehrzeilig, speziell für Passwörter, bei denen der eingetippte Text nicht sichtbar ist) usw. Als Sammelbegriff für alle diese sichtbaren Elemente eines Fensters benutzen wir hier den Begriff Interaktionselemente (Widgets, UI Controls). Diese Elemente sind in einer bestimmten Weise im Fenster angeordnet. Es gibt unterschiedliche Arten solcher Anordnungen (Layouts) wie zum Beispiel: die Elemente stehen nebeneinander, die Elemente sind untereinander angeordnet, die Elemente bilden eine regelmäßige oder unregelmäßige Gitterstruktur, die Elemente sind auf einen größeren zentralen Bereich und mehrere kleinere Bereiche am oberen, unteren, linken und rechten Rand verteilt usw. In der Regel kommt man bei komplexeren Fensterinhalten nicht mit einem einzigen Layout aus. Stattdessen werden Layouts kombiniert. So kann man zum Beispiel einen zentralen Bereich haben, in denen sich mehrere Elemente in einer Gitterstruktur befinden, und einen oberen Randbereich mit nebeneinander angeordneten Elementen. Zur Realisierung des Layouts gibt es unsichtbare Container-Elemente (Behälter), in die andere Elemente eingefügt und dann entsprechend des Layouts des Containertyps angeordnet werden. Um die Kombination von Layouts zu bewerkstelligen, können Container in andere Container eingefügt werden. Man erhält damit eine Baumstruktur für alle sichtbaren und unsichtbaren Elemente eines Fensters mit dem Fenster als Wurzel. Ein Fenster hat in der Vielzahl der Fälle einen Rahmen mit einem Titel und Buttons zum Minimieren und Maximieren der Fenstergröße sowie einen Button zum Schließen. Über den Rahmen lässt sich das Fenster mit der Maus auf die gewünschte Größe ziehen. In einer Anwendung kann es selbstverständlich mehrere Fenster geben mit ähnlichem oder völlig unterschiedlichem Aussehen. Manche Fenster werden während der Benutzung der Anwendung dynamisch erzeugt und auch wieder geschlossen.

## 4.1.2 Erstes JavaFX-Beispiel

Während diese Aussagen nahezu universell für die unterschiedlichsten Systeme für grafische Benutzeroberflächen gelten, wird speziell in JavaFX ein Fenster als *Stage* (Bühne) bezeichnet. Das erste Fenster einer Anwendung wird dabei nicht im Anwendungscode erzeugt, sondern wenn der selbst geschriebene Programmcode betreten wird, gibt es bereits die erste Stage, die einer bestimmten Methode der eigenen Anwendung als Parameter zum Befüllen übergeben wird. Weitere Fenster können dann bei Bedarf im eigenen Programmcode erzeugt werden. Eine weitere Besonderheit bei JavaFX ist, dass zu einer Stage keine Elemente hinzugefügt werden können. Man braucht zunächst noch eine *Scene* (Szene). Diese Scene muss der Stage hinzugefügt werden. In die Scene kann dann ein einziges Element eingefügt werden. Da man in der Regel immer mehrere Interaktionselemente braucht, wird dieses eine Element fast immer ein Container sein.

Das „Kochrezept“ für eine JavaFX-Anwendung sieht so aus: Man leitet aus der vorgegebenen Klasse *Application* eine eigene Klasse ab. Darin überschreibt man die Methode *start*, die einen Parameter des Typs *Stage* hat. In der Methode *start* baut man die Oberfläche auf, steckt diese in eine Scene und die Scene in die übergebene Stage. Für die Stage kann noch ein Titel angegeben werden. Damit die Stage als Fenster auf dem Bildschirm erscheint, muss sie mit Hilfe der Methode *show* sichtbar gemacht werden. Außerdem kann man die Main-Methode auch in die aus *Application* abgeleitete Klasse setzen. Darin sollte dann die vererbte statische Methode *launch* aufgerufen werden. Der Programmcode unserer ersten *HelloWorld*-Anwendung sieht folglich wie in Listing 4.1 aus. Als Fensterinhalt haben wir hier zwei Labels (Anzeigeflächen) mit den Texten „Hallo“ und „Welt“, die in einer *VBox* vertikal (also untereinander) angeordnet werden (eine *HBox* würde die Labels dagegen horizontal, also nebeneinander platzieren).

**Listing 4.1**

```
import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;

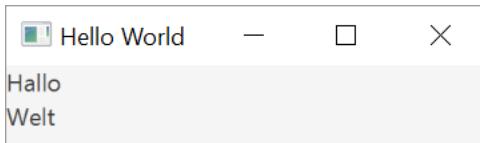
public class HelloWorld extends Application
{
    public void start(Stage primaryStage)
    {
        Label l1 = new Label("Hallo");
        Label l2 = new Label("Welt");
        VBox root = new VBox();
        root.getChildren().add(l1);
        root.getChildren().add(l2);

        Scene scene = new Scene(root, 240, 40);
        primaryStage.setScene(scene);
        primaryStage.setTitle("Hello World");
        primaryStage.show();
    }

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

```
{
    launch(args);
}
```

Wenn man die Anwendung ausführt, erscheint wenig überraschend das in Bild 4.1 dargestellte Fenster auf dem Bildschirm. Drückt man auf das Kreuz oben rechts im Rahmen des Fensters, schließt sich nicht nur das Fenster, sondern auch die Anwendung terminiert (das heißt, der entsprechende Prozess verschwindet aus dem System).

**Bild 4.1**

Erstes JavaFX-Beispiel: „Hallo Welt“  
(Listing 4.1)

Auf drei Punkte des Programms von Listing 4.1 soll noch eingegangen werden: Obwohl VBox ein Container ist, kann man die Methode *add* nicht direkt auf ein VBox-Objekt anwenden. Stattdessen muss man sich über *getChildren* erst die Liste der enthaltenen Elemente geben lassen. In diese Liste fügt man dann die Elemente ein, die in der VBox enthalten sein sollen. Ein zweiter Hinweis ist, dass man beim Erzeugen einer Scene neben dem Oberflächenelement (im Beispiel das VBox-Objekt) auch die Breite (im Beispiel 240) und Höhe (im Beispiel 40) der Scene angeben kann. Wenn man die Breite und Höhe nicht angibt, was auch möglich ist, dann berechnet JavaFX die Größe des Fensters, das die Scene umgibt, so, dass die Scene nach Möglichkeit vollständig dargestellt werden kann. Eine letzte Bemerkung betrifft die Beziehung zwischen *launch* und *start*. Man kann dies teilweise mit *start* und *run* bei den Threads vergleichen: Wir wenden auf ein Thread-Objekt in unserem Code *start* an, damit die Methode *run* in einem separaten Thread ausgeführt wird. Ähnlich ist es hier: Wir rufen *launch* im Main-Thread auf. Unsere selbst geschriebene Methode *start* wird dann aber in einem anderen Thread ausgeführt (dazu später mehr in Abschnitt 4.5). Anders als beim Aufruf von *start* der Klasse Thread ist aber *launch* blockierend; aus *launch* kehrt man also erst zurück, nachdem das Fenster geschlossen wurde.

### 4.1.3 Ereignisbehandlung

Ein Fenster dient nicht nur der Ausgabe von Informationen wie in unserem ersten Beispiel, sondern bietet im Allgemeinen für eine Benutzerin Möglichkeiten zur Eingabe und damit zur Interaktion. Die Aktionen einer Benutzerin (Eingaben über Tastatur, Mausbewegungen, Mausklicks usw.) werden Ereignisse genannt. Die Reaktion des Programms auf solche Ereignisse wird Ereignisbehandlung genannt. Bevor wir uns dazu ein JavaFX-Beispiel ansehen, soll das Prinzip der Ereignisbehandlung an einem einfacheren Beispiel erläutert werden. Wir bleiben dazu in der Metapherwelt des Theaters, die JavaFX durch Stage und Scene vorgibt. Angenommen, wir wollen in einem Programmcode nachbilden, dass ein Schauspieler im Laufe seines Lebens unterschiedliche Rollen spielen kann. Wenn er in der Rolle Hamlet ist, reagiert er auf das Ereignis, dass der Regisseur „Action“ brüllt, in einer ganz bestimmten Weise. Wenn er später die Rolle des Faust spielt, reagiert er auf dasselbe

Signal des Regisseurs völlig anders. Wie er reagiert, hängt von der Rolle ab, die er gerade spielt. Im Programmcode aus Listing 4.2 wird die Idee einer Rolle als eine funktionale Schnittstelle repräsentiert. Implementierungen dieser Schnittstelle sind dann konkrete Rollen wie Hamlet und Faust, die in diesem Fall auch als Lambda-Ausdrücke angegeben werden können. Einem Schauspieler (Actor) kann über die Methode setOnAction eine konkrete Rolle zugewiesen werden, die er spielt, wenn jemand action auf ihn anwendet. Die Rolle kann im Laufe des Lebens eines Schauspielers mehrfach wechseln.

**Listing 4.2**

```
interface Role
{
    public void play();
}

class Hamlet implements Role
{
    public void play()
    {
        System.out.println("Sein oder Nichtsein, das ist hier die Frage");
    }
}

class Faust implements Role
{
    public void play()
    {
        System.out.println("Da steh' ich nun, ich armer Tor, " +
                           "und bin so klug als wie zuvor!");
    }
}

class Actor
{
    private Role role;

    public void setOnAction(Role role)
    {
        this.role = role;
    }

    public void action()
    {
        if(role != null)
        {
            role.play();
        }
    }
}

public class Theater
{
    public static void main(String[] args)
    {
        Actor actor = new Actor();
        actor.action(); //es passiert nichts
```

```

        actor.setOnAction(new Hamlet());
        actor.action(); //Ausgabe: Sein oder Nichtsein ...
        actor.setOnAction(new Faust());
        actor.action(); //Ausgabe: Da steh' ich nun ...

        actor.setOnAction(()->System.out.println("Sein oder ..."));
        actor.action(); //Ausgabe: Sein oder Nichtsein ...
        actor.setOnAction(()->System.out.println("Da steh' ich nun ..."));
        actor.action(); //Ausgabe: Da steh' ich nun ...
    }
}

```

Die Ereignisbehandlung in JavaFX ist dieser Schauspielermetapher recht ähnlich. In JavaFX entspricht ein *Button* einem Schauspieler. Das Brüllen von „Action“ durch den Regisseur übernehmen wir, indem wir mit der Maus auf den Button klicken. Das Pendant zur Methode *action* und deren Aufruf befindet sich in der JavaFX-Bibliothek und ist nicht Teil unseres eigenen Programmcodes wie in Listing 4.2. Wie der Button bei einem Buttonklick reagiert, kann wie oben durch Aufruf der Methode *setOnAction* festgelegt werden. Damit können wir nun relativ leicht das nächste JavaX-Beispiel aus Listing 4.3 verstehen: Die Oberfläche besteht aus einem Label und zwei Buttons, die alle wie zuvor mit Hilfe einer VBox untereinander angeordnet sind. Das Label zeigt den Wert eines Zählers an (initial 0). Mit einem der Buttons kann man den Zähler um eins erhöhen, mit dem anderen Button auf seinen Anfangswert 0 zurücksetzen.

### **Listing 4.3**

```

import javafx.application.Application;
import javafx.stage.Stage;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;

public class Counter extends Application
{
    private Label label;
    private int counter;

    private void increment()
    {
        counter++;
        label.setText("" + counter);
    }

    private void reset()
    {
        counter = 0;
        label.setText("" + counter);
    }

    public void start(Stage primaryStage)
    {
        label = new Label();
        Button b1 = new Button("Erhöhen");
        b1.setOnAction(e -> increment());
        Button b2 = new Button("Zurücksetzen");

```

```

        b2.setOnAction(e -> reset());
        VBox root = new VBox();
        root.getChildren().add(label);
        root.getChildren().add(b1);
        root.getChildren().add(b2);
        reset();
    }

    Scene scene = new Scene(root, 200, 70);
    primaryStage.setScene(scene);
    primaryStage.setTitle("Zähler");
    primaryStage.show();
}

public static void main(String[] args)
{
    launch(args);
}
}

```

Bild 4.2 zeigt das Fenster, nachdem der Zähler einige Male erhöht wurde.



**Bild 4.2**

Fenster zum Zählerbeispiel (Listing 4.3)

Die Methode `setOnAction` der Klasse `Button` hat einen Parameter des Typs `EventHandler<ActionEvent>`. `EventHandler` ist eine generische Schnittstelle, die wie folgt definiert ist:

```

public interface EventHandler<T extends Event> extends EventListener
{
    public void handle(T event);
}

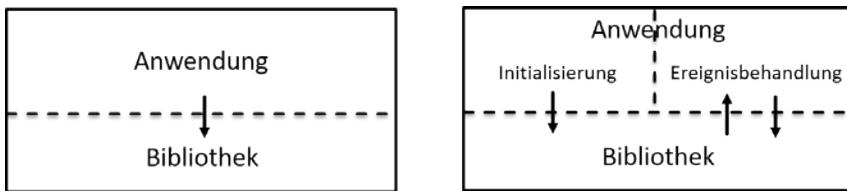
```

Das heißt also, dass man die Methode `handle` mit einem Parameter des Typs `ActionEvent` implementieren muss. Verwendet man dazu einen Lambda-Ausdruck, was möglich ist, weil `EventHandler` eine funktionale Schnittstelle ist, dann muss der Parameter links vom Pfeil angegeben werden, wobei die Typangabe in Listing 4.3 weggelassen wurde. Man hätte den Typ des Parameters aber auch mit angeben können (dann sind allerdings Klammern obligatorisch):

```
b1.setOnAction((ActionEvent e) -> increment());
```

In Listing 4.3 wurde der `ActionEvent`-Parameter im Rumpf der implementierten Methode `handle` (rechts vom Pfeil) nicht verwendet. Im Allgemeinen kann man über einen Event-Parameter weitere Auskünfte über das eingetretene Ereignis abfragen (bei Mausereignissen zum Beispiel die Koordinaten des Mauszeigers).

Die hier beispielhaft implementierte Ereignisbehandlung beinhaltet eine Umkehrung von üblichen Nutzungsbeziehungen. Am Anfang der Programmierausbildung lernt man, wie man einfache, eigene Anwendungen entwickelt und dazu Methoden einer vorhandenen Bibliothek nutzt (zum Beispiel zur Ausgabe auf die Console, zum Erzeugen von Listen und zum Hinzufügen von Elementen in diese). Auch in Kapitel 3 dieses Buches wurden Klassen und Methoden der Concurrent-Bibliothek aus unseren eigenen Anwendungen heraus genutzt (Semaphore, Atomic-Klassen, Locks und Conditions, CompletableFutures usw.). In diesen Fällen werden Methoden, die in einer Bibliothek implementiert sind, aus dem eigenen Programmcode heraus aufgerufen. Da man in einer Schichtendarstellung die eigene Anwendung oberhalb der Bibliothek zeichnet, ist die Aufrufrichtung von oben nach unten, wie das in Bild 4.3 in der linken Hälfte zu sehen ist. Man spricht aus diesem Grund auch von Downcalls. Bei grafischen Benutzeroberflächen kann man eine Anwendung in zwei Teile teilen: in einen Initialisierungsteil (in Listing 4.3 besteht dieser aus der Methode start) und einen Teil für die Ereignisbehandlung (in Listing 4.3 besteht dieser aus den Methoden reset und increment). Während der Initialisierungsteil ebenfalls Downcalls verwendet (zum Beispiel Aufrufe der Methoden setOnAction und show), ist es bei der Ereignisbehandlung umgekehrt: Eigene Methoden wie increment und reset werden beim Eintreten bestimmter Ereignisse vom Code der Bibliothek aufgerufen. Man spricht von Upcalls. In den aufgerufenen Methoden wird in der Regel wieder auf Bibliotheksmethoden zugegriffen (im Beispiel oben war das `setText`), so dass während der Ereignisbehandlung wieder Downcalls stattfinden. Diese Situation ist in Bild 4.3 auf der rechten Seite dargestellt.



**Bild 4.3** Downcalls und Upcalls

Nachdem nun der Einstieg in JavaFX geschafft ist, wenden wir uns den sogenannten Properties zu. Ohne ein Verständnis der Properties ist der Zugang zu einem großen Teil von JavaFX nicht möglich.

## ■ 4.2 Properties, Bindings und JavaFX-Collections

### 4.2.1 Properties

Die zuvor vorgestellte Ereignisbehandlung für Buttons in JavaFX ist eine Konkretisierung eines allgemeineren Prinzips, das man als Entwurfsmuster *Observer* (Beobachter) bezeichnet. In der Welt dieses Entwurfsmusters gibt es beobachtbare Objekte und Beobachter. Beobachter können sich an beobachtbaren Objekten anmelden. Wenn dann ein bestimmtes



```
public class PropertyWithListeners
{
    public static void main(String[] args)
    {
        SimpleIntegerProperty prop = new SimpleIntegerProperty();
        MySimpleChangeListener listener = new MySimpleChangeListener();
        prop.addListener(listener);
        for(int i = 1; i <= 20; i++)
        {
            int newValue = (int)(Math.random()*10) - 5;
            System.out.println("Änderung: " + newValue);
            prop.set(newValue);
        }
    }
}
```

Der Methode *changed*, die bei einer Änderung der Property aufgerufen wird, werden drei Parameter übergeben: einmal die Property selbst, die sich geändert hat (*ObservableValue* ist eine Schnittstelle, die von *SimpleIntegerProperty* implementiert wird), sowie der alte und der neue Wert. Die Werte sind in diesem Fall vom Typ *Number*. *Number* ist eine Oberklasse, aus der die Java-Wrapper-Klassen wie beispielsweise *Integer*, *Long* und *Double* abgeleitet sind.

Selbstverständlich hätte man den Listener auch hier wieder als Lambda-Ausdruck angeben können:

```
prop.addListener((observable, oldValue, newValue)
    -> System.out.println(">>>geändert von " + oldValue +
        " zu " + newValue));
```

Eine spezielle Variante von Properties sind *ReadOnlyProperties*. Diese scheinen im ersten Augenblick sinnlos zu sein, denn wenn eine Property nur gelesen und nicht geändert werden kann, dann macht die Anmeldung eines Listeners wenig Sinn, da dieser dann ja nie benachrichtigt würde. Tatsächlich ist es aber so, dass JavaFX sogenannte *ReadOnlyWrapper*-Klassen bereitstellt. Von einem Wrapper kann man sich eine *ReadOnlyProperty* geben lassen. Der relevante Attributwert kann über die *ReadOnlyProperty* nicht geändert werden, wohl aber über den Wrapper. Man hat damit die Möglichkeit, dass man einen bestimmten Wert nur selber ändern kann, weil man den Wrapper nicht nach außen gibt, dass man aber anderen die Möglichkeit gibt, diese Änderungen über eine *ReadOnlyProperty* zu beobachten. Auch hier gibt es für die verschiedenen Datentypen entsprechende *ReadOnlyWrapper*- und *ReadOnlyProperty*-Klassen wie z.B. *ReadOnlyIntegerWrapper* und *ReadOnlyIntegerProperty* oder *ReadOnlyBooleanWrapper* und *ReadOnlyBooleanProperty*. Die Wrapper-Klassen sind aus der jeweiligen *SimpleProperty*-Klasse abgeleitet. Die folgenden Codezeilen beschreiben die grundsätzliche Idee noch einmal:

```
ReadOnlyIntegerWrapper wrapper = new ReadOnlyIntegerWrapper();
ReadOnlyIntegerProperty readonly = wrapper.getReadOnlyProperty();
readonly.addListener((o, oldValue, newValue)
    -> System.out.println("!!!geändert von " + oldValue +
        " zu " + newValue));
//über Wrapper kann Wert geändert werden:
wrapper.set(newValue);
//aber nicht über ReadOnlyProperty:
```

```
//readonly.set(newValue); => Syntaxfehler, da es Methode set nicht gibt
```

Properties und ReadOnlyProperties sind für JavaFX deshalb sehr bedeutend, weil jedes JavaFX-Element solche Properties besitzt. Bereits ein relativ einfaches JavaFX-Element wie ein Label hat eine ganze Reihe solcher Properties (aufgrund der Vererbungsstruktur sind das schon mehr als 100), darunter eine StringProperty für den angezeigten Text, eine BooleanProperty für das Unterstreichen des Texts, eine ObjectProperty für den Font und eine weitere ObjectProperty für die Ausrichtung des Textes. Mit Hilfe der Property für den Text beispielsweise kann man sich benachrichtigen lassen, wenn sich der Text des Labels verändert hat. Über diese Property kann auch die Beschriftung des Labels geändert werden:

```
Label label = new Label();
label.textProperty().set("Hallo Welt");
```

In unserem bisherigen Beispiel wurde die Beschriftung eines Labels mit der Methode *setText* geändert. Dies ist die einfachere und naheliegendere Möglichkeit, wenn man sich nicht mit Properties herumschlagen will. Aber auch *setText* bewirkt eine Änderung der entsprechenden Text-Property. Dieses Prinzip gilt für alle Eigenschaften aller JavaFX-Elemente: Man kann sich die entsprechende Property geben lassen und darüber lesen und schreiben (falls es keine ReadOnlyProperty ist). Alternativ gibt es in vielen Fällen auch die entsprechenden Getter- und Setter-Methoden wie z.B. *getText* und *setText* bei Label. Die Nutzung von Properties ist immer dann notwendig, wenn man mit Listenern arbeiten möchte oder mit Bindings, wovon im Folgenden die Rede sein wird.

## 4.2.2 Bindings

Bindings sind ein Mechanismus, um die Werte von Properties zu koppeln, so dass eine Wertänderung bei der einen Property eine entsprechende Wertänderung bei der anderen Property bewirkt. Über Listener könnte man das auch selbst programmieren: Man meldet einen Listener bei der einen Property an. Der Listener funktioniert so, dass er bei jeder Benachrichtigung den Wert der anderen Property auf den gemeldeten neuen Wert setzt. Bindings nehmen einem diese Arbeit ab und haben eine ganze Reihe weiterer Funktionen.

Man unterscheidet zwischen einer unidirekionalen und einer bidirektonalen Kopplung. Im unidirektonalen Fall kann man den Wert der einen Property ändern, worauf sich auch der Wert in der zweiten Property ändert. Der Versuch, den Wert der zweiten Property aber direkt zu ändern, löst eine Ausnahme aus. Erst nach der Entkopplung kann man auch den Wert der zweiten Property wieder ändern:

```
SimpleIntegerProperty prop1 = new SimpleIntegerProperty();
SimpleIntegerProperty prop2 = new SimpleIntegerProperty();
prop2.bind(prop1); //prop2 ist an prop1 gekoppelt (nicht umgekehrt)
System.out.println(prop1.get() + " / " + prop2.get()); // => 0 / 0
prop1.set(101);
System.out.println(prop1.get() + " / " + prop2.get()); // => 101 / 101
try
{
    prop2.set(49);
}
```

```

catch(Exception e)
{
    System.out.println("prop2.set: das funktioniert nicht");
}
prop2.unbind();
prop2.set(73);
System.out.println(prop1.get() + " / " + prop2.get()); // => 101 / 73

```

Bei der bidirektionalen Kopplung kann man die Werte über beide Properties ändern. Die Werte in den beiden Properties sind immer dieselben:

```

SimpleIntegerProperty prop1 = new SimpleIntegerProperty();
SimpleIntegerProperty prop2 = new SimpleIntegerProperty();
prop1.bindBidirectional(prop2); //oder prop2.bindBidirectional(prop1);
System.out.println(prop1.get() + " / " + prop2.get()); // => 0 / 0
prop1.set(101);
System.out.println(prop1.get() + " / " + prop2.get()); // => 101 / 101
prop2.set(49);
System.out.println(prop1.get() + " / " + prop2.get()); // => 49 / 49

```

Unter bestimmten Bedingungen ist auch eine Kopplung von Properties unterschiedlicher Typen möglich (z.B. Integer- mit DoubleProperty oder Integer- mit StringProperty). Auch kann man ganze Kopplungsnetze erzeugen, die Rechenoperationen wie Additionen und Multiplikationen enthalten. Als ein einfaches derartiges Kopplungsnetz kann man sich mehrere IntegerProperties vorstellen, die über eine Multiplikation verknüpft werden. Der Ausgang dieser Multiplikation wird an eine weitere IntegerProperty gekoppelt. Diese zuletzt genannte IntegerProperty enthält dann immer das Produkt der Werte aller anderen Properties.

In gewisser Weise kann man sich die Kopplung von Properties wie bei der Tabellenkalkulation vorstellen, deren prominentester Vertreter sicher Excel ist. Hier kann man für Felder Formeln definieren. Der Wert eines solchen Felds ist von den Inhalten anderer Felder abhängig, wobei diese Felder wiederum von anderen Feldern abhängig sein können usw. Wie bei der Tabellenkalkulation führen Abhängigkeitszyklen auch bei den Bindings zu Fehlern. In JavaFX kann man mit Hilfe der Kopplung von Properties der JavaFX-Elemente erreichen, dass sich eine Änderung eines Elements automatisch auch auf andere Elemente auswirkt. Ein einfaches, wenn auch nicht besonders sinnvolles Beispiel wäre die Kopplung der Text-Property eines Labels an die Text-Property eines Eingabefeldes:

```

Label output = new Label();
TextField input = new TextField();
output.textProperty().bind(input.textProperty());

```

So sieht man alles, was man in das Eingabefeld tippt, nicht nur im Eingabefeld selber, sondern unmittelbar bei jedem Tastendruck auch im Label.

### 4.2.3 JavaFX-Collections

Unter Collections versteht man in Java eine Sammlung von Schnittstellen und Implementierungen, um mehrere Elemente in einem Behälter zu verwalten. Im Wesentlichen sind die Behältersorten Listen, Mengen und Hash-Tabellen (in unterschiedlichen Varianten). Die Collections von JavaFX erweitern nun die Collections von Java um die Property-Charakteristik. Das heißt, dass man bei den Collections somit auch Beobachter anmelden kann, um sich über Änderungen benachrichtigen zu lassen. Änderungen sind in diesem Fall das Hinzufügen, Ersetzen, Entfernen oder Vertauschen von Elementen.

Die Schnittstelle *ObservableList* erweitert die Schnittstelle List aus dem Java-Collections-Framework so, dass man daran Listener anmelden kann (naheliegender wäre es, wenn man an einer ObservableList einen Observer anmelden könnte). Analog wird *ObservableMap* aus Map und *ObservableSet* aus Set abgeleitet.

Diese Collections werden in JavaFX von den Container-Elementen, die ein bestimmtes Layout für die in ihr enthaltenen Elemente realisieren, verwendet. Wir haben als einen solchen Container schon die Klasse VBox kennengelernt. Es wurde auch erwähnt, dass man zu VBox nicht direkt etwas hinzufügen kann, sondern dass man sich über die Methode getChildren erst eine Liste geben lassen muss, in die man die Elemente setzt. Die Methode getChildren liefert eine ObservableList zurück. Wenn man möchte, kann man so auf Veränderungen der Inhalte einer VBox reagieren.

## ■ 4.3 Elemente von JavaFX

Wie zu Beginn dieses Kapitels erläutert wurde, besitzt bei grafischen Benutzeroberflächen der Inhalt eines Fensters eine baumartige Struktur. Die Elemente, die Nachfolger im Baum haben, sind Container-Elemente, die ein bestimmtes Layout für die darin enthaltenen Elemente realisieren. Labels und Buttons beispielsweise sind Interaktionselemente, die keine Nachfolger im Baum haben. In diesem Abschnitt wird ein Überblick über die wichtigsten Container-Typen und Interaktionselemente gegeben. Außerdem gehen wir noch auf die Grafikprogrammierung ein und legen dar, was JavaFX außerdem noch zu bieten hat.

### 4.3.1 Container

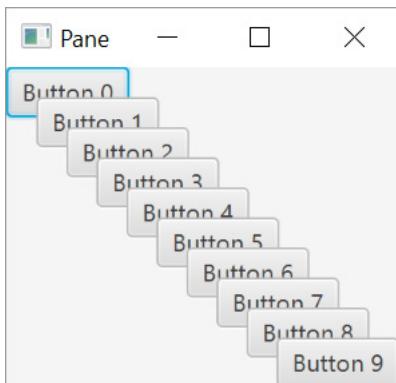
Sowohl die Klassen für die Container als auch für die Interaktionselemente haben als Basisklasse die Klasse *Node*. Hieraus ist indirekt (also über mehrere Ableitungsstufen) die Klasse *Pane* abgeleitet. Sie ist die Basisklasse für alle Container-Klassen mit der Besonderheit, dass sie kein Layout realisiert. Wenn man also seine Elemente selber positionieren möchte, dann kann man dies mit dem Container-Typ *Pane* verwirklichen, indem man allen in der *Pane* enthaltenen Elementen über *setLayoutX* und *setLayoutY* ihre Position zuweist. Die folgenden Programmzeilen demonstrieren das Gesagte:

```

Pane root = new Pane();
for(int i = 0; i < 10; i++)
{
    Button b = new Button("Button " + i);
    b.setLayoutX(i * 15);
    b.setLayoutY(i * 15);
    root.getChildren().add(b);
}

```

Die linke, obere Ecke der Pane hat die Koordinaten (0, 0). Die x-Achse verläuft in positiver Richtung nach rechts, die y-Achse in positiver Richtung nach unten. Durch die Angabe der Position eines Elements (im Beispiel eines Buttons) wird die linke, obere Ecke dieses Elements relativ zum Container, in dem sich dieses Element befindet, festgelegt. Im Beispiel hat das erste Element die Position (0, 0). Der Button befindet sich also genau in der oberen, linken Ecke der Pane. Die darauf folgenden Buttons sind jeweils 15 Pixel weiter rechts und weiter unten zu finden. Sie liegen jeweils über dem weiter links oben stehenden Button, weil sie später in die Pane eingefügt werden. Somit sieht das Fenster so wie in Bild 4.4 aus.



**Bild 4.4**

Beispiel für selbst definiertes Layout mit einer Pane

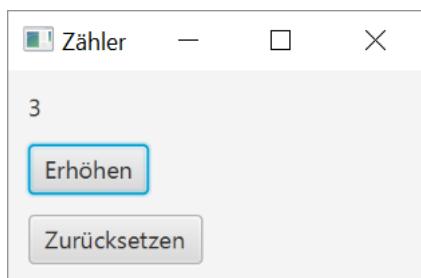
Wenn man das Fenster vergrößert oder verkleinert, ändert sich weder an der Position noch an der Größe der Buttons etwas. Im Falle einer Vergrößerung bekommt man mehr Freifläche, bei einer Verkleinerung sind nicht mehr alle Buttons sichtbar. Man könnte aber sein eigenes Layout auch so programmieren, dass man auf eine Änderung der Größe des Containers durch Änderung der Position und/oder der Größe der enthaltenen Elemente reagiert. Das Verändern der Größe eines Containers kann man deshalb immer leicht verfolgen, weil natürlich Breite und Höhe eines Containers Properties sind, in diesem Fall `ReadOnlyDoubleProperties`.

Aus der Klasse `Pane` ohne Layout sind mehrere Container-Klassen abgeleitet, die alle ein bestimmtes Layout für die enthaltenen Elemente realisieren:

- **VBox:** Die enthaltenen Elemente werden untereinander (vertikal) angeordnet.
- **HBox:** Die enthaltenen Elemente werden nebeneinander (horizontal) angeordnet.
- **FlowPane:** Die Elemente werden wie in einem Fließtext angeordnet. Das heißt, man positioniert so viel wie möglich in eine Zeile. Wenn es nicht mehr passt, setzt man die folgenden Elemente in die nächste Zeile.

- **BorderPane**: Es werden fünf Bereiche definiert, in die Elemente gesetzt werden können: oben, unten, links, rechts und zentral. Dafür bietet die Klasse spezielle Methoden an, um die Position anzugeben: `setTop`, `setBottom`, `setLeft`, `setRight` und `setCenter`.
- **StackPane**: Alle Elemente werden übereinander gesetzt.
- **TilePane**: Die Elemente sind in einer Gitterstruktur angeordnet. Alle Gitterelemente sind gleich groß.
- **GridPane**: Wie bei einer TilePane sind die Elemente in einem Gitter angeordnet. Es gibt aber viele Variationsmöglichkeiten für die Gitterstruktur. Nicht alle Spalten müssen die gleiche Breite haben. Dasselbe gilt für die Höhe der Zeilen. Ein Element in diesem Container kann sich auch über mehrere benachbarte Spalten und/oder Zeilen erstrecken. Die Klasse bietet spezielle Methoden zum Hinzufügen der Elemente an, bei denen die Position (Nummer der Zeile und Nummer der Spalte) und optional auch die Ausdehnung (Anzahl der Zeilen und Anzahl der Spalten) angegeben werden kann.
- **AnchorPane**: Bei diesem Layout kann man die Elemente am Rand des Containers verankern.

Hat man eine Layout-Klasse ausgewählt, dann kann man das Layout der Elemente noch auf viele weitere Arten beeinflussen. Wir wollen hierzu beispielhaft einige Hinweise für das einfache vertikale Layout geben, das durch VBox realisiert wird. So kann man beispielsweise die Zwischenräume der Elemente und einen Abstand der Elemente vom oberen, unteren, linken und rechten Rand festlegen. Wenn man Bild 4.5 mit Bild 4.2 vergleicht, dann sieht man hierfür ein Beispiel.

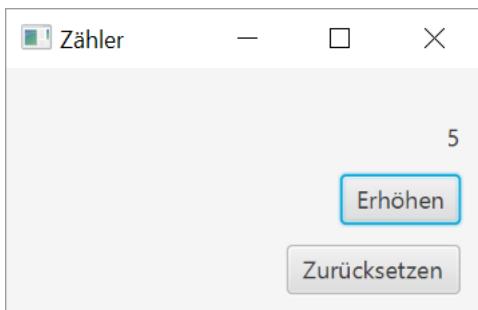
**Bild 4.5**

Fenster zum Zählerbeispiel mit Zwischenräumen und Abständen vom Rand

Das Layout in Bild 4.5 ergibt sich übrigens nicht automatisch dadurch, dass man das Fenster aus Bild 4.2 vergrößert. Bei einer Vergrößerung sieht man lediglich mehr freie Fläche rechts und unten, die Elemente bleiben an ihrer Position. Um das Aussehen von Bild 4.5 zu erhalten, wurde der Programmcode zur Festlegung des Zwischenraums und des Abstands vom Rand entsprechend erweitert:

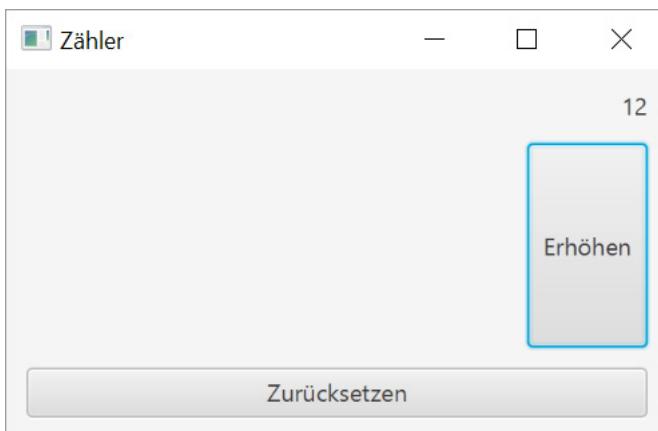
```
VBox root = new VBox(10); //Parameter: Raum zwischen den Elementen der VBox
Insets padding = new Insets(10); //"Füllmaterial" zur Abstandshaltung
root.setPadding(padding); //Abstand vom Rand
```

Wie schon erwähnt wurde, ändert sich die Position und die Größe der Elemente beim Vergrößern und Verkleinern des Fensters nicht. Dies liegt u. a. daran, dass die Standardausrichtung einer VBox links oben ist. Wenn man dies zum Beispiel auf rechts unten ändert, dann wandern unsere drei Elemente bei einer Änderung der Fenstergröße immer relativ zum rechten, unteren Rand mit, wie dies in Bild 4.6 zu sehen ist.

**Bild 4.6**

Fenster mit vertikalem Layout und einer Ausrichtung nach rechts unten

Bisher haben wir uns noch nicht darum gekümmert, dass sich die Größe der Elemente an den zur Verfügung stehenden Platz anpasst. Aber auch hierfür gibt es Einstellmöglichkeiten. In Bild 4.7 ist das Ergebnis zu sehen, wenn im Programmcode vorgegeben wird, dass einer der Buttons immer die zur Verfügung stehende Höhe und der andere Button die zur Verfügung stehende Breite ausnutzt.

**Bild 4.7** Fenster mit Buttons, die die verfügbare Höhe bzw. Breite nutzen

Mit diesen wenigen Hinweisen wollen wir das Thema Layout beenden und uns einen Überblick über die Interaktionselemente von JavaFX verschaffen.

### 4.3.2 Interaktionselemente

Wie die Container-Klassen haben alle Klassen der Interaktionselemente die Basisklasse `Node`. Über mehrere Stufen ist aus `Node` die Klasse `Control` abgeleitet. Alle Klassen der Interaktionselemente sind direkt oder indirekt aus `Control` abgeleitet. Wir haben bereits `Labels`, `Buttons` und ganz rudimentär auch schon `TextField` kennengelernt. Hier ist ein Überblick über die wichtigsten Interaktionselemente von JavaFX:

- **Labeled-Klassen:** `Labeled` ist eine abstrakte Basisklasse für alle Interaktionselemente, die Text anzeigen. Entsprechend gibt es eine ganze Reihe entsprechender Properties wie den

Text selbst, dessen Ausrichtung (linksbündig, zentriert, rechtsbündig), den Zeichensatz und die boolesche Eigenschaft, ob der Text unterstrichen ist oder nicht. Aus Labeled ist *Label* abgeleitet. Außerdem werden aus Labeled über eine oder mehrere Stufen sogenannte Command Buttons und Choice Buttons abgeleitet.

Zu den Command Buttons gehört neben der schon bekannten Klasse *Button* auch *Hyperlink*. Wie bei Button kann man beim Klicken auf einen Hyperlink eine Aktion auslösen, wobei das Aussehen dieses Elements, wie der Klassenname andeutet, einer Linkdarstellung in einem Browser nachempfunden ist.

Zu den Choice Buttons zählen u. a. *CheckBox*, *ToggleButton* und *RadioButton*. Diese Klassen stellen Interaktionselemente dar, die aus- und wieder abgewählt werden können. Der Selektionszustand wird durch eine boolesche Property repräsentiert. Eine CheckBox in JavaFX kann zwei oder optional sogar drei Zustände besitzen. Der dritte Zustand stellt den Zustand unbestimmt dar. Über eine boolesche Property wird festgelegt, ob der dritte Zustand erlaubt ist, und über eine weitere boolesche Property, ob sich die CheckBox in diesem Zustand befindet. Der Wert der Selektions-Property hat nur dann eine Bedeutung, wenn der Zustand nicht unbestimmt ist. Optisch besitzt eine CheckBox neben einem Text eine quadratische Fläche, in der ein Häkchen im ausgewählten Zustand zu sehen ist. Der unbestimmte Zustand wird durch ein Minuszeichen in der quadratischen Fläche symbolisiert. Wenn man auf den Selektionsstatus einer CheckBox im Programm unmittelbar reagieren möchte und nicht den Status zu einem späteren Zeitpunkt einfach nur abfragen möchte, dann kann man einen *ChangeListener* an der booleschen Selektions-Property anmelden.

ToggleButtons und RadioButtons sind einander ähnlich (*RadioButton* ist aus *ToggleButton* abgeleitet). Auch sie sind Elemente, die selektiert oder deseletiert werden können, unterscheiden sich aber in ihrem Aussehen. ToggleButtons sehen aus wie Buttons. Sie bleiben nach dem Anklicken im gedrückten Zustand, bis sie erneut angeklickt werden, während RadioButtons neben dem Text einen Kreis haben, in dem sich im ausgewählten Zustand ein Punkt befindet. Sowohl ToggleButtons als auch RadioButtons können in einer *ToggleGroup* zusammengefasst werden. Dies bewirkt, dass nur höchstens ein Element der Gruppe selektiert sein kann. Wird ein Element der Gruppe selektiert, so wird ein eventuell schon selektiertes Element der Gruppe deseletiert.

- Auswahlelemente: Über mehrere ToggleButtons oder RadioButtons, die in einer ToggleGroup zusammengefasst sind, kann man eine von mehreren Optionen auswählen. Über mehrere CheckBoxes ist die Realisierung einer Mehrfachauswahl möglich. Wenn es sehr viele Auswahloptionen gibt, sind diese Realisierungsformen aber nicht mehr geeignet. Deshalb gibt es mehrere Elemente, die Einfach- und Mehrfachauswahlmöglichkeiten bieten. Dazu gehören ChoiceBox, ComboBox, ListView, DatePicker, ColorPicker und FileChooser.

Eine *ComboBox* ist eine etwas mächtigere Variante einer *ChoiceBox*. Mit einer ComboBox kann eine Einfachauswahl in sehr platzsparender Weise realisiert werden. Bei einer ComboBox sieht man nämlich immer nur das ausgewählte Element; die anderen werden nur ausgeklappt, wenn die ComboBox angeklickt wird. Wenn auch in der ComboBox nur Texte angezeigt werden, so können dennoch nicht nur Strings hinzugefügt werden, sondern Objekte jeder beliebigen Klasse (ComboBox ist eine generische Klasse, wobei der Typparameter dem Typ der einzufügenden Elemente entspricht). Die in der ComboBox

angezeigten Texte sind in der Regel die von der Methode `toString` zurückgelieferten Strings. Für das ausgewählte Element gibt es eine Property. Will man auf die Änderung der Auswahl reagieren, meldet man einen ChangeListener an dieser Property an.

Auch `ListView` ist eine generische Klasse. Wie viele der zur Auswahl stehenden Elemente angezeigt werden, hängt davon ab, wie viel Platz der ListView eingeräumt wird. Können nicht alle Elemente angezeigt werden, so steht ein Scrollbar (Rollbalken) zur Verfügung, mit dem man den angezeigten Ausschnitt der Liste verschieben kann. Die ListView kann auf Einfach- oder Mehrfachauswahl eingestellt werden.

Wie die Namen der Klassen verdeutlichen, erlaubt ein `ColorPicker` die Auswahl einer Farbe, ein `DatePicker` die Auswahl eines Datums und ein `FileChooser` die Auswahl einer Datei im Dateibaum.

- Texteingabeelemente: `TextInputControl` ist die abstrakte Basisklasse für die Texteingabeelemente. Diese Klasse enthält u.a. Properties für den Text, den selektierten Bereich des Texts und die Cursor-Position. Daraus abgeleitet sind `TextField` und `TextArea`. `TextField` ist einzeilig. Beim Drücken der Enter-Taste wird ein Action-Ereignis ausgelöst, auf das man reagieren kann, wenn man mit `setOnAction` einen EventHandler<ActionEvent> registriert hat. `TextArea` ist ein mehrzeiliger Eingabebereich. Das Drücken von Enter bewirkt einen Zeilenumbruch. Es wird kein Action-Ereignis ausgelöst. Ein EventHandler kann dement sprechend auch gar nicht angemeldet werden. Die Klasse `PasswordField` ist aus `TextField` abgeleitet. Auch sie stellt ein einzeiliges Eingabefeld dar. Beim Tippen sind aber die eingetippten Zeichen nicht zu sehen. Stattdessen wird ein einstellbares Ersatzzeichen (wie z.B. \*) angezeigt. Wie der Name der Klasse sagt, dient dieses Interaktionselement zum Eingeben von Passwörtern und dergleichen.
- Einstellung numerischer Werte: Der Einstellung und Anzeige numerischer Werte dienen horizontale und vertikale Schieberegler. In JavaFX gibt es hierzu die Klasse `Slider`. Die drei wichtigsten Attribute sind der minimale, maximale und aktuelle Wert, die alle jeweils durch eine DoubleProperty repräsentiert werden. Weitere Properties sind für das optische Erscheinungsbild zuständig wie die Beschriftung der Skala und der Abstand der Markierungspositionen.
- Weitere Elemente: Daneben gibt es zahlreiche weitere Interaktionselemente, darunter zum Beispiel: `ProgressBar` und `ProgressIndicator` zur Fortschrittsanzeige, eine Menüleiste für Menüs mit unterschiedlichen Arten von Menüelementen, eine `TreeView` zur Darstellung einer Baumstruktur, eine `TableView` zur Darstellung einer Tabelle sowie eine `TreeTableView` als Mischform zwischen `TreeView` und `TableView`.

### 4.3.3 Grafikprogrammierung

Unter Grafikprogrammierung verstehen wir das Zeichnen beliebiger Formen in ein Fenster. In JavaFX wird dies erreicht durch Objekte verschiedener Klassen, die alle aus `Shape` abgeleitet sind, wobei `Shape` auch wie alle Container und Interaktionselemente aus `Node` abgeleitet ist. Das Thema Farbe spielt nicht nur bei der Grafikprogrammierung eine Rolle. So kann zum Beispiel auch die Schriftfarbe eines Labels festgelegt werden. In JavaFX wird eine Farbe (Klasse `Color`) verallgemeinert durch die abstrakte Klasse `Paint`. Daraus ist nicht nur `Color` abgeleitet, sondern auch `LinearGradient` und `RadialGradient`, womit man lineare bzw.

radiale Farbübergänge definieren kann. Auch *ImagePattern* für Bildmuster ist aus Paint abgeleitet. An vielen Stellen kann man in JavaFX als Farbe ein Objekt vom Typ Paint angeben, wodurch dann Objekte der genannten Klassen möglich sind. Wir konzentrieren uns hier auf einfache Farben und verwenden hierzu die als Konstanten vordefinierten Color-Objekte wie Color.BLACK, Color.RED, Color.GREEN, Color.BLUE, Color.WHITE usw.

Die aus Node abgeleitete Klasse Shape hat eine ObjectProperty<Paint> für die Linienfarbe, eine ObjectProperty<Paint> für die Füllfarbe im Innern und eine DoubleProperty für die Liniendicke. Wenn ein Objekt nicht ausgefüllt werden soll, erreicht man dies durch Setzen der Füllfarbe auf null. Füllfarben haben nicht für alle Elemente eine Bedeutung (eine Linie kann nicht sinnvoll gefüllt werden). Aus Shape abgeleitet sind:

- *Line* mit DoubleProperties für die x- und y-Koordinaten des Start- und Endpunkts,
- *Rectangle* mit DoubleProperties für die x- und y-Koordinate des linken, oberen Punkts, für die Breite und für die Höhe,
- *Circle* mit DoubleProperties für die x- und y-Koordinate des Mittelpunkts und für den Radius,
- *Ellipse* mit DoubleProperties für die x- und y-Koordinate des Mittelpunkts, für den Radius in x- und y-Richtung,
- *Polyline* mit einer ObservableList<Double>, in der abwechselnd die x- und y-Koordinaten einer Folge von Punkten stehen, die alle durch Linienstücke verbunden sind (für n Punkte hat die Liste  $2 * n$  Elemente, nur eine gerade Anzahl von Elementen in dieser Liste macht aus diesem Grund Sinn),
- *Polygon*, welches eine Polyline darstellt, bei der der letzte und der erste Punkt auch durch eine Linie verbunden sind,
- *Arc*, welches einen Kreisbogen bzw. im ausgefüllten Fall eine Art Kuchenstück eines Kreises bzw. einer Ellipse darstellt,
- *QuadCurve* und *CubicCurve* für Bezier-Kurven mit einem oder zwei Stützpunkten,
- *Text* mit einer StringProperty für den Text, DoubleProperties für die Position des Textes als x- und y-Koordinate, einer Property, welche Stelle des Textes mit der (x, y)-Position gemeint ist (das linke Ende der Grundlinie, das Zentrum usw.), einer Property für den Font, einer für die Ausrichtung usw.

Da Shape-Objekte eine Position haben, macht es wenig Sinn, sie in einen Container mit einem Layout zu setzen, da ein Container die Position entsprechend seines Layouts ändert. Es bietet sich deshalb an, als Container Objekte der Klasse Pane zu nutzen, die kein Layout realisiert. Die Koordinaten beziehen sich auf die linke, obere Ecke der Pane. Die x-Achse verläuft nach rechts, die y-Achse nach unten. Negative x- und y-Werte sind möglich. Die so positionierten Objekte sind dann gar nicht oder nur zum Teil zu sehen. Ebenso verhält es sich mit x- und y-Werten, die größer sind als die aktuelle Breite bzw. Höhe der Pane. Ein Objekt, das so nicht sichtbar ist, kann aber sichtbar werden, wenn das Fenster und damit auch die Pane größer gezogen werden.

In Listing 4.5 befindet sich eine einfache Anwendung, die einige Shape-Objekte (Line, Circle und Rectangle) erzeugt und in einer Pane ablegt.

**Listing 4.5**

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.scene.layout.Pane;
import javafx.scene.paint.Color;
import javafx.scene.shape.*;
import javafx.stage.Stage;

public class SomeShapes extends Application
{
    public void start(Stage primaryStage)
    {
        Pane root = new Pane();
        drawShapes(root);

        primaryStage.setTitle("Einige Formen");
        primaryStage.setScene(new Scene(root, 250, 80));
        primaryStage.show();
    }

    private void drawShapes(Pane root)
    {
        Line line = new Line(40, 10, 10, 40);
        line.setStroke(Color.RED);
        line.setStrokeWidth(5);
        root.getChildren().add(line);

        Circle c1 = new Circle(60, 25, 20);
        c1.setStroke(null);
        c1.setFill(Color.GREEN);
        root.getChildren().add(c1);
        Circle c2 = new Circle(90, 25, 20);
        c2.setStroke(Color.RED);
        c2.setFill(null);
        c2.setStrokeWidth(5);
        root.getChildren().add(c2);

        Rectangle r1 = new Rectangle(120, 10, 40, 40);
        r1.setStroke(null);
        r1.setFill(Color.GREEN);
        root.getChildren().add(r1);
        Rectangle r2 = new Rectangle(170, 10, 40, 40);
        r2.setStroke(Color.RED);
        r2.setFill(null);
        r2.setStrokeWidth(5);
        root.getChildren().add(r2);
    }

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

Das resultierende Fenster zeigt Bild 4.8.

**Bild 4.8**

Beispiel für ein Programm mit einigen Shape-Objekten

Unsere nächste Beispielanwendung wird ein Programm zum Freihandzeichnen sein. Dazu müssen wir die Grafikprogrammierung mit der Reaktion auf Mausereignisse kombinieren. Auf Mausereignisse kann nicht nur im Kontext der Grafikprogrammierung reagiert werden. Bereits in der Basisklasse Node, aus der alle Klassen der Container, der Interaktions-elemente und der Shape-Klassen abgeleitet sind, ist festgelegt, dass man auf Mausereignisse reagieren kann. Wichtige Mausereignisse sind Pressed (Maustaste wurde gedrückt), Released (Maustaste wurde losgelassen), Moved (Maus wurde mit nicht gedrückter Taste bewegt) und Dragged (Maus wurde mit gedrückter Taste bewegt). Mit den entsprechenden Methoden `setOnMousePressed`, `setOnMouseReleased`, `setOnMouseMoved` und `setOnMouseDragged` können entsprechende Listener angemeldet werden. Der ParameterTyp all dieser Methoden ist wie bei `setOnAction` ein EventHandler. In diesem Fall ist der generische Typ allerdings auf `MouseEvent` (oder einer Oberklasse davon) festgelegt. Das heißt, im typischen Fall benötigt man ein Objekt einer Klasse, die eine Methode handle mit dem Parameter `MouseEvent` implementiert. Selbstverständlich kann hierfür wieder ein Lambda-Ausdruck angegeben werden. Vom `MouseEvent`-Objekt kann man alle möglichen Informationen abfragen, insbesondere auch die (x, y)-Koordinate, an der das Mausereignis stattgefunden hat. Hierfür gibt es mehrere Methoden. In einem Fall sind die Koordinaten relativ zu dem Objekt angegeben, an dem der MouseListener angemeldet wurde. Andere Methoden liefern die Koordinaten relativ zur Scene oder sogar relativ zur Bildschirmecke oben links.

Manchmal ist es schwer verständlich, warum denn auf das Klicken eines Buttons nicht auch durch einen MouseListener reagiert wird, sondern durch einen ActionListener. Prinzipiell ist es überhaupt kein Problem, das so zu programmieren. Das Aktivieren eines Buttons stellt jedoch ein Ereignis auf einer höheren Abstraktionsstufe dar als ein Mausereignis, da es auch noch andere Möglichkeiten gibt zur Aktivierung eines Buttons, wie z. B. das Drücken der Enter-Taste in dem Fall, dass der Button gerade den Fokus hat. Ein Action-Ereignis wird immer dann ausgelöst, wenn der Button aktiviert wurde, in welcher Weise das auch immer passiert ist, der MouseListener jedoch nur dann, wenn ein Mausereignis stattgefunden hat.

Im folgenden Beispiel geht es um das Freihandzeichnen mit gedrückter Maustaste. Wir verwenden eine Pane für unsere grafischen Objekte und melden auch an dieser Pane unsere MouseListener an, damit wir die Koordinaten relativ zu dieser Pane erhalten. Wenn die Maustaste gedrückt wird, dann soll die Anwendung so reagieren, als würden wir einen Stift auf ein Blatt Papier setzen. In diesem Fall entsteht bereits ein Punkt. Also müssen wir auf ein `MousePressed`-Ereignis reagieren. Wann immer ein `MouseDragged`-Ereignis gemeldet wird, ziehen wir eine Linie vom letzten uns bekannten Ereignis zu dem Ort, wo das aktuelle Ereignis stattgefunden hat. Die aktuellen Koordinaten liefert der `MouseEvent`, die Koordinaten des letzten Ereignisses merken wir uns in Attributen unserer Klasse. In welchen Abständen die `Dragged`-Ereignisse gemeldet werden, hängt von der Geschwindigkeit der Mausbewegung und auch von der Kapazität des benutzten Computers ab. Beim Loslassen der

Maustaste ist keine zusätzliche Aktion nötig. Deshalb reagieren wir auf MouseReleased-Ereignisse nicht. Um das Gezeichnete auch wieder löschen zu können, gibt es einen Löschen-Button am unteren Rand des Fensters. Wir verwenden deshalb eine BorderPane und setzen in ihr Zentrum die Pane für die Grafikobjekte. Aus Gründen, die später noch deutlich werden, setzen wir in den unteren Teil der BorderPane nicht nur den Löschen-Button, sondern zusätzlich noch ein Label. Button und Label fassen wir in einer HBox zusammen. Alle diese Ideen sind in Listing 4.6 als Programmcode wiederzufinden.

#### Listing 4.6

```
import javafx.application.Application;
import javafx.geometry.Insets;
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.scene.shape.Line;
import javafx.scene.shape.Rectangle;
import javafx.stage.Stage;

public class DrawingLines extends Application
{
    private Pane graphicsPane;
    private double x, y;

    public void start(Stage primaryStage)
    {
        BorderPane root = new BorderPane();
        graphicsPane = new Pane();
        root.setCenter(graphicsPane);
        HBox hbox = new HBox(20);
        Button b = new Button("Löschen");
        hbox.getChildren().add(b);
        hbox.getChildren().add(new Label("Irgendein unwichtiger Text"));

        hbox.setPadding(new Insets(10));
        root.setBottom(hbox);

        graphicsPane.setOnMousePressed
        (
            e -> mousePressed(e.getX(), e.getY())
        );
        graphicsPane.setOnMouseDragged
        (
            e -> mouseDragged(e.getX(), e.getY())
        );
        b.setOnAction
        (
            e -> clear()
        );

        primaryStage.setTitle("Freihandzeichnen");
        primaryStage.setScene(new Scene(root, 330, 300));
        primaryStage.show();
    }

    private void mousePressed(double newX, double newY)
    {
```

```

        x = newX;
        y = newY;
        mouseDragged(x, y);
    }

    private void mouseDragged(double newX, double newY)
    {
        graphicsPane.getChildren().add(new Line(x, y, newX, newY));
        x = newX;
        y = newY;
    }

    private void clear()
    {
        graphicsPane.getChildren().clear();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}

```

Das Zeichnen funktioniert weitgehend wie erhofft. Man kann zum Beispiel mit gedrückter Maustaste auch das Fenster am rechten Rand verlassen und weiterzeichnen. Was man zunächst nicht sieht, wird dann aber sichtbar, wenn man das Fenster verbreitert. Was etwas überrascht, ist die Tatsache, dass man auch in den unteren Bereich, in dem sich die HBox mit Button und Label befindet, hineinzeichnen kann (s. Bild 4.9).



**Bild 4.9** Programm zum Freihandzeichnen

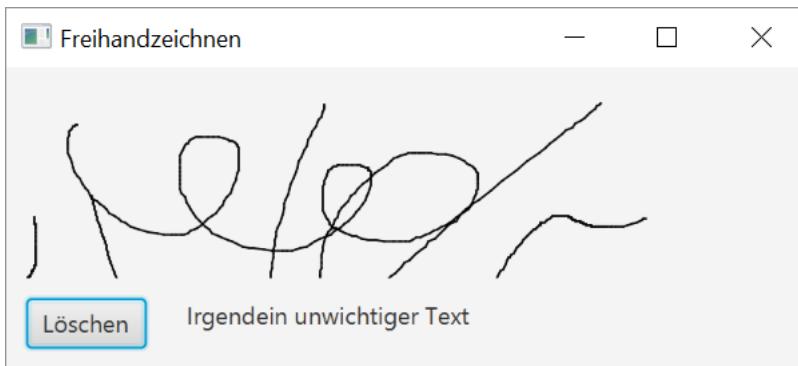
Derselbe Effekt stellt sich ein, wenn man zunächst nur in die Pane zeichnet und dann das Fenster in vertikaler Richtung verkleinert. Eine mögliche Abhilfe schafft ein Clipping-Rechteck (Clipping = Abschneiden), dessen Breite und Höhe an die aktuelle Breite und Höhe der Pane gekoppelt werden. Dies erreicht man durch die folgenden Code-Zeilen, die der Methode start aus Listing 4.6 hinzugefügt werden sollten:

```

Rectangle clipRect = new Rectangle();
clipRect.widthProperty().bind(graphicsPane.widthProperty());
clipRect.heightProperty().bind(graphicsPane.heightProperty());
graphicsPane.setClip(clipRect);

```

Nach dieser Änderung stellt sich der erwünschte Effekt ein, wie in Bild 4.10 zu sehen ist.



**Bild 4.10** Programm zum Freihandzeichnen mit Clipping

#### 4.3.4 Weitere Funktionen von JavaFX

Mit einem Überblick über Container, Interaktionselemente und die Grafikprogrammierung haben wir den Kernbereich von JavaFX überblicksartig kennengelernt. JavaFX bietet aber noch sehr viel mehr. Einiges davon wollen wir hier noch anreisen:

- **Interaktive Erstellung der Oberflächen:** Bisher haben wir in unseren Beispielprogrammen die Oberfläche durch den Programmcode aufgebaut. Alternativ kann man sich auch seine Oberfläche mit dem Tool Scenebuilder „zusammenklicken“ und dann als FXML-Datei abspeichern (FXML ist ein spezielles XML-Format). In seinem Programm kann man dann einer Methode den Namen dieser Datei angeben, worauf diese Datei gelesen und so interpretiert wird, dass die entsprechenden Container und Interaktionselemente erzeugt werden. Es ist sogar möglich, in der FXML-Datei die Methoden für die Ereignisbehandlung zu spezifizieren. Wenn man im Programmcode auf ein bestimmtes Objekt der Oberfläche zugreifen möchte (zum Beispiel auf ein Label zum Ändern des Texts), benötigt man eine Referenz auf dieses Objekt. Da man dieses Objekt nicht selbst erzeugt hat, liegt diese Referenz nicht vor. Es gibt aber eine entsprechende Lookup-Methode, mit Hilfe derer man sich passende Objekte geben lassen kann.
- **Anwendung mit mehreren Fenstern:** Unsere Anwendungen bestehen alle aus lediglich einem Fenster, das nicht in unserem Code erzeugt wird, sondern beim Aufruf der Methode start schon existiert und als Parameter übergeben wird. Es ist aber überhaupt kein Problem, explizit neue Stage-Objekte zu erzeugen, zu befüllen und mit show anzuzeigen. Spezielle Formen von Fenstern sind Dialoge. Diese kann man komplett selbst gestalten. Für viele Fälle, in denen Dialoge zum Anzeigen von Warnungen oder Fehlern oder zum Stellen von Ja-Nein-Fragen („Wollen Sie diese Anwendung wirklich beenden?“) verwendet werden, bietet JavaFX vorgefertigte Dialogklassen wie *Alert* und *ChoiceDialog* an.
- **Drag and Drop:** Auch Drag and Drop kann in JavaFX-Anwendungen durch entsprechende Programmierung eingebaut werden. Drag and Drop ist nicht nur innerhalb der eigenen Anwendung möglich, sondern auch anwendungsübergreifend. Die anderen Anwendun-

gen müssen dabei nicht notwendig Java-Anwendungen sein, sondern man kann Drag and Drop zum Beispiel auch zwischen der eigenen Anwendung und einer gängigen Textverarbeitung realisieren, um somit Textteile zwischen der eigenen Anwendung und der Textverarbeitungsanwendung mit der Maus zu verschieben.

- **Diagramme:** Zur Darstellung von Zahlenreihen bietet JavaFX unterschiedliche Diagrammarten wie Tortendiagramme, Balkendiagramme, Liniendiagramme, Flächendiagramme, Verteilungsdiagramme und Blasendiagramme an.
- **Animationen:** JavaFX umfasst auch eine Reihe von Klassen zur Unterstützung der Programmierung von Animationen.
- **Effekte:** Man kann für seine Interaktionselemente unterschiedliche Effekte wie Spiegelung und Reflexion einstellen. Vermutlich ist das die Begründung für die Bezeichnung JavaFX, denn die Aussprache des englischen Begriffs Effects klingt ähnlich wie die englische Aussprache der Buchstabenfolge FX.
- **CSS:** Die aus der Welt der Browser bekannten Cascading Style Sheets (CSS) können auch für JavaFX-Elemente verwendet werden. Wenn die Oberflächen ein einheitliches Aussehen entsprechend den stilistischen Vorgaben einer Firma oder einer Hochschule (Corporate Design) haben sollen, so kann man dies relativ einfach über CSS erreichen.
- **Transformation.** Auf alle Elemente der Oberfläche lassen sich geometrische Transformationen wie Translation, Rotation, Skalierung und Scherung anwenden. Somit ist es zum Beispiel möglich, einen Button um 90 Grad (inklusive Beschriftung) zu drehen.
- **Einbettung von Web-Seiten, Audios und Videos:** Inhalte von Web-Seiten lassen sich einfach in eine JavaFX-Anwendung einbetten. Auch unterstützt JavaFX das Abspielen von Audios und Videos.

Damit haben wir nun einen ersten, stark komprimierten Eindruck von JavaFX. Unsere Anwendungen waren sehr einfach, so dass wir uns über die Struktur des Programmcodes keine besonderen Gedanken gemacht haben. Im Laufe der Jahre haben sich in der Welt der Informatik einige nützliche Strukturierungsprinzipien herauskristallisiert, die sich bei der Implementierung größerer Anwendungen mit grafischen Benutzeroberflächen inzwischen etabliert haben. Davon wird im folgenden Abschnitt die Rede sein.

## ■ 4.4 MVP

Für Anwendungen mit grafischen Benutzeroberflächen spielen die MV\*-Architekturmuster eine entscheidende Rolle: *MVC (Model – View – Controller)*, *MVP (Model – View – Presenter)* und *MVVM (Model – View – ViewModel)*. Aus meiner Sicht ist es unerheblich, mit welchem dieser Muster sich Anfänger zuerst beschäftigen. Wichtig ist, dass man das Prinzip eines dieser Muster gut versteht und selbst anwenden kann. Eine Umstellung auf ein anderes Muster sollte dann nicht mehr schwer fallen.

Die Kenntnis und das Verständnis dieser Muster sind bei der Software-Entwicklung in dreifacher Hinsicht nützlich:

- Diese Muster helfen, die selbst zu entwickelnde Anwendung klar zu strukturieren. Sie bieten eine Art Leitfaden zur Strukturierung an.
- Mit einer klaren Strukturierung der Software wird das systematische Testen einfacher. Auch Änderungen an der Software sollten leichter machbar sein.
- Es ist wesentlich einfacher, den Code anderer zu verstehen, wenn dieser nach einem dieser Muster gestrickt ist. Man wird sich dann viel schneller in diesem Code zurecht finden und direkt eine Vorstellung davon haben, welche Funktionen die einzelnen Programmteile haben und wie sie zusammenwirken.

In diesem Buch wollen wir uns das MVP-Muster näher ansehen. Nach einer Beschreibung des MVP-Prinzips im Allgemeinen soll dieses anhand eines Beispiels erläutert werden.

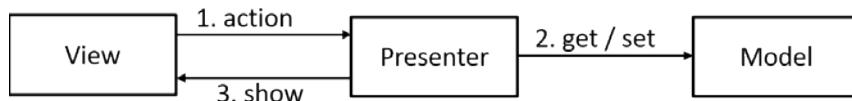
#### 4.4.1 Prinzip von MVP

Mit MVP werden drei Komponenten einer Anwendung bezeichnet: die Modell-, die Darstellungs- und die Präsentationskomponente (Model – View – Presenter). Jede Komponente kann aus einer oder mehreren Java-Klassen bestehen.

- **Model:** Die Modellkomponente ist für die sogenannte Geschäftslogik verantwortlich. Sie bildet den logischen Kern der Anwendung. Sie enthält typischerweise die für die Anwendung relevanten Daten sowie Methoden zum Abfragen und Ändern dieser Daten. Dabei werden die Konsistenzbedingungen gewahrt. Wenn zum Beispiel die Modellkomponente ein Konto repräsentiert, das nicht überzogen werden darf, dann sind die Methoden zum Ändern so ausgelegt, dass der Kontostand nicht negativ wird. Falls eine persistente Speicherung der Daten notwendig ist, so wird dies ebenfalls von der Modellkomponente realisiert, indem sie zum Beispiel auf das Dateisystem oder auf eine Datenbank lesend und schreibend zugreift. Die Modellkomponente muss im Idealfall vollkommen unabhängig von der grafischen Oberfläche sein, so dass sie auch in einem Programm mit einer kommandozeilenbasierten Benutzerschnittstelle verwendet oder auf einen Server ohne Benutzerschnittstelle ausgelagert werden könnte.
- **View:** Die Darstellungskomponente ist – stark verkürzt gesagt – das, was man auf dem Bildschirm als Fenster sieht. Dies ist aber nicht korrekt, denn bei MVP geht es um die Software. Deshalb ist die Darstellungskomponente der Teil der Software, der die Oberfläche aufbaut und verändert. In JavaFX kann dies neben Java-Code auch FXML-Dateien (s. voriger Abschnitt) umfassen. In der Regel enthält die View eine oder mehrere textuelle und grafische Darstellungen der Modellkomponente. Im Allgemeinen muss eine View nicht das gesamte Modell repräsentieren. So ist es z.B. möglich, dass unterschiedliche Teile des Modells durch unterschiedliche Views dargestellt werden. Außerdem ist die Darstellungskomponente dafür verantwortlich, auf Benutzeraktionen zu reagieren und darüber die Präsentationskomponente zu informieren.
- **Presenter:** Die Präsentationskomponente ist für die Ablaufsteuerung zuständig. Ein bestimmter Ablauf wird in der Regel durch eine Benutzeraktion angestoßen, die zunächst von der View-Komponente bearbeitet und an die Presenter-Komponente weitergegeben wird. Eine typischer Ablauf besteht darin, dass der Presenter die nötigen Daten von der View erhält oder sich diese von der View beschafft, die Modellkomponente zur Verarbei-

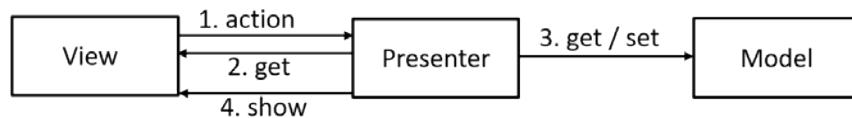
tung dieser Daten beauftragt und abhängig vom Ergebnis dieser Verarbeitung die View anweist, bestimmte Informationen, die von der Modellkomponente zurückgeliefert oder separat abgefragt werden, darzustellen. Wie diese Darstellung im Detail erfolgt, ist allein eine Angelegenheit der Darstellungskomponente.

Das Zusammenspiel der drei Komponenten M, V und P ist schematisch in Bild 4.11 dargestellt. Insbesondere kann an den Pfeilen abgelesen werden, welche Komponente welche andere Komponente kennt. So ist also auch zu sehen, dass sich die M- und V-Komponenten nicht kennen, sondern dass die P-Komponente eine Vermittlerrolle zwischen den beiden anderen einnimmt. Dies ist ein wesentliches Merkmal von MVP (im Gegensatz zu MVC).



**Bild 4.11** Zusammenspiel der Komponenten beim MVP-Architekturmuster

Bild 4.11 sollte wirklich nur schematisch verstanden werden, denn es sind viele Varianten denkbar. So ist in Bild 4.12 eine Variante gezeigt, in der der Presenter sich die nötigen Eingabedaten aktiv von der View beschafft, während in Bild 4.11 davon ausgegangen wird, dass die View alle nötigen Daten beim Aufruf von action liefert.

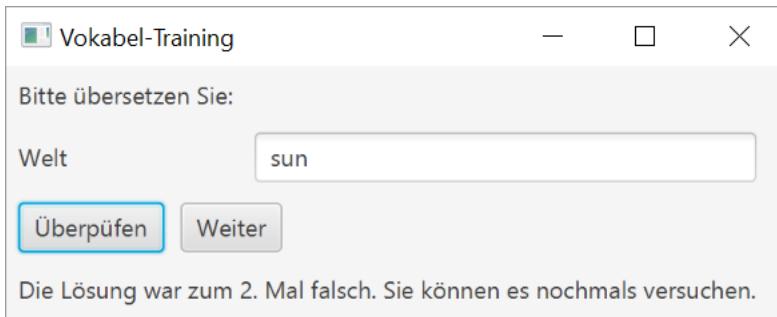


**Bild 4.12** Variante des Zusammenspiels beim MVP-Architekturmuster

Viele weitere Varianten, auch über Bindings, sind denkbar. Auch kann der Presenter in manchen Fällen erkennen, dass die eingegebenen Daten fehlerhaft oder unvollständig sind und daraufhin von sich aus die View anweisen, eine Fehlermeldung anzuzeigen, ohne dabei mit der Modellkomponente in Kontakt zu treten.

#### 4.4.2 Beispiel zu MVP

Als einfaches Beispiel einer MVP-Anwendung soll ein Vokabeltrainer entwickelt werden. Die Anwendung zeigt ein Fenster (s. Bild 4.13), in dem ein Wort angezeigt wird, das übersetzt werden soll. Man kann seine Übersetzung eintippen und prüfen lassen, ob die Eingabe korrekt war oder nicht. Wenn man möchte, kann man jederzeit über den Weiter-Button zur nächsten Vokabel gehen.



**Bild 4.13** Fenster eines Vokabeltrainers

Zuvor wurde erwähnt, dass alle drei Komponenten M, V und P jeweils aus mehreren Klassen bestehen können. Bei diesem ersten einfachen Beispiel gibt es pro Komponente allerdings nur eine einzige Klasse.

Wir beginnen unsere Implementierung mit der Modellkomponente. Ihre Hauptaufgabe besteht darin, einen neuen zu übersetzenden Begriff vorzuschlagen sowie zu überprüfen, ob die Eingabe dazu passt. Bitte erinnern Sie sich daran, dass die Modellkomponente keinerlei Bezug zu JavaFX haben sollte. Eine mögliche Implementierung findet sich in Listing 4.7.

#### **Listing 4.7**

```
import java.util.*;

public class Model
{
    private HashMap<String, String> vocabulary;
    private String[] keyWords;

    public Model()
    {
        vocabulary = new HashMap<>();
        fillVocab();
        keyWords = vocabulary.keySet().toArray(new String[0]);
    }

    public String choose()
    {
        int index = (int)(Math.random() * keyWords.length);
        return keyWords[index];
    }

    public boolean check(String lang1, String lang2)
    {
        return lang2.equals(vocabulary.get(lang1));
    }

    private void fillVocab() throws Exception
    {
        vocabulary.put("Hund", "dog");
        vocabulary.put("Katze", "cat");
    }
}
```

```
        vocabulary.put("Tisch", "table");
        vocabulary.put("Stuhl", "chair");
        //usw.
    }
}
```

Das Modell beinhaltet eine zweispaltige Tabelle in Form einer HashMap für die Begriffe und ihre Übersetzung. Um leichter zufällig einen Eintrag aus der HashMap auszuwählen, werden einfach die Strings der linken Spalte der HashMap (also die Keys) in ein String-Array kopiert. Dann kann man durch die zufällige Wahl eines Index den Begriff ermitteln, der als nächstes zur Übersetzung vorgeschlagen wird.

Im zweiten Schritt programmieren wir die View. Zur Erinnerung sei nochmals erwähnt, dass wir hier den Programmcode für den initialen Aufbau der Oberfläche, für die Veränderungen der Oberfläche sowie für die Weiterleitung von Ereignissen an den Presenter finden. Die View muss zu diesem Zweck eine Referenz auf den Presenter haben, der der View im Konstruktor als Parameter übergeben wird. Listing 4.8 enthält die View-Klasse.

#### **Listing 4.8**

```
import javafx.event.*;
import javafx.geometry.Insets;
import javafx.scene.layout.*;
import javafx.scene.control.*;

public class View
{
    private Presenter presenter;

    private GridPane pane;
    private Label question;
    private TextField solution;
    private Label status;

    public View(Presenter presenter)
    {
        this.presenter = presenter;
        initView();
    }

    private void initView()
    {
        pane = new GridPane();
        pane.setPadding(new Insets(6));
        pane.setHgap(10);
        pane.setVgap(10);
        pane.add(new Label("Bitte übersetzen Sie:"), 0, 0);
        question = new Label();
        pane.add(question, 0, 1);
        solution = new TextField();
        pane.add(solution, 1, 1);
        HBox buttons = new HBox(8);
        Button okay = new Button("Überprüfen");
        buttons.getChildren().add(okay);
        Button next = new Button("Weiter");
        buttons.getChildren().add(next);
    }
}
```

```
pane.add(buttons, 0, 2, 2, 1);
status = new Label();
pane.add(status, 0, 3, 2, 1);

EventHandler<ActionEvent> h = e ->
    presenter.check(question.getText(),
                    solution.getText());
solution.setOnAction(h);
okay.setOnAction(h);
next.setOnAction(e -> presenter.choose());
}

public Pane getUI()
{
    return pane;
}

public void showNewWord(String word)
{
    question.setText(word);
    solution.setText("");
}

public void eraseMessage()
{
    status.setText("");
}

public void showOkayMessage()
{
    status.setText("Die Lösung war richtig.");
}

public void showContinuationErrorMessage(int tries)
{
    if(tries <= 1)
    {
        status.setText("Die Lösung war falsch. " +
                      "Sie können es nochmals versuchen.");
    }
    else
    {
        status.setText("Die Lösung war zum " + tries + ". Mal " +
                      "falsch. Sie können es nochmals versuchen.");
    }
}

public void showFinalErrorMessage(int tries)
{
    status.setText("Die Lösung war zum " + tries + ". Mal falsch. " +
                  "Es geht weiter mit dem nächsten Wort.");
}

public void showNoInputMessage()
{
    status.setText("Es wurde keine Lösung angegeben.");
}
```

In der Methode initView wird die Oberfläche mit Hilfe eines GridPane aufgebaut. Beim Klicken auf den Weiter-Button wird die Methode choose des Presenters aufgerufen. Beim Klicken auf den Überprüfen-Button oder beim Drücken von Enter im Eingabefeld wird die Methode check des Presenters aufgerufen. Durch die Methode getUI kann man sich eine Referenz auf die GridPane geben lassen, um sie in eine Scene einer Stage zu setzen. Alle übrigen Methoden ändern Inhalte der Oberfläche.

Die Präsentationskomponente soll schließlich den Ablauf steuern und dabei sowohl die Dienste der Modell- als auch der Darstellungskomponente nutzen. Sie benötigt deshalb sowohl eine Referenz auf das Model als auch auf die View, die in der Methode setModelAndView gesetzt werden. Der Presenter enthält außerdem ein Attribut zum Zählen der Fehlversuche. Sein Code ist in Listing 4.9 zu sehen.

**Listing 4.9**

```
public class Presenter
{
    private static final int MAX_TRIES = 3;

    private Model model;
    private View view;
    private int tries;

    public void setModelAndView(Model model, View view)
    {
        this.model = model;
        this.view = view;
    }

    public void choose()
    {
        String word = model.choose();
        view.showNewWord(word);
        view.eraseMessage();
        tries = 0;
    }

    public void check(String lang1, String lang2)
    {
        lang2 = lang2.trim();
        if(lang2.length() == 0)
        {
            view.showNoInputMessage();
            return;
        }
        if(model.check(lang1, lang2))
        {
            view.showOkayMessage();
        }
        else
        {
            tries++;
            if(tries < MAX_TRIES)
            {
                view.showContinuationErrorMessage(tries);
            }
        }
    }
}
```

```
        else
        {
            choose();
            view.showFinalErrorMessage(MAX_TRIES);
        }
    }
}
```

Der Presenter realisiert zwei Abläufe: In der Methode choose wird eine neue Vokabel vom Model angefordert und auf der Oberfläche angezeigt. Eine bisher angezeigte Statusmeldung wird gelöscht. Die Anzahl der Fehlversuche wird auf 0 zurückgesetzt. In der Methode check wird zunächst überprüft, ob überhaupt etwas eingegeben wurde außer Leerzeichen, die durch die Methode trim entfernt werden. Gab es keine Eingabe, wird eine entsprechende Meldung in der View angezeigt. Andernfalls wird die Eingabe von der Modellkomponente überprüft. Falls die Eingabe richtig war, wird eine Erfolgsmeldung angezeigt. Andernfalls wird die Anzahl der Fehlversuche um eins erhöht. Ist die Anzahl der maximalen Versuche noch nicht erreicht, wird lediglich eine entsprechende Nachricht angezeigt. Andernfalls wird die nächste Vokabel angezeigt verbunden mit einer entsprechenden Nachricht.

Es fehlt schließlich noch ein Stückchen „Leim“, das die drei Komponenten „zusammenklebt“ und die View in einem Fenster anzeigt. Dieser fehlende Codeteil befindet sich in Listing 4.10.

### **Listing 4.10**

```
import javafx.application.Application;
import javafx.scene.Scene;
import javafx.stage.Stage;

public class Main extends Application
{
    public void start(Stage primaryStage)
    {
        Presenter p = new Presenter();
        View v = new View(p);
        Model m = new Model();
        p.setModelAndView(m, v);
        p.choose(); //automatisches Anstoßen eines ersten Ablaufs

        Scene scene = new Scene(v.getUI());
        primaryStage.setScene(scene);
        primaryStage.setTitle("Vokabel-Training");
        primaryStage.show();
    }

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

Damit haben wir unsere erste MVP-Anwendung vollständig implementiert. Vielleicht erscheint den Leseinnen und Lesern der Aufwand für die Nutzung des MVP-Musters bei dieser einfachen Anwendung zu hoch zu sein. Tatsächlich zahlt sich diese Strukturierung

erst bei größeren Anwendungen aus. Um aber dennoch ein klein wenig ein Gefühl für den Nutzen zu bekommen, betrachten wir zwei eminent wichtige Aspekte bei der Software-Entwicklung, nämlich das Testen und das Ändern.

Alle Komponenten können gut einzeln getestet werden. Bei der Modellkomponente ist dies besonders offensichtlich, da sie keine Abhängigkeiten zu den anderen Komponenten hat. Aber auch die Präsentationskomponente lässt sich mit entsprechenden Mock-Objekten für View und Model isoliert testen. Bei der Darstellungskomponente sieht ein Test so aus, dass man einerseits prüft, ob die Oberfläche zu Beginn richtig aussieht und ob sie sich bei Aufruf der entsprechenden Methoden in korrekter Weise verändert. Zum anderen sollte man prüfen, ob bei Benutzeraktionen die erwünschten Methoden des Presenters mit den zu erwartenden Parametern aufgerufen werden. Hierzu kann man ein Mock-Objekt anstelle des realen Presenters einsetzen.

Auch bei Änderungen ist aufgrund der klaren Aufgabenteilung klar, welche Komponenten betroffen sind. Oft wird es Änderungen geben, die mehrere Komponenten betreffen. Im Folgenden werden aber aus didaktischen Gründen nur solche Änderungen aufgeführt, die genau auf eine Komponente beschränkt sind:

- Mögliche Änderungen an der Modellkomponente: Nur die Modellkomponente muss geändert werden, wenn man die Begriffspaare ändern möchte, zum Beispiel statt Deutsch-Englisch nun Deutsch-Französisch. Auch könnte man sich vorstellen, dass die Begriffspaare in der Modellkomponente nicht „hart codiert“ sind, sondern aus einer Datei oder einer Datenbank zu Beginn oder bei Bedarf eingelesen werden. Eine weitere mögliche Änderung ist, dass die Methode check registriert, dass die angegebenen Begriffe nicht zueinander passen und bei der Auswahl eines neuen Begriffs dies berücksichtigt, indem mehrfach falsch übersetzte Ausdrücke mit größerer Wahrscheinlichkeit erneut angeboten werden. Auch könnte man beim Überprüfen etwas nachsichtiger sein und offensichtliche Tippfehler ignorieren.
- Mögliche Änderungen an der Darstellungskomponente: Die Oberfläche könnte selbstverständlich komplett anders aussehen, angefangen beim Layout. Aber auch die Beschriftungen auf der Oberfläche könnte man ändern; zum Beispiel könnte man diese in eine andere Sprache übersetzen. Gut vorstellbar ist auch, dass die Oberfläche zu Beginn nicht durch Programmcode aufgebaut wird, sondern durch Einlesen einer FXML-Datei. Eine weitere mögliche Änderung betrifft die Art, wie Fehlermeldungen angezeigt werden. Momentan gibt es hierfür ein Status-Label, in das Texte geschrieben werden. Alternativ könnte man die Meldungen in Dialogen mit Hilfe der Klasse Alert zur Ansicht bringen. Alles das sind Aspekte, die sich ausschließlich auf die Darstellung beziehen und weder mit der Geschäftslogik noch mit der Ablaufsteuerung zu tun haben.
- Mögliche Änderungen an der Präsentationskomponente: Da in diesem Fall die Abläufe sehr einfach sind, gibt es nicht viele Variationsmöglichkeiten. Mögliche Änderungen könnten sein, dass man die Zählung fehlerhafter Eingaben unterlässt und die Benutzerin unbegrenzt oft versuchen lässt, oder dass man beim Klicken auf den Weiter-Button nur dann die nächste Vokabel anzeigt, wenn mindestens ein Versuch unternommen wurde, den aktuell angezeigten Begriff zu übersetzen.

Vielleicht helfen diese Hinweise, den Nutzen des MVP-Musters noch etwas besser auch anhand einer kleinen Anwendung zu erfassen. In größeren Anwendungen kann es durchaus vorkommen, dass man mehrere Views hat, die gleichzeitig oder alternativ zu sehen

sind. Unter Umständen bietet es sich dann an, für jede View einen separaten Presenter zu implementieren, um die Einzelteile überschaubar zu halten.

## ■ 4.5 Threads und JavaFX

Nach dieser Einführung in JavaFX wenden wir uns nun wieder dem Thema Threads zu und ganz speziell der Frage, welche Beziehungen es zwischen Threads und JavaFX gibt.

### 4.5.1 Threads für JavaFX

Wir beginnen unsere Betrachtungen mit einer typischen Anwendung, wie wir sie beispielsweise in Listing 4.10 gesehen haben. In der Main-Methode wird launch aufgerufen, was irgendwie einen Aufruf der Methode start nach sich zieht. Da wir gehört haben, dass man Stage-Objekte auch explizit erzeugen kann, könnte man meinen, dass man auf launch verzichten und start nach Erzeugung eines Stage- und Main-Objekts auch in seinem Code aufrufen könnte. Also versuchen wir die Main-Methode von Listing 4.10 wie folgt zu ersetzen:

```
public static void main(String[] args)
{
    Stage s = new Stage();
    Main m = new Main();
    m.start(s);
}
```

Aber bereits die erste Anweisung, nämlich der Versuch ein Stage-Objekt zu erzeugen, löst eine Ausnahme des Typs IllegalStateException aus mit der Fehlermeldung „Not on FX application thread; currentThread = main“. Uns wird damit mitgeteilt, dass ein Stage-Objekt nicht im Main-Thread erzeugt werden darf. Um dies besser zu verstehen, schauen wir uns im Folgenden an, welche Threads durch JavaFX erzeugt werden und von welchem dieser Threads die Methode start ausgeführt wird. Davor werden natürlich die Änderungen in main wieder rückgängig gemacht, sodass main jetzt wieder als einzige Anweisung einen Aufruf von launch enthält. In die Methode start werden die folgenden zwei Zeilen Code eingefügt:

```
System.out.println("start: thread = " + Thread.currentThread().getName());
GroupTree.dumpAll();
```

Dabei greifen wir auf die statische Methode GroupTree.dumpAll aus Listing 2.33 zurück, die den Baum aller Threads eines Prozesses ausgibt. Wir sehen beim Starten unserer Anwendung folgende Ausgabe:

```
start: thread = JavaFX Application Thread
java.lang.ThreadGroup[name=system,maxpri=10]
    Thread[Reference Handler,10,system]
    Thread[Finalizer,8,system]
    Thread[Signal Dispatcher,9,system]
    Thread[Attach Listener,5,system]
java.lang.ThreadGroup[name=main,maxpri=10]
    Thread[main,5,main]
    Thread[QuantumRenderer-0,5,main]
    Thread[Thread-1,5,main]
    Thread[JavaFX Application Thread,5,main]
    Thread[JavaFX-Launcher,5,main]
    Thread[Thread-3,5,main]
```

Wenn man diese Ausgabe von `GroupTree.dumpAll` mit der von Abschnitt 2.8 vergleicht, dann sieht man, dass die letzten fünf Threads dazugekommen sind. Einer dieser Threads, und zwar der mit dem Namen „JavaFX Application Thread“, führt die Methode `start` aus, wie die erste Ausgabezeile beweist. Wenn wir den Namen des aktuellen Threads zum Beispiel auch in den Methoden, die als `ActionListener` oder `MouseListener` dienen (in Listing 4.6 fungieren als `ActionListener` die Methode `clear` und als `MouseListener` die Methoden `mousePressed` und `mouseDragged`, im MVP-Beispiel könnte man dafür die Methoden des Presenters auswählen, da diese von den Listenern der View aufgerufen werden), ausgeben, dann sieht man, dass auch in diesem Fall der „JavaFX Application Thread“ am Werk ist. Jetzt verstehen wir auch, was mit der Fehlermeldung „Not on FX application thread; current-Thread = main“ zu der oben geworfenen Ausnahme des Typs `IllegalStateException` gemeint war, nämlich dass das Stage-Objekt vom „JavaFX Application Thread“ erzeugt werden sollte und nicht vom Main-Thread. Welche Bedeutung dieser Thread für die Praxis hat, wollen wir uns am folgenden Beispiel anschauen.

## 4.5.2 Länger dauernde Ereignisbehandlungen

Wir kehren zu einem unseren ersten JavaFX-Beispiele zurück, nämlich zu der Zähler-Anwendung von Listing 4.3. Angenommen, wir möchten diese Anwendung nun so ändern, dass bei jedem Drücken des Erhöhen-Buttons der Zähler um 10 erhöht wird. Die Erhöhung soll allerdings nicht auf einen Schlag erfolgen, sondern schrittweise. Das heißt, der Zähler soll 10 Mal um eins erhöht werden und jeder Zwischenstand soll auf der Oberfläche angezeigt werden. Damit dies nicht zu schnell abläuft und die Zwischenstände von Menschen wahrgenommen werden können, muss nach jeder Erhöhung und Anzeige des Zählers eine gewisse Zeit gewartet werden. Als Wartezeit wählen wir eine Sekunde. Aus Gründen, die später noch deutlich werden, geben wir die Nachricht, bevor sie in das Label geschrieben wird, auch auf der Console aus. Die Methode `increment` aus Listing 4.3 ändert sich damit wie folgt:

```

private void increment()
{
    for(int i = 1; i <= 10; i++)
    {
        counter++;
        String message = "" + counter;
        System.out.println(message);
        label.setText(message);
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
        }
    }
}

```

Wenn wir unsere so geänderte Anwendung ausprobieren, erleben wir eine Überraschung. Wenn wir auf den Erhöhen-Button drücken, dann sehen wir auf der Console, dass der Zähler im Sekudentakt hochgezählt wird. Allerdings ändert sich der Fensterinhalt nicht. Erst, nachdem alle Erhöhungen durchgeführt wurden, springt die Zähleranzeige auf einen Schlag um 10 nach oben. Genau dies wollten wir nicht, sondern wir wollten die allmähliche Erhöhung des Zählers im Fenster verfolgen. Es gibt aber noch ein weiteres Problem: Während der Erhöhung des Zählers reagiert unsere Anwendung zunächst auf keine weiteren Klicks, weder auf einen Klick unseres Zurücksetzen-Buttons noch auf die von JavaFX im Fensterrahmen bereitgestellten Buttons zum Minimieren, Maximieren und Schließen des Fensters. Das Wort „zunächst“ im vorigen Satz ist wichtig, denn nach der Rückkehr aus der Methode increment erfolgen die verspäteten Reaktionen auf die inzwischen geklickten Buttons. Dies schließt das einfache oder mehrfache Klicken des Erhöhen-Buttons mit ein. Wenn man also beispielsweise sechs Mal sehr schnell hintereinander den Erhöhen-Button klickt, wird der Zähler um 60 erhöht. Während einer ganzen Minute ist die Anwendung dann scheinbar nicht mehr benutzbar.

Der Grund für das beschriebene Verhalten ist beim „JavaFX Application Thread“ zu finden. Dieser Thread ist für die Ereignisbehandlung einer JavaFX-Anwendung zuständig und führt im Prinzip den folgenden Code aus:

```

while(true)
{
    hole nächstes Ereignis aus Ereigniswarteschlange;
    erzeuge Event e;
    für alle für dieses Ereignis registrierten Listener
    {
        führe Listener-Methode aus mit Parameter e;
    }
}

```

Die komplette Ereignisbehandlung wird von diesem einen Thread in sequenzieller Weise durchgeführt. Das heißt: Ereignisse wie Mausklicks und Mausbewegungen werden in eine Ereigniswarteschlange eingefügt. Der „JavaFX Application Thread“ wartet an dieser Warteschlange, bis das nächste Ereignis entnommen werden kann. Danach werden alle relevan-

ten Listener der Reihe nach aufgerufen. Wenn also eine Listener-Methode länger läuft, dann finden natürlich in dieser Zeit keine weiteren Ereignisbehandlungen mehr statt. Dies erklärt das Nichtreagieren auf weitere Aktionen während dieser Zeit. Alle Aktionen werden aber in die oben erwähnte Ereigniswarteschlange eingereiht. Deshalb erfolgt später doch noch eine Reaktion auf die Benutzeraktionen. Während Ereignisse behandelt werden, findet auch kein Neuzeichnen des Fensterinhals statt. Deshalb sehen wir in unserem Fenster das schrittweise Erhöhen des Zählers, das durch `setText` angezeigt werden sollte, nicht. Erst nach der Rückkehr aus der Methode `increment` wird der Fensterinhalt neu gezeichnet, wobei der Text des Labels dann schon auf den finalen Zustand gesetzt wurde. Das Gesagte gilt übrigens auch dann, wenn in einer JavaFX-Anwendung mehrere Fenster vorhanden sind. Auch dann gibt es nur einen einzigen Thread zur Ereignisbehandlung. Während also eine länger dauernde Ereignisbehandlung läuft, kann man mit keinem der Fenster dieser Anwendung arbeiten. Auch werden die Inhalte aller Fenster dieser Anwendung in dieser Zeit nicht aktualisiert.

Wer bis jetzt glaubt, dass das bisherige Beispiel der schrittweisen Zählererhöhung sehr künstlich und für die Praxis nicht relevant ist, täuscht sich. Man stelle sich eine JavaFX-Anwendung vor, in der durch eine Benutzeraktion eine länger dauernde Datenübertragung angestoßen wird, wie sie in verteilten Anwendungen, denen wir uns in den folgenden Kapiteln zuwenden, üblich ist. Während dieser Übertragung soll auf der Oberfläche immer wieder angezeigt werden, wie viele Bytes schon übertragen wurden. Mit der bisher gezeigten Art der Programmierung ist dies nicht möglich, wie wir gesehen haben. Wenn dann außerdem die Anwendung scheinbar auf nichts mehr reagiert, dann kann es durchaus passieren, dass ein Benutzer (hier ist die männliche Form ganz bewusst gewählt) wahllos mit der Maustaste auf der Anwendung herum hämmert, was dann dazu führen kann, dass viele unbeabsichtigte Aktionen angestoßen werden, die nach Beendigung der Datenübertragung ausgeführt werden. Derartige Anwendungen sind also wenig gebrauchstauglich. Wir sollten deshalb die folgende Regel unbedingt einhalten:



Regel 1: Die Durchführung einer Ereignisbehandlung sollte in möglichst kurzer Zeit abgeschlossen sein. In keinem Fall sollten also während einer Ereignisbehandlung Methoden wie z.B. `sleep` und `wait` aufgerufen werden.

Eine ganz naheliegende Lösung, um diese Regel einzuhalten, ist die Benutzung von Threads. Wir ändern also unsere Methode `increment` wie folgt:

```
private void increment()
{
    new Thread(()->incrementAsync()).start();
}

private void incrementAsync()
{
    //bisheriger Inhalt von increment mit sleep: ...
}
```

Beim Ausprobieren erleben wir die nächste Enttäuschung. Auch jetzt sehen wir kein allmähliches Anwachsen des Zählers. Schlimmer noch, die Anzeige ändert sich überhaupt

nicht, auch nach 10 Sekunden nicht. Stattdessen sehen wir in der Console im Sekundentakt neben der bisherigen Ausgabe, dass Ausnahmen geworfen werden. Es handelt sich dabei um Ausnahmen mit einer uns bereits bekannten Fehlermeldung: „Not on FX application thread; currentThread = Thread-4“. Die Ausnahme passiert in der Methode incrementAsync in der Zeile, in der der Inhalt des Labels mit setText geändert werden sollte. Die Fehlermeldung besagt, dass Thread-4 (das ist der in increment gestartete Thread) versucht, den Label-Text zu ändern, dass dies aber nur dem „JavaFX Application Thread“ erlaubt ist (genauso wie die Erzeugung eines Stage-Objekts, was wir zu Beginn dieses Abschnitts im Main-Thread versucht hatten). Die Ausnahme führt übrigens trotz nicht vorhandenem Try-Catch-Block um setText nicht zum Abbruch von incrementAsync. Deshalb sehen wir die Ausnahmemeldung 10 Mal im Sekundentakt. Der Zähler wird zwar um 10 erhöht, aber im Fenster wird dies nicht angezeigt.

Wir scheinen jetzt in einem Dilemma zu stecken: Einerseits brauchen wir Threads wegen der länger dauernden Ereignisbehandlung, andererseits sind Threads aber kontraproduktiv wegen des Zugriffs auf die Oberfläche. Es gibt jedoch einen recht einfachen Ausweg aus diesem Dilemma: Die statische Methode *runLater* der Klasse *Platform* ermöglicht, einen Auftrag in die Ereigniswarteschlange einzureihen, aus der der „JavaFX Application Thread“ seine Ereignisse entnimmt:

```
public class Platform
{
    public static void runLater(Runnable doRun) {...}
}
```

Der Auftrag muss als Runnable-Objekt angegeben werden (das heißt, als Objekt einer Klasse, die Runnable implementiert). Selbstverständlich können wir den Parameter von runLater deshalb als Lambda-Ausdruck angeben. Es ist also ergänzend zu der obigen Regel folgende weitere Regel zu beachten:



**Regel 2:** Der Zugriff auf Elemente der grafischen Oberfläche darf ausschließlich durch den „JavaFX Application Thread“ erfolgen. Wenn ein anderer Thread auf die Oberfläche zugreifen möchte, so beauftragt er dazu den „JavaFX Application Thread“ durch Aufruf der Methode Platform.runLater. Der Auftrag muss in Form eines Runnable-Objekts übergeben werden.

Wir ersetzen also die Anweisung label.setText in der Methode increment, die inzwischen incrementAsync heißt, durch folgende Zeile:

```
Platform.runLater(()->label.setText(message));
```

Jetzt endlich funktioniert unsere Anwendung bei oberflächlichem Ausprobieren so, wie wir uns das von Anfang an gewünscht haben. Unsere Anwendung ist jedoch nur scheinbar korrekt. Wir benutzen inzwischen Threads und Parallelität. Und dies ist, wie Sie gelernt haben, immer wieder tückisch. Man kann nämlich den Erhöhen-Button mehrfach relativ schnell hintereinander klicken, was bewirkt, dass die Methode incrementAsync parallel von mehreren Threads ausgeführt wird. Das heißt, dass diese Threads parallel schreibend und lesend auf den Zähler zugreifen. Während diese Threads laufen, kann außerdem auch der

Zurücksetzen-Button gedrückt werden. Dies bewirkt einen Aufruf der Methode reset durch den „JavaFX Application Thread“. Auch in dieser Methode wird der Zähler verändert und gelesen. Die beschriebene Problematik existiert schon in unserer Programmvariante vor der Nutzung von Platform.runLater. Da aber diese Variante sowieso falsch war, wurde dieser Punkt oben nicht thematisiert.

Um aber jetzt zu einer korrekten Lösung zu gelangen, müssen wir die Zugriffe auf den Zähler synchronisieren oder alternativ dafür sorgen, dass nur noch ein einziger Thread auf den Zähler zugreift. Da die zweite Variante weniger Codeänderungen notwendig macht, gehen wir im Folgenden diesen Weg. Dies soll aber nicht bedeuten, dass dies in allen Fällen die bessere Variante ist. Wir lagern in der Methode incrementAsync nicht nur die Aktualisierung des Label-Texts als Auftrag an den „JavaFX Application Thread“ aus, sondern auch die Erhöhung und das Lesen des Zählers. Damit werden alle Zugriffe auf den Zähler nur noch vom „JavaFX Application Thread“ durchgeführt, so dass auf eine Synchronisation verzichtet werden kann. Bevor wir den geänderten Programmcode zeigen, beschreiben wir den Grund für eine weitere Änderung.

Wenn man die Anwendung während des Hochzählens durch Drücken auf das Kreuz oben rechts im Fensterrahmen schließt, dann verschwindet zwar das Fenster sofort vom Bildschirm, aber anhand der Ausgaben auf der Console sieht man, dass der Thread noch weiterläuft. Erst nach dem Ende dieses Threads stirbt auch der Prozess. Dies ist nicht verwunderlich, da der gestartete Thread kein Hintergrund-Thread ist. Wenn man möchte, dass der Prozess direkt nach dem Schließen des Fensters endet, dann sollte man den gestarteten Thread als Hintergrund-Thread (Daemon Thread) deklarieren. Die Methoden increment und incrementAsync haben also nach all diesen Änderungen folgendes Aussehen:

```
private void increment()
{
    Thread t = new Thread(()->incrementAsync());
    t.setDaemon(true);
    t.start();
}

private void incrementAsync()
{
    for(int i = 1; i <= 10; i++)
    {
        Platform.runLater(()->reallyIncrement());
        try
        {
            Thread.sleep(1000);
        }
        catch(InterruptedException e)
        {
        }
    }
}

private void reallyIncrement()
{
    counter++;
    String message = "" + counter;
    System.out.println(message);
```

```
    label.setText(message);
}
}
```

Wir wenden uns unserer Anwendung jetzt wieder aus der Benutzerperspektive zu. Wie schon erwähnt wurde, kann während der schrittweisen Erhöhung der Zähler zurückgesetzt werden. Wenn der Zurücksetzen-Button beispielsweise betätigt wird, nachdem der Zähler auf 6 angewachsen ist, dann ist der Endwert nur noch 4. Dies sollte nicht überraschen und auch bezüglich der Benutzung kein Problem darstellen. Wie auch gerade diskutiert wurde, ist es möglich, den Erhöhen-Button während des Hochzählens noch einmal oder sogar mehrmals zu drücken. Die Anwendung ist so programmiert, dass bei jedem Drücken ein neuer Thread gestartet wird. Diese parallel laufenden Threads bewirken, dass der Zähler jetzt öfter und schneller als im Sekudentakt erhöht wird. In der hier vorliegenden Anwendung kann man sicher damit leben. Ob dies in einer anderen Anwendung auch so sein soll, hängt von der Anwendung ab. Wenn man diese parallele Auftragsverarbeitung nicht wünscht, so kann man auf unterschiedliche Arten dagegen angehen:

- Eine Möglichkeit besteht darin, neue Aufträge während einer laufenden Auftragsverarbeitung einfach zu ignorieren.
- Alternativ kann auch verhindert werden, dass während der Auftragsverarbeitung neue Aufträge generiert werden. Im Falle der hier vorliegenden Anwendung könnte man dies dadurch bewerkstelligen, dass man die beiden Buttons zu Beginn der Methode increment durch Aufruf einer entsprechenden Methode der View auf Disabled setzt (Methode *setDisable* mit Parameter true, die Buttons werden ausgegraut, das Drücken der Buttons löst keine Ereignisse mehr aus). Am Ende der Auftragsbearbeitung (also als letzte Anweisung der Methode incrementAsync) sollte man die Buttons dann wieder auf Enabled setzen (Aufruf einer View-Methode über Platform.runLater, in der Aufrufe der Methode *setDisable* mit Parameter false stattfinden).
- Eine weitere Möglichkeit, um gegen die parallele Auftragsbearbeitung vorzugehen, besteht darin, dass man zwar weiterhin erlaubt, dass während der Bearbeitung neue Aufträge generiert werden, dass man diese aber sequenziell abarbeitet. Dazu könnte man beispielsweise beim Programmstart einen Thread-Pool mit einem einzigen Thread erzeugen und bei jedem Klicken des Erhöhen-Buttons, also in der Methode increment, einen Auftrag diesem Thread-Pool übergeben. Die Aufträge werden somit rein sequenziell abgearbeitet. Ein Zugriff auf die Oberfläche ist natürlich dabei nicht erlaubt. Falls der Thread des Thread-Pools und der „JavaFX Application Thread“ u. a. schreibend auf gemeinsamen Daten arbeiten, ist eine Synchronisation nötig. Diesen Aspekt sollte man grundsätzlich nie aus den Augen verlieren.

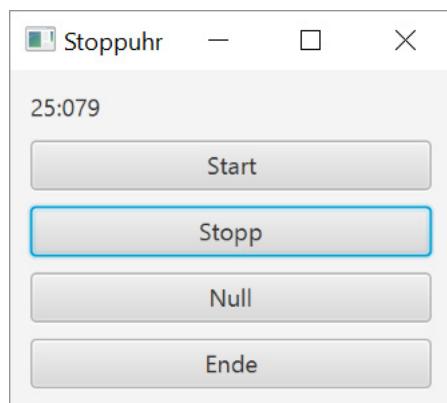
Welche dieser Varianten gewählt werden sollte, hängt von der Anwendung ab.

### 4.5.3 Beispiel Stoppuhr

Das soeben behandelte Beispiel, bei dem ein Zähler nach dem Drücken einer Taste allmählich anwächst, wollen wir nun in eine etwas sinnvollere Anwendung transformieren, und zwar in eine einfache Anwendung für eine Stoppuhr (s. Bild 4.14). Es soll je einen Button

zum Starten, zum Anhalten und zum Nullstellen der Uhr sowie zum Beenden des Programms geben.

Bevor wir näher auf die Implementierung schauen, sollen zwei wesentliche Entwurfsentscheidungen vorab erläutert werden: Bei der Implementierung soll es nicht so sein, dass man einen Zähler für die vergangene Zeit in Millisekunden hat und man diesen Zähler zum Beispiel jeweils um  $x$  erhöht ( $x$  zum Beispiel 10) und danach  $x$  Millisekunden schläft. Dies kann leicht dazu führen, dass die real vergangene Zeit größer ist als die angezeigte, da man dabei zum einen die Rechenzeit des eigenen Programms, aber auch die der anderen Programme vernachlässigt. Zum anderen könnte das Schlafen länger dauern als angegeben (wir erinnern uns, dass nach einem Schlafen von  $x$  Millisekunden mehr Zeit vergangen sein kann als  $x$  Millisekunden). Deshalb merken wir uns stattdessen die Zeit, wann die Stoppuhr gestartet wurde, und zeigen alle  $x$  Millisekunden an, wie viel Zeit seit dem Starten vergangen ist.



**Bild 4.14**

Benutzeroberfläche für eine Stoppuhr

Ferner soll statt einer Weiterentwicklung unseres vorhergehenden Zählerbeispiels der Programmcode für die Stoppuhr nach dem MVP-Architekturmuster gestaltet werden. Dies ist die zweite wesentliche Entwurfsentscheidung.

Damit können wir uns nun dem Programmcode zuwenden, der in Listing 4.11 dargestellt ist. Das Modell besteht aus der Klasse Clock. Bei reset wird die aktuelle Zeit in einem Attribut der Klasse gespeichert. Wenn man mit getTime die Zeit abfragt, dann wird die Zeit zurückgeliefert, die seit dem letzten Reset vergangen ist. Die View wird durch die Klasse ClockView realisiert. In ihr befindet sich der initiale Aufbau des Fensterinhalts sowie eine Methode zum Anzeigen der vergangenen Zeit (Sekunden und Millisekunden durch Doppelpunkt getrennt, Millisekunden immer mit drei Ziffern, Anzeige als String in einem Label). Der Presenter besteht aus den Klassen ClockPresenter und Ticker. Für jeden der vier Buttons in ClockView gibt es eine Methode in ClockPresenter, die beim Klicken des entsprechenden Buttons aufgerufen wird. Ticker ist eine Thread-Klasse, die für das wiederholte Anzeigen der vergangenen Zeit zuständig ist.

#### Listing 4.11

```
import javafx.application.*;
import javafx.geometry.Insets;
```

```
import javafx.scene.Scene;
import javafx.scene.control.*;
import javafx.scene.layout.*;
import javafx.stage.Stage;

/* Model */
class Clock
{
    private long startTime;

    public long getTime()
    {
        return System.currentTimeMillis() - startTime;
    }

    public void reset()
    {
        startTime = System.currentTimeMillis();
    }
}

/* View */
class ClockView
{
    private ClockPresenter presenter;

    private VBox root;
    private Label timeLabel;

    public ClockView(ClockPresenter presenter)
    {
        this.presenter = presenter;
        initView();
    }

    private void initView()
    {
        root = new VBox(8);
        root.setPadding(new Insets(10));
        timeLabel = new Label();
        root.getChildren().add(timeLabel);
        Button b1 = new Button("Start");
        b1.setOnAction(e->presenter.start());
        b1.setMaxWidth(Double.MAX_VALUE);
        root.getChildren().add(b1);
        Button b2 = new Button("Stopp");
        b2.setOnAction(e->presenter.stop());
        b2.setMaxWidth(Double.MAX_VALUE);
        root.getChildren().add(b2);
        Button b3 = new Button("Null");
        b3.setOnAction(e->presenter.reset());
        b3.setMaxWidth(Double.MAX_VALUE);
        root.getChildren().add(b3);
        Button b4 = new Button("Ende");
        b4.setMaxWidth(Double.MAX_VALUE);
        b4.setOnAction(e->presenter.exit());
        root.getChildren().add(b4);
    }
}
```

```
public Pane getUI()
{
    return root;
}

public void showTime(long elapsedTime)
{
    long seconds = elapsedTime / 1000;
    long milliSecs = elapsedTime % 1000;
    String prefix;
    if(milliSecs < 10)
    {
        prefix = "00";
    }
    else if(milliSecs < 100)
    {
        prefix = "0";
    }
    else
    {
        prefix = "";
    }
    timeLabel.setText(seconds + ":" + prefix + milliSecs);
}

/* Presenter */
class TickerThread extends Thread
{
    private final static long UPDATE_INTERVAL = 10; // Milliseconds
    private Clock clock;
    private ClockView view;

    public TickerThread(Clock clock, ClockView view)
    {
        this.clock = clock;
        this.view = view;
        setDaemon(true);
        start();
    }

    public void run()
    {
        try
        {
            while(!isInterrupted())
            {
                Platform.runLater(() -> view.showTime(clock.getTime()));
                Thread.sleep(UPDATE_INTERVAL);
            }
        }
        catch(InterruptedException e)
        {
        }
    }
}
```

```
class ClockPresenter
{
    protected Clock clock;
    protected ClockView view;
    private TickerThread thread;

    public void setModelAndView(Clock clock, ClockView view)
    {
        this.clock = clock;
        this.view = view;
    }

    public void start()
    {
        if(thread == null)
        {
            clock.reset();
            thread = new TickerThread(clock, view);
        }
    }

    public void stop()
    {
        if(thread != null)
        {
            thread.interrupt();
            thread = null;
        }
    }

    public void reset()
    {
        clock.reset();
        view.showTime(clock.getTime());
    }

    public void exit()
    {
        Platform.exit();
    }
}

/* Main */
public class ClockManager extends Application
{
    public void start(Stage primaryStage)
    {
        ClockPresenter p = new ClockPresenter();
        ClockView view = new ClockView(p);
        Clock clock = new Clock();
        p.setModelAndView(clock, view);
        p.reset();

        Scene scene = new Scene(view.getUI());
        primaryStage.setScene(scene);
        primaryStage.setTitle("Stoppuhr");
        primaryStage.show();
    }
}
```

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

Im vorhergehenden Beispiel wurden drei Möglichkeiten diskutiert, wie man bei Bedarf verhindern kann, dass eine weitere Hintergrundaktivität angestoßen wird, falls schon eine läuft. In diesem Beispiel haben wir eine Realisierung der ersten angegebenen Möglichkeit (Ignorieren neuer Aufträge) gesehen: Der Presenter merkt sich, ob schon ein Thread läuft. Nur falls keiner läuft, wird einer gestartet. Ansonsten geschieht nichts. Alternativ könnte man auch die zweite Möglichkeit umsetzen (Verhindern des Erzeugens neuer Aufträge). Bei dieser Anwendung könnte man dies außer durch das Disabled-Setzen des Start-Buttons auch dadurch realisieren, dass man statt zwei Buttons für das Starten und Anhalten der Uhr nur einen einzigen Button hat, der beide Funktionen erfüllt: Wenn die Stoppuhr nicht läuft, dient der Button zum Starten, wenn sie läuft, zum Anhalten der Uhr. Selbstverständlich muss die Beschriftung des Buttons dann von der View entsprechend gewechselt werden.

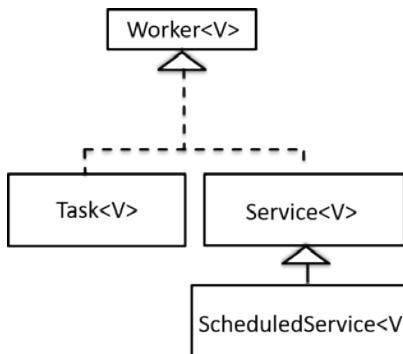
Die Oberfläche enthält einen Ende-Button, wie in Bild 4.14 zu sehen ist. Beim Klicken auf diesen Button wird die Anwendung beendet. Eigentlich wäre dieser Button nicht nötig. Er ist hier vorhanden um zu demonstrieren, dass auch die Beendigung des Programms eventuell noch einen bestimmten Ablauf notwendig machen könnte, zum Beispiel das persistente Speichern von Daten. Hier in diesem Beispiel ist dies nicht nötig. Deshalb wird in der Methode exit des Presenters lediglich Platform.exit aufgerufen, was die JavaFX-Anwendung beendet und damit in der Folge auch den Prozess, da danach keine Vordergrund-Threads mehr laufen (in diesem Fall hätten wir auch System.exit verwenden können, was direkt und ohne Bedingungen den eigenen Prozess zwangsbeendet).

Man könnte sich jetzt noch die Frage stellen, ob im obigen Programm eine Synchronisation notwendig ist, die bisher vergessen wurde. Man kann nämlich bei laufender Uhr jederzeit auf den Null-Button klicken und die Uhr dadurch auf 0 zurückstellen. Somit könnte man meinen, dass während des wiederholten Lesens der Uhrzeit durch den Ticker-Thread gleichzeitig der Stand der Uhr verändert werden könnte. Diese Gefahr besteht aber nur scheinbar. In der Methode run der Klasse Ticker ist der Aufruf von clock.getTime mit in Platform.runLater enthalten. Das heißt, dass clock.getTime vom „JavaFX Application Thread“ ausgeführt wird. Der Thread, der auf das Klicken des Null-Buttons reagiert, ist bekanntlich ebenfalls der „JavaFX Application Thread“. Und sogar die Erzeugung des Clock-Objekts und der initiale Aufruf von reset des Presenters, der reset für Clock zur Folge hat, wird in der Methode start und damit auch von diesem Thread ausgeführt. Das bedeutet zusammengekommen, dass in dieser Anwendung nur der „JavaFX Application Thread“ auf das Clock-Objekt zugreift und es keine parallelen Zugriffe gibt. Die Situation ist also bezüglich des Zugriffs auf das Modell hier so wie in unserer zuvor besprochenen Zähleranwendung: Nur ein einziger Thread greift auf die Daten des Modells zu, so dass keine Synchronisation notwendig ist.

Es sei schließlich noch darauf hingewiesen, dass durch Änderung der Konstanten UPDATE\_INTERVAL im Presenter die Uhranzeige öfter oder seltener aktualisiert wird, dass dies aber keinen Einfluss auf die Geschwindigkeit der Uhr hat, da sich diese an der real vergangenen Zeit orientiert. Das heißt, bei jeder Aktualisierung wird die richtige Zeit angezeigt.

#### 4.5.4 Tasks und Services in JavaFX

Wie man sich sicher leicht vorstellen kann, kommt das Anstoßen länger dauernder Aktivitäten über eine grafische Benutzeroberfläche in vielen Anwendungen vor. JavaFX bietet deshalb hierfür eine Unterstützung an, die im Wesentlichen aus der generischen Schnittstelle *Worker* sowie den generischen Klassen *Task*, *Service* und *ScheduledService* besteht (Bild 4.15). Alle drei Klassen sind abstrakt. Das heißt, man kann sie nur nutzen, indem man eigene Klassen daraus ableitet.



**Bild 4.15**

Schnittstelle und Klassen zur Unterstützung länger dauernder Ereignisbehandlungen

Mit Hilfe einer Task kann eine einmalige Ausführung einer länger dauernden Aktivität realisiert werden. Wenn dies wiederholt erfolgen soll, muss man jedes Mal ein neues Task-Objekt (also ein Objekt einer aus Task abgeleiteten Klasse) erzeugen. Man kann die wiederholte Aktivierung aber auch mit Hilfe eines Services realisieren, der allerdings auch immer wieder Task-Objekte erzeugt. ScheduledService schließlich kann genutzt werden, wenn die Ausführung einer Aktivität periodisch (zum Beispiel jede Minute) angestoßen werden soll.

Das JavaFX-Konzept der Properties spielt auch im Zusammenhang mit Tasks und Services eine wichtige Rolle. Mit Hilfe der Properties der Tasks und Services kann die Kommunikation zwischen den Threads, welche die länger dauernde Aktivität durchführen, und dem „JavaFX Application Thread“ erfolgen. Bei entsprechender Nutzung kann man ganz ohne Platform.runLater auskommen (eine Benutzung von Platform.runLater ist aber nicht verboten und durchaus möglich). Dazu ist es ganz wichtig zu wissen, dass die Methoden der Listener, die an diesen Properties angemeldet sind, immer vom „JavaFX Application Thread“ ausgeführt werden. Das heißt, bei Änderung eines Werts dieser Properties darf in den Listener-Methoden auf die JavaFX-Oberfläche zugegriffen werden. Auch eine Kopplung von Properties der JavaFX-Elemente an diese Properties ist deshalb möglich. Man könnte nun meinen, dass die Aktivitäten-Threads nun einfach die Werte der Properties ändern können. Dies stimmt aber nicht. Im Gegenteil: Eine direkte Änderung der Werte der Properties ist diesen Threads nicht erlaubt. Für die meisten dieser Properties wird dies schon dadurch verhindert, dass sie ReadOnly-Properties sind. Dann stellt sich natürlich die Frage, wie sich an den Properties überhaupt etwas ändern lässt. Hierzu gibt es zwei Möglichkeiten: Entweder gibt es spezifische Methoden zum expliziten Ändern der Werte, deren Aufruf den Aktivitäten-Threads ganz ausdrücklich erlaubt ist, oder die Wertänderung erfolgt implizit durch das JavaFX-System. Welcher dieser beiden Wege möglich ist, hängt von der jeweiligen Property ab. Im Folgenden schauen uns wir die wichtigsten Properties der Klasse Task einmal etwas näher an:

- **message** (ReadOnlyStringProperty): Diese Property dient dazu, Meldungen von den Aktivitäten-Threads an den „JavaFX Application Thread“ zu übertragen. Zur Änderung des Message-Strings der Property stellt die Klasse Task die Methode *updateMessage* bereit, die von den Aktivitäten-Threads aufgerufen werden darf.
- **title** (ReadOnlyStringProperty): Ähnlich wie message. Analog zu *updateMessage* gibt es die Methode *updateTitle* zum Ändern dieses Property-Werts.
- **value** (ReadOnlyObjectProperty<V>): Der Typ V dieser Property ist der generische Typ der Klasse Task. Zum Ändern der Referenz auf ein Objekt des Typs V gibt es analog zu *updateMessage* und *updateTitle* die Methode *updateValue* mit einem Parameter des Typs V. Da V alles sein kann, hat man damit die größtmögliche Flexibilität zur Übertragung von Informationen von den Aktivitäten-Threads an den „JavaFX Application Thread“.
- **workDone** und **totalWork** (jeweils ReadOnlyDoubleProperty): Wie die Namen andeuten, geht es dabei um die bereits geleistete Arbeit (workDone) in Relation zur gesamten Arbeit (totalWork). Beide Angaben werden durch Werte des Typs double repräsentiert. Mit der Methode *updateProgress* werden beide Werte auf einen Schlag gesetzt.
- **progress** (ReadOnlyDoubleProperty): Dies ist nun eine Property, zu der es keine direkte Methode für das Ändern gibt. Der Wert dieser Property ist immer workDone/totalWork. Sie ändert sich beim Aufruf von *updateProgress* automatisch mit.
- **state** (ReadOnlyObjectProperty<Worker.State>): Eine Task durchläuft ähnlich wie ein Thread unterschiedliche Zustände. Zu Beginn ist sie im Zustand READY. Nachdem eine Task gestartet wurde, ist sie im Zustand RUNNING. Nach Beendigung kann sie in unterschiedliche Zustände wechseln, je nachdem, ob sie regulär (Zustand SUCCEEDED), durch eine Ausnahme (Zustand FAILED) oder durch Abbruch (Zustand CANCELLED) zu Ende gelaufen ist. Man kann so ihren Zustand beobachten. Man kann diesen Zustand aber nicht explizit über eine bestimmte Methode setzen, was keinen Sinn machen würde.

Um die länger laufende Aktivität zu programmieren, leitet man seine eigene Klasse aus Task ab und überschreibt die Methode *call*. Darin kann man die aus Task geerbten Update-Methoden wie *updateValue* und *updateProgress* aufrufen, um Zwischenstände an den „JavaFX Application Thread“ zu übermitteln. Selbstverständlich sollte man in *call* nicht auf die Oberfläche zugreifen, zumindest nicht direkt, sondern nur mittels Platform.runLater. Der Rückgabetyp von *call* ist der Typparameter der Klasse Task (also für „extends Task<Long>“ muss *call* den Rückgabetyp Long haben); dieser Typ ist somit identisch mit dem Typ der Value-Property. Die Value-Property übernimmt den von *call* zurückgelieferten Wert. Die Klasse Task beinhaltet keinen Code zur Ausführung der Methode *call* in einem Thread. Das muss vollständig separat von der Anwendungsentwicklerin dazu programmiert werden. Man hat aber dadurch freie Hand und kann sich entscheiden, ob man beispielsweise für jedes neue Task-Objekt auch einen neuen Thread erzeugen und starten möchte, oder ob man lieber die Task-Objekte einem Thread-Pool zur Ausführung übergeben möchte. Da die Klasse Task die Runnable-Schnittstelle implementiert und somit eine Run-Methode hat, in der *call* aufgerufen wird, kann ein Task-Objekt sowohl einem Thread-Konstruktor als auch einem Thread-Pool durch Aufruf der Methode *execute* übergeben werden.

Die Klasse Task besitzt die Methode *cancel* zum Abbrechen einer laufenden Hintergrundaktivität. Ein Aufruf von *cancel* hat aber nur dann einen Effekt, wenn die Methode *call* so programmiert ist, dass sie dabei „mitspielt“. Das heißt, in *call* sollte immer wieder überprüft

werden, ob cancel aufgerufen wurde (dafür gibt es die Methode *isCancelled*). Ist dies der Fall, sollte man aus call zurückkehren. Im Prinzip gilt hier genau dasselbe wie beim zwangsweisen Beenden eines Threads über Interrupt (vgl. Unterabschnitt 2.4.2). Mit der Anwendung von cancel auf das Task-Objekt wird zusätzlich auch ein Interrupt-Signal an den ausführenden Thread geschickt, sodass blockierende Operationen wie sleep und wait abgebrochen werden. Bekanntlich wird das Interrupt-Flag durch das Fangen der Ausnahme Interrupted-Exception wieder zurückgesetzt. Dies gilt aber für das Cancel-Flag nicht. Dieses ist also auch nach dem Catch-Block immer noch gesetzt. In der Dokumentation zu JavaFX wird an mehreren Stellen dringend empfohlen, die Tasks so zu programmieren, dass sie abgebrochen werden können.

Als ein einfaches Beispiel zur Nutzung des Task-Konzepts verändern wir die Ablaufsteuerung (Presenter-Teil) unserer MVP-Anwendung einer Stoppuhr aus Listing 4.11 (Modell- und Darstellungskomponente bleiben unverändert). Die Klasse TickerThread wird nicht mehr benötigt. Stattdessen gibt es nun die aus Task abgeleitete Klasse TickerTask. Da wir die View-Klasse ClockView, die einen ClockPresenter als Konstruktor-Parameter besitzt, unverändert lassen wollen, leiten wir unsere neue Ablaufsteuerungsklasse ClockPresenter-Task von ClockPresenter ab. Das hat außerdem den Vorteil, dass wir nur zwei Methoden, nämlich start und stop, implementieren müssen, denn die restlichen Methoden setModel-AndView, reset und exit können unverändert übernommen werden. Weil in start ein Zugriff auf das Modell vom Typ Clock und die Darstellungskomponente vom Typ ClockView nötig ist, wurden diese Attribute in Listing 4.11 bereits protected deklariert. Die veränderte Stoppuhr-Variante ist in Listing 4.12 zu finden.

#### **Listing 4.12**

```
import javafx.application.*;
import javafx.concurrent.*;
import javafx.scene.Scene;
import javafx.stage.Stage;

/* Presenter */
class TickerTask extends Task<Long>
{
    private final static long UPDATE_INTERVAL = 10; // Milliseconds
    private Clock clock;

    public TickerTask(Clock clock)
    {
        this.clock = clock;
    }

    protected Long call()
    {
        while(!isCancelled())
        {
            updateValue(clock.getTime());
            try
            {
                Thread.sleep(UPDATE_INTERVAL);
            }
            catch(InterruptedException e)
            {

```

```
        }
    }
    return clock.getTime();
}

class ClockPresenterTask extends ClockPresenter
{
    private TickerTask task;

    public void start()
    {
        if(task == null)
        {
            clock.reset();
            task = new TickerTask(clock);
            task.valueProperty().addListener((obs, oldVal, newVal)
                -> view.showTime(newVal));
            Thread thread = new Thread(task);
            thread.setDaemon(true);
            thread.start();
        }
    }

    public void stop()
    {
        if(task != null)
        {
            task.cancel();
            task = null;
        }
    }
}

/* Main */
public class ClockManagerTask extends Application
{
    public void start(Stage primaryStage)
    {
        ClockPresenterTask p = new ClockPresenterTask();
        ClockView view = new ClockView(p);
        Clock clock = new Clock();
        p.setModelAndView(clock, view);
        p.reset();

        Scene scene = new Scene(view.getUI());
        primaryStage.setScene(scene);
        primaryStage.setTitle("Stoppuhr mit Task");
        primaryStage.show();
    }

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

In Listing 4.12 erfolgt die Kommunikation zwischen der TickerTask und dem „JavaFX Application Thread“ über die Value-Property, wobei der generische Value-Typ in diesem Fall auf Long gesetzt ist. In der Methode start des neuen Presenters wird an die Value-Property ein Listener angemeldet. Dieser Listener wird immer dann aufgerufen, wenn sich die Value-Property geändert hat. Daraufhin wird der neue Wert an die View gegeben. An dieser Stelle sei nochmals daran erinnert, dass der Listener vom „JavaFX Application Thread“ ausgeführt wird und somit in der Methode showTime der View-Klasse auf die Oberfläche zugegriffen werden darf. Der Wert der Value-Property wird in der Methode call der TickerTask durch updateValue verändert. Die Methoden der Modellklasse Clock müssen nun aber synchronisiert werden. Den Grund für diese hier nicht gezeigte Änderung werden Sie sicher selber herausfinden.

Bitte beachten Sie noch dieses kleine Detail: Zu Listing 2.17 wurde gesagt, dass die While-Schleife in den Try-Block geschachtelt sein sollte, da die umgekehrte Schachtelung oft nicht funktionieren würde. In der TickerTask wurde zu Demonstrationszwecken ganz bewusst die umgekehrte Schachtelung gewählt (der Try-Block befindet sich also in der While-Schleife). Dies führt in diesem Fall aber zu keinen Problemen, da nicht das Interrupt-Flag, sondern das Cancel-Flag abgefragt wird, was im Falle des Fangens des Interrupts gesetzt bleibt und nicht wie das Interrupt-Flag zurückgesetzt wird. Natürlich hätte die ursprüngliche Schachtelung auch weiterhin funktioniert.

Im vorhergehenden Beispiel wurde die Value-Property verwendet. Auf Änderungen dieser Property reagierte ein Listener durch Awendung der Methode showTime auf die View mit dem neuen Wert der Property als Parameter. Speziell für die Progress-Property einer Task würde sich auch eine direkte Kopplung mit der Progress-Property eines ProgressBars (JavaFX-Element zur Fortschrittsanzeige als Balken) oder eines ProgressIndicators (JavaFX-Element zur Fortschrittsanzeige als Kreis) anbieten.

Wie oben schon erwähnt wurde, kann mit Hilfe einer Task eine einmalige Ausführung einer länger dauernden Aktivität realisiert werden. Im Beispielprogramm von Listing 4.12 musste deshalb für jedes Neuanstoßen der länger dauernden Aktivität ein neues Task-Objekt erzeugt werden. Folglich musste anschließend an die Value-Property des neuen Task-Objekts ein Listener angemeldet werden. Mit Hilfe eines Services wird die Aufgabe ein klein wenig einfacher. Ein Service ist für die wiederholte Ausführung länger dauernder Aktivitäten vorgesehen. Dabei stellt ein Service keinen Ersatz für eine Task dar, sondern eine Ergänzung, denn bei der wiederholten Ausführung erzeugt der Service jedes Mal ein neues Task-Objekt. Damit dies möglich ist, muss aus der abstrakten Klasse Service eine eigene Klasse abgeleitet und darin die abstrakte Methode createTask überschrieben werden, die bei jedem Aufruf ein neues Task-Objekt zurückliefern muss. Die generischen Typparameter von Task und Service müssen übereinstimmen. Im Gegensatz zu einer Task kann man einen Service direkt mit der Methode *start* starten. Für das Thread-Management kann man dem Service einen Executor angeben. Tut man dies nicht, so wird beim Starten des Service automatisch ein neuer Hintergrund-Thread (Daemon Thread) erzeugt und gestartet. Ein erneutes Starten des Service ist erst möglich, wenn der Service nicht mehr aktiv ist (das kann über die Methode *isRunning* abgefragt werden) und wenn zuvor die Methode *reset* aufgerufen wurde. Man kann auch die Methode *restart* verwenden, die einen eventuell noch aktiven Service zuerst mit *cancel* stoppt und anschließend *reset* und *start* aufruft.

In der Regel muss nur ein einziges Service-Objekt erzeugt werden, da damit die wiederholte Ausführung von Tasks möglich ist. Ein Service besitzt dieselben Properties wie eine Task. Wenn man nun an eine Property eines Service einen Listener anmeldet oder eine andere Property an diese Property des Service koppelt, dann gilt dies für die jeweilige Property der vom Service erzeugten Tasks. In Listing 4.13 variiieren wir das Stoppuhrbeispiel so, dass jetzt ein Service benutzt wird. Die aus Service abgeleitete Klasse TickerService ist sehr einfach: Im Wesentlichen wird createTask überschrieben und darin jeweils ein neues TickerTask-Objekt erzeugt und zurückgeliefert (TickerTask findet man in Listing 4.12).

**Listing 4.13**

```
import javafx.application.*;
import javafx.concurrent.*;
import javafx.scene.Scene;
import javafx.stage.Stage;

/* Presenter */
class TickerService extends Service<Long>
{
    private Clock clock;

    public TickerService(Clock clock)
    {
        this.clock = clock;
    }

    protected Task<Long> createTask()
    {
        return new TickerTask(clock);
    }
}

class ClockPresenterService extends ClockPresenter
{
    private TickerService service;

    public void setModelAndView(Clock clock, ClockView view)
    {
        super.setModelAndView(clock, view);
        service = new TickerService(clock);
        service.valueProperty().addListener((obs, oldVal, newVal)
            -> show(newVal));
    }

    private void show(Long newVal)
    {
        if (newVal != null)
        {
            view.showTime(newVal);
        }
    }

    public void start()
    {
        if (!service.isRunning())
        {
```

```

        clock.reset();
        service.reset();
        service.start(); //oder restart ohne reset
    }
}

public void stop()
{
    service.cancel();
}
}

/* Main */
public class ClockManagerService extends Application
{
    public void start(Stage primaryStage)
    {
        ClockPresenterService p = new ClockPresenterService();
        ClockView view = new ClockView(p);
        Clock clock = new Clock();
        p.setModelAndView(clock, view);
        p.reset();

        Scene scene = new Scene(view.getUI());
        primaryStage.setScene(scene);
        primaryStage.setTitle("Stoppuhr mit Service");
        primaryStage.show();
    }

    public static void main(String[] args)
    {
        launch(args);
    }
}
}

```

Neben TickerService bringt nur die Klasse ClockPresenterService noch etwas Neues: In der Initialisierungsmethode setModelAndView wird einmalig ein TickerService erzeugt und an dessen Value-Property ein Listener angemeldet, was dann auch für die Value-Properties der im Folgenden erzeugten TickerTask-Objekte gilt.

Bei einem Service muss über start bzw. restart das erneute Ausführen einer Task explizit angestoßen werden. Mit Hilfe der Klasse ScheduledService wird die automatisch wiederholte Ausführung von Tasks in einem vorgebbaren zeitlichen Abstand möglich. Wie bei Service muss auch aus ScheduledService eine eigene Klasse abgeleitet und darin die Methode createTask überschrieben werden. Um jetzt für ScheduledService kein komplett neues Beispiel erklären zu müssen, bleiben wir bei unserer Stoppuhr. Allerdings implementieren wir eine neue Task-Klasse namens SingleTickerTask, die nur einen einzigen Tick ausführt (und also nicht wie bisher eine Schleife beinhaltet). Die aus ScheduledService abgeleitete Klasse namens ScheduledTickerService liefert in createTask solche SingleTickerTask-Objekte zurück. Es wird ein Objekt dieser ScheduledTickerService-Klasse erzeugt und durch Anwendung von setPeriod wird eingestellt, wie häufig eine Task gestartet werden soll. Das heißt, wir benötigen kein Sleep mehr in unserem Programmcode, sondern überlassen dies dem ScheduledService. Listing 4.14 zeigt den Programmcode.

**Listing 4.14**

```
import javafx.application.*;
import javafx.concurrent.*;
import javafx.scene.Scene;
import javafx.stage.Stage;
import javafx.util.Duration;

/* Presenter */
class SingleTickerTask extends Task<Long>
{
    private Clock clock;

    public SingleTickerTask(Clock clock)
    {
        this.clock = clock;
    }

    protected Long call()
    {
        return clock.getTime();
    }
}

class ScheduledTickerService extends ScheduledService<Long>
{
    private Clock clock;

    public ScheduledTickerService(Clock clock)
    {
        this.clock = clock;
    }

    protected Task<Long> createTask()
    {
        return new SingleTickerTask(clock);
    }
}

class ClockPresenterScheduledService extends ClockPresenter
{
    private final static long UPDATE_INTERVAL = 10; // Milliseconds

    private ScheduledTickerService service;

    public void setModelAndView(Clock clock, ClockView view)
    {
        super.setModelAndView(clock, view);
        service = new ScheduledTickerService(clock);
        service.valueProperty().addListener((obs, oldVal, newVal)
            -> show(newVal));
        service.setPeriod(Duration.millis(UPDATE_INTERVAL));
    }

    private void show(Long newVal)
    {
        if (newVal != null)
        {
```

```
        view.showTime(newVal);
    }
}

public void start()
{
    if (!service.isRunning())
    {
        clock.reset();
        service.reset();
        service.start();
    }
}

public void stop()
{
    service.cancel();
}
}

/* Main */
public class ClockManagerScheduledService extends Application
{
    public void start(Stage primaryStage)
    {
        ClockPresenterScheduledService p =
            new ClockPresenterScheduledService();
        ClockView view = new ClockView(p);
        Clock clock = new Clock();
        p.setModelAndView(clock, view);
        p.reset();

        Scene scene = new Scene(view.getUI());
        primaryStage.setScene(scene);
        primaryStage.setTitle("Stoppuhr mit Scheduled Service");
        primaryStage.show();
    }

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

Auf eine kleine Besonderheit sei abschließend noch hingewiesen: In der Methode call der Klasse SingleTickerTask befindet sich kein Aufruf von updateValue, wie dies in der bisherigen TickerTask der Fall war. Zur Erklärung sei daran erinnert, dass der von call zurückgegebene Wert in die Value-Property übernommen wird. Folglich reicht es aus, wenn call einen passenden Wert zurückliefert.

## ■ 4.6 Zusammenfassung

In diesem Kapitel wurde ein Überblick über die Programmierung grafischer Benutzeroberflächen mit JavaFX gegeben. Es wurde dann etwas ausführlicher auf das MVP-Architekturmuster und auf Wechselwirkungen zwischen Threads und grafischen Benutzeroberflächen eingegangen.

Obwohl es nicht auf den ersten Blick erkennbar ist, dient dieses Kapitel u. a. auch als Grundlage für den nun folgenden Teil über die Entwicklung verteilter Anwendungen. Zum einen werden wir in einigen Beispielen Clients mit einer grafischen Benutzeroberfläche ausstatten. Wenn dann eine Benutzeraktion eine Kommunikation mit einem Server auslöst, die länger dauern kann, muss nach den in diesem Kapitel eingeführten Prinzipien diese Kommunikation in einem eigenen Thread durchgeführt werden. Dieser Thread darf dann aber nichts selbst in einem Fenster anzeigen, sondern muss dazu den „JavaFX Application Thread“ beauftragen.

Zum anderen stellt das MVP-Architekturmuster nicht nur eine Strukturierungshilfe für rein lokale Anwendungen mit grafischer Benutzeroberfläche dar, sondern auch für verteilte Anwendungen.

# 5

# Verteilte Anwendungen mit Sockets

Ab diesem Kapitel wenden wir uns der Entwicklung verteilter Anwendungen zu. Damit verlassen wir die lokale Sichtweise, in der es um das Zusammenspiel von Threads innerhalb eines Prozesses ging, und beschäftigen uns nun mit dem Zusammenwirken von Threads unterschiedlicher Prozesse, die in der Regel auf unterschiedlichen Rechnern laufen. Mit unserer Küchen-Metapher aus Kapitel 1 kann dieser Sachverhalt folgendermaßen ausgedrückt werden: Ab jetzt betrachten wir die Interaktion von Köchen, die nicht wie bisher in derselben Küche, sondern in unterschiedlichen Küchen arbeiten. Die Küchen befinden sich dabei in der Regel in unterschiedlichen Gebäuden (Rechnern), können aber als Spezialfall auch im selben Gebäude liegen.

In diesem und dem folgenden Kapitel geht es um eigenständige Client-Server-Anwendungen. Damit ist gemeint, dass wir sowohl einen Client als auch einen dazu passenden Server selbst programmieren. In Kapitel 7 werden im Gegensatz dazu webbasierte Anwendungen behandelt, bei denen der Client ein Web-Browser ist und ein Web-Server um selbst geschriebene Programme, so genannte Servlets, erweitert wird. Diese Servlets erzeugen dynamische Webseiten. Auf Server-Seite ist somit kein vollständiges Programm, sondern nur eine Art Anhänger für einen Web-Server zu programmieren, auf Client-Seite sogar gar nichts.

Die Entwicklung eigenständiger Client-Server-Anwendungen werden wir auf zwei Arten kennen lernen:

- In diesem Kapitel geht es um die Socket-Programmierung. Die so genannte Socket-Schnittstelle bietet eine Programmierschnittstelle zur Verwendung der Transportprotokolle UDP und TCP an.
- Das folgende Kapitel dreht sich um RMI (Remote Method Invocation). Mit RMI kann von einem Rechner aus eine Methode eines Objekts aufgerufen werden, das sich auf einem anderen Rechner befindet. RMI hat zum Ziel, die Entwicklung verteilter Client-Server-Anwendungen zu erleichtern, indem auf diese Weise dem Entwickler die Kommunikationsaspekte so weit wie möglich verborgen bleiben. RMI basiert auf Sockets. Insofern stellt es eine höhere Schnittstelle mit einem höheren Abstraktionsniveau als Sockets dar.

Zunächst wird eine kurze Einführung in das Themengebiet der Rechnernetze gegeben, um für die darauf folgenden Ausführungen gerüstet zu sein.

## ■ 5.1 Einführung in das Themengebiet der Rechnernetze

### 5.1.1 Schichtenmodell

Die Übermittlung von Daten von einem Rechner zu einem anderen über ein Rechnernetz ist eine nicht triviale Aufgabe. Deshalb wird diese Aufgabe gemäß dem bekannten Prinzip „Teile und herrsche“ in mehrere Teilaufgaben zerlegt. Die Lösung jeder Teilaufgabe wird durch eine so genannte *Schicht* erfüllt. Wir wollen das *Schichtenmodell* zunächst an einer Metapher erläutern:

Eine Geschäftsführerin einer Firma X möchte einer anderen Firma Y einen Auftrag erteilen. Zu diesem Zweck holt sie bei der Geschäftsführerin der anderen Firma Y ein Angebot ein. Nach Prüfung dieses Angebots erteilt die Geschäftsführerin der Firma X der Firma Y den Auftrag. Diese Folge von Kommunikationsschritten wird als *Kommunikationsprotokoll* (oder kurz *Protokoll*) auf der Geschäftsführerinnenebene (Geschäftsführerinnenschicht) bezeichnet. Ein Protokoll legt fest, von welcher Art die ausgetauschten Informationen sind sowie alle zulässigen Folgen von Kommunikationsschritten. Das Protokoll zwischen den beiden Geschäftsführerinnen ist jedoch insofern virtuell, als sich in unserem Beispiel die beiden Geschäftsführerinnen nicht direkt miteinander unterhalten. Stattdessen erteilt die Geschäftsführerin der Firma X ihrem Sachbearbeiter den Auftrag, ein Angebot von der Firma Y einzuholen. Dabei teilt die Geschäftsführerin dem Sachbearbeiter lediglich die wichtigsten Informationen, u. a. den Namen der Firma Y, mündlich mit. Diese Interaktion zwischen Geschäftsführerin und Sachbearbeiter bildet die *Dienstschnittstelle* zwischen den beiden. Das heißt, der Sachbearbeiter stellt an dieser Schnittstelle einen Dienst zur Verfügung, der von der Geschäftsführerin durch Erteilung eines mündlichen Auftrags in Anspruch genommen wird. Der Sachbearbeiter führt den ihm erteilten Auftrag aus, indem er einen Brief an die Firma Y schreibt. Er fügt dabei weitere Informationen, wie genaue Anschrift, Absender, Datum, Anrede usw. hinzu und steckt diesen Brief in einen Umschlag. Dieser Brief wird bei der Firma Y von einem entsprechenden Sachbearbeiter entgegengenommen, der aus diesem Brief die relevanten Informationen herauszieht und diese über die Schnittstelle der Geschäftsführerin der Firma Y mitteilt. Das heißt, die vom Sachbearbeiter der Firma X hinzugefügte Information wird vom Sachbearbeiter der Firma Y wieder abgezogen, so dass die Information, die an der Schnittstelle der Geschäftsführerin der Firma Y übermittelt wird, derjenigen entspricht, die in der Firma X an der entsprechenden Schnittstelle als Auftrag übergeben wurde. Die Geschäftsführerin der Firma Y erhält also genau die von der Geschäftsführerin der Firma X angegebene Information.

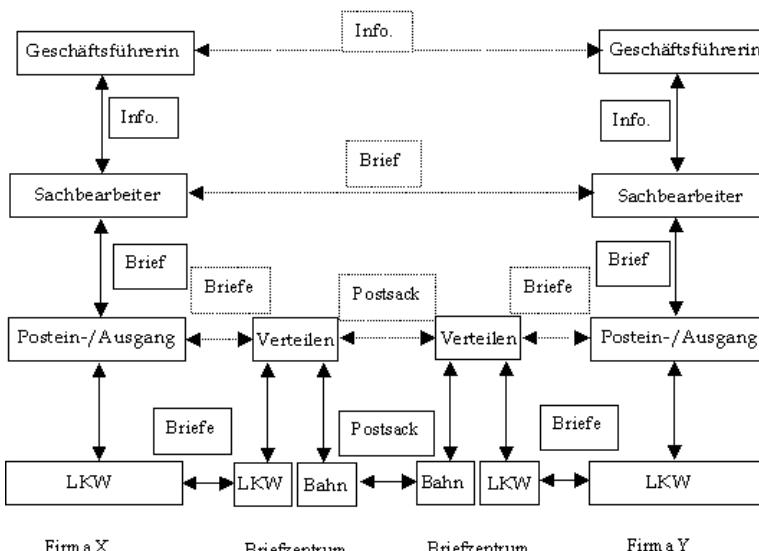
Wir haben in diesem Beispiel die wesentlichen Begriffe Protokoll, Schicht und Dienstschnittstelle kennengelernt: Das Protokoll innerhalb der Schicht der Geschäftsführerinnen wird durch Erteilung der entsprechenden Aufträge über die Schnittstelle zu den Sachbearbeitern durch ein Protokoll zwischen den Sachbearbeitern abgewickelt. Unser Beispiel soll den Leser nicht zu der falschen Annahme verleiten, dass jeder Auftrag an der Schnittstelle immer zum Senden genau einer Nachricht (in unserem Beispiel eines Briefes) auf der nächsten Schicht (der Sachbearbeiterschicht) führen muss. Es ist so z. B. denkbar, dass der Sachbearbeiter der Firma X von sich aus die Initiative ergreift, wenn er längere Zeit keine Antwort auf seine Anfrage erhält, und bei der Firma Y nachfragt, ob denn seine Anfrage

nicht eingetroffen sei. Der Sachbearbeiter der Firma Y könnte auch feststellen, dass noch Informationen fehlen, um das Angebot zu unterbreiten. Er würde dann um diese Informationen beim Sachbearbeiter der Firma X bitten. Diese zusätzlichen Kommunikationsschritte im Rahmen des Sachbearbeiterprotokolls finden ohne Kenntnis der beiden Geschäftsführerinnen statt. Die Geschäftsführerinnen brauchen sich somit nicht um dieses Problem zu kümmern, da sie wissen, dass ihre Sachbearbeiter die Kommunikation zuverlässig durchführen.

Wie schon zuvor die Geschäftsführerinnen, so treffen auch die Sachbearbeiter nicht wirklich aufeinander und übergeben sich die Briefe persönlich. Stattdessen wird der Übermittlungsdienst der Post verwendet. Schnittstelle zur Post sind ein Eingangs- und Ausgangskorb, aus dem von einem Bediensteten der Post die Briefe abgeholt bzw. in den Briefe, die für die Firma bestimmt sind, hineingelegt werden. Auf der Postschicht kommt nun allerdings der folgende neue Aspekt dazu: Die Briefe werden von dem Bediensteten der Post nicht direkt zur Firma Y befördert, sondern zunächst zum Briefzentrum der Region, in der die Firma X ihren Sitz hat. In diesem Briefzentrum werden die Briefe sortiert und zur Weiterleitung vorbereitet. Von diesem Briefzentrum wird unser Brief dann zum Briefzentrum der Region befördert, in dem die Firma Y ansässig ist. Und von dort gelangt er dann durch einen Postbediensteten in den Eingangskorb der Firma Y. Das Protokoll zwischen den Briefzentren besteht im Austausch von Briefen in Postsäcken.

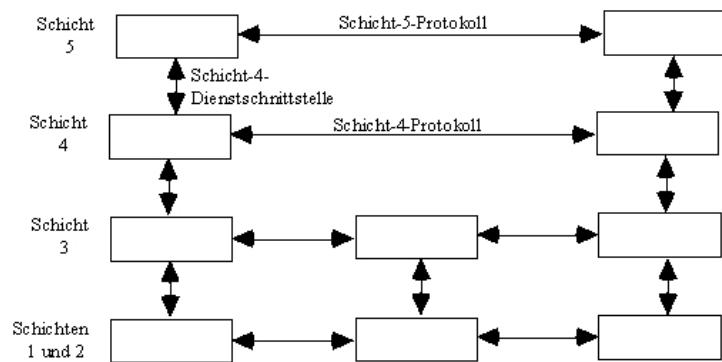
Diese Postsäcke wiederum werden von den Angestellten der Briefzentren nicht selbst zum nächsten Briefzentrum gebracht, sondern dazu werden unterschiedliche Beförderungssysteme wie etwa LKW, Bahn, Schiff oder Flugzeug eingesetzt. Die entsprechende Schnittstelle und das Protokoll mögen sich die Lesenden selbst überlegen.

Die beschriebene Unterteilung des Kommunikationsvorgangs in Schichten ist in Bild 5.1 zusammengefasst. Entlang der durchgezogenen Linien wird wirklich etwas befördert, während die gestrichelten Linien den Beförderungsweg zeigen, so wie es von den Betroffenen der entsprechenden Schicht gesehen wird. Für die Sachbearbeiter z.B. scheint es so, als würden sie direkt miteinander einen Brief austauschen.



**Bild 5.1** Schichtenmodell anhand eines Beispiels aus dem täglichen Leben

Eine abstrakte Form von Bild 5.1 sehen Sie in Bild 5.2. Dabei ist nur eine Zwischenstation statt der beiden Briefzentren gezeigt. Gemeint ist, dass eine beliebig große Zahl von Zwischenstationen vorhanden sein kann. Außerdem sind die Schichten „von unten nach oben“ durchnummieriert. Dabei wurden die untersten Schicht die Zahlen 1 und 2 zugeordnet, damit die Nummerierung mit derjenigen der Internet-Protokolle übereinstimmt. In unserem Beispiel entsprechen die Schichten 1 und 2 der Briefbeförderungsschicht, Schicht 3 der Postschicht, Schicht 4 der Sachbearbeiterschicht und Schicht 5 schließlich der Geschäftsführerinnenschicht.



**Bild 5.2** Schichtenmodell in abstrakter Darstellung

Im Internet haben die Schichten die folgenden Funktionen:

- Die Schichten 1 und 2 sind im Internet zuständig für die Übertragung von Datenblöcken zwischen zwei Rechnern, die direkt miteinander verbunden sind, also beispielsweise über eine Telefonleitung, über ein drahtgebundenes lokales Netz des Typs *Ethernet* oder über ein *drahtloses Funknetz*, an das beide angeschlossen sind. Die erste Schicht ist dabei zuständig für die Übertragung einzelner Bits und wird daher *Bitübertragungsschicht (Physical Layer)* genannt. Die zweite Schicht, die als *Leitungsschicht (Data Link Layer)* bezeichnet wird, hat die Aufgabe, einzelne Bits zu einem Datenblock in einem bestimmten Format zusammenzufassen. Die Schichten 1 und 2 sind von Netztyp zu Netztyp unterschiedlich. So werden die Bits 0 und 1 durch ein Modem anders codiert als durch einen Ethernet-Adapter. Außerdem werden die Daten je nach benutzter Netztechnologie unterschiedlich formatiert. Die Formatierung bedeutet, dass bestimmte Informationen wie z. B. Adressen eine gewisse Struktur und Bitlänge haben und an einer bestimmten Stelle im Datenblock stehen. Bei den lokalen Netzen wie Ethernet kommen durch das gemeinsame Medium noch zwei weitere Funktionen auf der Leitungsschicht hinzu: Zum einen wird ein Verfahren benötigt, welches garantiert, dass immer nur höchstens eine Station sendet. Zum anderen müssen die an dem gemeinsamen Medium angeschlossenen Rechner adressiert werden. Bei einer Punkt-zu-Punkt-Leitung kann eine empfangende Station davon ausgehen, dass die ankommenden Daten für sie bestimmt sind, bei einem gemeinsamen Medium dagegen nicht. Diese beiden Funktionen werden als *Medium-Zugangskontrolle (Medium Access Control oder kurz MAC)* bezeichnet und bilden eine Teilschicht innerhalb der Leitungsschicht. Die entsprechenden Adressen heißen deshalb *MAC-Adressen*.
- Das *Internet*, das „Netz der Netze“, verbindet Rechner miteinander, die an verschiedenen Netzen mit unterschiedlichen Netztechnologien angeschlossen sind. Das Protokoll, wel-

ches dies zu leisten vermag, war der Namenspatron des Internet: das *Internet Protocol* oder kurz *IP*. Das IP-Protokoll ist auf der dritten Schicht, die im Allgemeinen als *Vermittlungsschicht (Network Layer)* bezeichnet wird, angesiedelt. In dieser Schicht wird für eine einheitliche, strukturierte und weltweit eindeutige Adressierung gesorgt; jedem Netzanschluss wird eine eindeutige *IP-Adresse* zugeordnet. Mit Hilfe des IP-Protokolls werden Datenpakete über verschiedene Netze weitergeleitet, bis sie beim Zielrechner ankommen. Dabei wird in jedem Rechner die Adresse erneut gelesen und der nächste Rechner bestimmt. Ein Datenpaket kann wie ein Brief bei der Post auch einmal verloren gehen. Weiterhin kann es – wiederum wie bei der Post – passieren, dass ein Datenpaket verzögert ausgeliefert wird. Die Übertragungszeit schwankt also im Allgemeinen. Dadurch kann es auch vorkommen, dass ein als zweites gesendetes Datenpaket vor dem zuerst abgeschickten Datenpaket beim Empfänger ankommt und die Datenpakete somit in ihrer Reihenfolge vertauscht werden.

- Auf der dritten Schicht werden Rechner adressiert. Auf der vierten Schicht, der *Transportschicht (Transport Layer)*, wird diese Adressierung ergänzt um „Kontaktpunkte“ innerhalb des Rechners. Im Beispiel aus dem täglichen Leben entspricht die Adressierung auf der dritten Schicht dem Teil Firma, Postleitzahl, Ort, Straße und Hausnummer (oder Postfach). Mit der Adressierung auf der vierten Schicht wird eine Abteilung oder ein Mitarbeiter innerhalb der Firma (oder eine Person einer Familie oder WG) gezielt angesprochen. Die Adressierung auf der vierten Schicht erfolgt durch so genannte *Portnummern*. Im Internet gibt es zwei verbreitete Transportprotokolle: *UDP (User Datagram Protocol)* und *TCP (Transmission Control Protocol)*. UDP hat neben der Adressierung mit Portnummern keine weitere Funktion. Es übernimmt daher die Eigenschaften des IP-Protokolls; wie IP ist es *verbindungslos* und *unzuverlässig* (d. h. *Verluste* und *Reihenfolgevertauschungen* sind möglich). Im Gegensatz dazu behebt das TCP-Protokoll die Nachteile von IP. Dabei überwacht die eine Seite, ob die andere Seite nach gegebener Zeit den Empfang einer zuvor gesendeten Nachricht bestätigt. Falls dies nicht der Fall ist, wird die Nachricht nochmals gesendet. Aus der Sicht der Transportprotokolle bleibt verborgen, dass die Kommunikation unter Umständen über mehrere Zwischenstationen abgewickelt wird. Für sie scheint es so, als seien sie in direktem Kontakt miteinander, da sie den Dienst der darunter liegenden IP-Schicht nutzen.
- In der fünften Schicht, der *Anwendungsschicht (Application Layer)*, wird die Kommunikation der Anwendungen abgewickelt. Solche Anwendungen sind z. B. das *World Wide Web (WWW)*, *elektronische Post* oder *Audio-Video-Konferenzen*. Auch können selbst definierte Anwendungsprotokolle, wie sie z. B. auch in diesem Kapitel definiert werden, zum Einsatz kommen.

Die soeben beschriebenen Schichten haben ihre konkrete Entsprechung in einem Rechner durch Software-Module und Hardware-Bausteine. Typischerweise sind die Schichten 1 und 2 durch Adapterkarten (z. B. Ethernet-Adapter) und die entsprechenden *Treiber* eines Betriebssystems realisiert. Die dritte und vierte Schicht sind in der Regel im Betriebssystemkern eingebettet, während die fünfte Schicht durch Anwendungsprozesse realisiert wird. Oft werden die Software-Module, die das UDP-, TCP- bzw. das IP-Protokoll im Rechner implementieren, auch kurz als UDP, TCP bzw. IP bezeichnet. So ist mit einem Satz wie „TCP nimmt die Daten entgegen“ gemeint: „Das Software-Modul, das das TCP-Protokoll implementiert, nimmt die Daten entgegen.“

### 5.1.2 IP-Adressen und DNS-Namen

Jedem Netzanschluss, wovon ein Rechner mehrere haben kann (z.B. einen Ethernet-Anschluss und einen Anschluss an ein drahtloses Netz), ist eine weltweit eindeutige *IP-Adresse* zugeordnet. In der Version 4 von IP (*IPv4*) besteht diese Adresse aus 32 Bits. Eine IP-Adresse wird für die menschlichen Nutzer in der so genannten „*punktierten Dezimalnotation*“ dargestellt. Dabei werden je 8 Bits als Dezimalzahl geschrieben, und diese Zahlen werden mit Punkten voneinander getrennt (z.B. 143.93.53.147).

Die Adressen werden in unterschiedliche Klassen eingeteilt, wobei wir auf die Unterschiede zwischen den Klassen A, B und C hier nicht eingehen wollen. Die erste Zahl der Adressen dieser Klassen A, B und C ist kleiner als 224. Damit werden immer einzelne Rechner angeprochen (*Unicast-Adressen*). Ist die erste Zahl im Bereich von 224 bis 239 einschließlich, so handelt es sich um eine Adresse der Klasse D, einer so genannten Multicast-Adresse. Damit kann eine Gruppe von Rechnern adressiert werden. Die Mitglieder einer solchen Gruppe sind diejenigen Rechner, die der entsprechenden Gruppe beigetreten sind. Neben seiner eigenen individuellen Adresse kann ein Rechner damit auch noch eine oder mehrere *Multicast-Adressen* haben. IP-Adressen, deren erste Zahl zwischen 240 und 255 liegt (größer als 255 ist wegen der 8 Bits nicht möglich), gehören zur Adressklasse E; sie gehören zu einem reservierten Bereich und werden in der Regel nicht benutzt. In der Version 6 von IP (*IPv6*) besteht eine IP-Adresse aus 128 Bits.

Da IP-Adressen für Menschen nicht leicht zu merken sind, wurden für die Rechner auch Namen eingeführt. Diese Namen haben eine hierarchische Struktur, wobei im Gegensatz zu Dateipfadnamen die Wurzel und die höchste Stufe der Hierarchie am Ende stehen. Anders ausgedrückt: Wenn man sich die Hierarchie als Baum vorstellt, wobei die Wurzel oben gezeichnet wird, dann beschreibt ein Dateipfadname einen Weg „von oben nach unten“ (d.h. von der Wurzel ausgehend), während ein Rechnername umgekehrt einem Pfad „von unten nach oben“ entspricht. Die einzelnen Komponenten sind durch Punkte voneinander getrennt (Beispiel: www.hochschule-trier.de, dabei ist „de“ die oberste Hierarchiestufe).

Die Namen werden vom so genannten *Domain Name System (DNS)* in IP-Adressen umgesetzt. Aus diesem Grund heißen die Rechnernamen auch DNS-Namen. Das DNS-System besteht aus einer Vielzahl von DNS-Servern, die sich zum Teil gegenseitig kennen und befragen. Einem Rechner können mehrere DNS-Namen zugeordnet sein (so genannte Aliase). Umgekehrt kann ein Rechner, der durch einen DNS-Namen identifiziert wird, mehrere IP-Adressen haben (z.B. bei mehreren Netzanbindungen). Das heißt, dass zwischen IP-Adressen und Rechnernamen eine m:n-Beziehung besteht.

Ein Sonderfall stellt eine IP-Adresse bzw. ein DNS-Name dar, der bei uns Menschen der Bezeichnung „ich“ entspricht. Die IP-Adresse 127.0.0.1 bzw. der DNS-Name *localhost* meint immer den eigenen Rechner.

### 5.1.3 Das Transportprotokoll UDP

In Abschnitt 5.1.1 wurde bereits erläutert, warum es Transportprotokolle gibt:

- Zum einen laufen in einem Rechner mehrere Anwendungen, die über das Internet kommunizieren, (quasi) gleichzeitig ab. Die ankommenden Daten müssen an die Anwendungen verteilt werden.
- Zum anderen besitzt IP einige Eigenschaften, die aus Anwendersicht nicht gerade wünschenswert sind. So können Sie sich leicht ausmalen, dass bei der Übertragung einer Datei (z.B. eines WWW-Dokuments) der Verlust oder die Reihenfolgevertauschung von IP-Paketen den Anwender nicht erfreuen dürfte. Die einzelnen Anwendungen könnten natürlich geeignete Gegenmaßnahmen treffen. Da diese Gegenmaßnahmen aber in die meisten der neu zu entwickelnden Anwendungen eingebaut werden müssten, ist es einfacher, wenn diese Problematik durch ein Transportprotokoll gelöst wird, das von vielen Anwendungen benutzt wird.

Das Transportprotokoll *UDP (User Datagram Protocol)* löst nur das erste der beiden geschilderten Problembereiche. Es erweitert die Funktionalität von IP lediglich um die Möglichkeit, eine spezifische Anwendung auf einem Rechner über eine so genannte *Portnummer* zu adressieren.

Mit dem Begriff *Datagramm* (ein Kunstwort, das aus Telegramm abgeleitet ist) wird eine Dateneinheit bezeichnet, die in ein IP-Paket gepackt und über das Netz verschickt wird. Ein UDP-Datagramm enthält neben der Zielportnummer auch die Quellportnummer. Die Quellportnummer wird vom Sender aus demselben Grund wie die IP-Quelladresse angegeben, damit nämlich der Empfänger dem Sender antworten kann.

Darüber hinaus fügt UDP dem IP-Protokoll keine weiteren Funktionen hinzu. Das bedeutet, dass alle Eigenschaften des IP-Protokolls erhalten bleiben. Die Eigenschaften von UDP sind somit:

- UDP ist wie IP *verbindungslos*.
- Bei der Benutzung von UDP können *Datagrammverluste* oder *Reihenfolgevertauschungen von Datagrammen* vorkommen. Außerdem kann ein Sender einen Empfänger mit Nachrichten „überfluten“, d.h. der Empfänger kann die Nachrichten nicht mit der Geschwindigkeit abarbeiten, mit der der Sender sie schickt.
- UDP ist *datagrammorientiert*. Damit ist Folgendes gemeint: Falls in einer Anwendung über eine Programmierschnittstelle eine gewisse Menge an Daten dem UDP-Protokoll übergeben wird, so werden diese Daten in genau ein Datagramm gesetzt und über IP verschickt. Die Programmierschnittstelle bietet auch Funktionen, um Daten entgegenzunehmen. Wenn nun auf der Seite des Empfängers Daten entgegengenommen werden sollen, so werden der Anwendung alle Daten des eingetroffenen Datagramms übergeben. Dies bedeutet, dass genau die Daten, die vom Sender auf einmal an UDP übergeben wurden, beim Empfänger als eine Einheit ankommen, falls sie nicht verloren gehen. Dieses Verhalten entspricht exakt dem Verhalten einer Message Queue (s. Abschnitt 3.2).

Man kann sich fragen, welchen Nutzen ein Protokoll hat, bei dem Daten verloren gehen oder in ihrer Reihenfolge vertauscht werden können. Entgegen dem ersten Eindruck gibt es dennoch einige sinnvolle Einsatzgebiete für UDP:

- Es gibt Anwendungen, bei denen der Verlust von Daten nicht so gravierend ist. Bei der

Übertragung eines Videos über das Internet (wie z.B. bei einer Audio-Video-Konferenz) werden ca. 20 digitalisierte, in der Regel auch komprimierte Bilder pro Sekunde versendet. Wenn dabei ab und zu einmal eines verloren geht, so ist das für den Betrachter kaum wahrnehmbar. Würde allerdings wie bei TCP der Verlust bemerkt und das Bild nach einer gewissen Wartezeit wiederholt übertragen, so wäre das für den menschlichen Betrachter weitaus störender.

- TCP ist ein Protokoll für genau zwei Partner. Deshalb kann TCP nicht mit Multicast genutzt werden. Man benötigt für Multicast daher ein neues Transportprotokoll oder kann UDP verwenden.
- TCP ist im Gegensatz zu UDP verbindungsorientiert. Der Verbindungsauf- und -abbau erfordert einen gewissen Aufwand. Wenn man nur wenige Daten zu übertragen hat (z.B. eine kurze Anfrage und eine kurze Antwort), ist dieser Aufwand relativ groß. Aus diesem Grund kann in solchen Situationen sinnvollerweise UDP eingesetzt werden, auch wenn Verluste und Reihenfolgevertauschungen nicht tolerierbar sind. Es werden dann Mechanismen in die Anwendung gegen diese Phänomene eingebaut, die aber in der Regel wesentlich einfacher sind als die ausgefeilten Mechanismen von TCP.

#### 5.1.4 Das Transportprotokoll TCP

*TCP (Transmission Control Protocol)* erlaubt nicht nur die gleichzeitige Internet-Nutzung mehrerer Anwendungen auf einem Rechner, sondern beinhaltet eine ganze Reihe weiterer Eigenschaften. TCP stellt das Gegenstück zu UDP dar:

- TCP ist im Gegensatz zum verbindungslosen UDP ein *verbindungsorientiertes Transportprotokoll*. Dies bedeutet, dass zwei Partner, die Daten austauschen wollen, zuvor eine Verbindung aufbauen müssen, die am Ende wieder abgebaut wird. Die Verbindungsorientierung ist kein Selbstzweck, sondern erleichtert die Realisierung der im nächsten Punkt genannten Zuverlässigkeit.
- TCP ist im Gegensatz zu UDP ein *zuverlässiges Protokoll*. Dies bedeutet, dass keine Daten verloren oder vertauscht werden können. Daneben besitzt es noch eine Reihe hilfreicher Mechanismen wie Flusskontrolle und Überlastkontrolle. Die *Flusskontrolle* verhindert eine Überflutung eines Empfängers mit Daten durch einen Sender, während die *Überlastkontrolle* einer Überlastung des Netzes entgegenwirkt.
- Im Gegensatz zum datagrammorientierten UDP ist TCP ein *datenstromorientiertes Protokoll*. Damit ist gemeint, dass der Empfänger einen Datenstrom entgegennimmt, dem er nicht ansieht, in welchen Portionen die Daten vom Sender geschickt wurden. Ein solches Verhalten haben Sie bereits in Abschnitt 3.3 bei den Pipes kennen gelernt.

Da bei den meisten Datenübertragungen Zuverlässigkeit verlangt wird und die besonderen Randbedingungen, in denen UDP eingesetzt wird, nicht vorliegen, wird in den meisten Anwendungen TCP eingesetzt. Dies gilt z.B. auch für die Übertragung von Dokumenten im Rahmen des WWW und die Übertragung elektronischer Post.

Zusammenfassend sind in **Tabelle 5.1** die wichtigsten Eigenschaften der beiden Transportprotokolle einander gegenübergestellt.

**Tabelle 5.1** Vergleich zwischen UDP und TCP

UDP	TCP
verbindungslos	verbindungsorientiert
unzuverlässig	zuverlässig mit Fluss- und Überlastkontrolle
datagrammorientiert	datenstromorientiert

## ■ 5.2 Socket-Schnittstelle

Die *Socket-Schnittstelle* wurde Anfang der 80er Jahre entwickelt, als Forscher der Berkeley-Universität (bei San Francisco) das UNIX-Betriebssystem um Netzfunktionalitäten erweiterten. Auch wenn es für viele heute kaum mehr vorstellbar ist, so muss man sich zum Verständnis dieser Entwicklung klar machen, dass ein Computer damals nur in den seltensten Fällen mit einem Netzanschluss ausgestattet war. Insbesondere integrierten die Forscher die TCP/IP-Protokolle in den UNIX-Betriebssystemkern. Um die nun vorhandene Kommunikationsfähigkeit nutzen zu können, erweiterten sie die UNIX-Kernschnittstelle entsprechend. Genau diese Erweiterung ist die so genannte Socket-Schnittstelle, die primär für Programme gedacht war, die in der Programmiersprache C – der „UNIX-Programmiersprache“ – geschrieben waren. Die ersten Internet-Anwendungen wie TELNET, FTP und elektronische Post basierten auf dieser Socket-Schnittstelle. Socket bedeutet übrigens Steckdose. Mit Hilfe dieser Metapher kann man sich gut vorstellen, dass die Socket-Schnittstelle eine „Kommunikationssteckdose“ für verteilte Client-Server-Anwendungen darstellt.

Die Arbeiten der Berkeley-Forscher waren historisch sehr bedeutsam, denn mit der raschen Verbreitung dieser BSD-UNIX-Versionen (BSD steht für Berkeley Software Distribution) trat das Internet seinen Siegeszug an. Diese ersten Erfolge führten dazu, dass die Socket-Schnittstelle in ähnlicher Form auch in anderen Betriebssystemen wie z.B. in Windows von Microsoft, in MacOS von Apple und in Großrechner-Betriebssystemen von IBM implementiert wurde und die entsprechenden Internet-Anwendungen auf diese Betriebssysteme portiert wurden. Die Internet-Kommunikationsfähigkeit eines Computers wurde damit immer mehr zum Standard.

Die Socket-Schnittstelle ist eine Schnittstelle zwischen der Transportschicht (Schicht 4) und der Anwendungsschicht (Schicht 5). In vielen Fällen bildet sie auch eine Schnittstelle zwischen dem Betriebssystemkern und den Anwendungen. Mit Hilfe der Socket-Schnittstelle können Anwendungen programmiert werden, die über UDP oder über TCP kommunizieren.

### 5.2.1 Socket-Schnittstelle zu UDP

Im Prinzip ist die Schnittstelle zu UDP sehr einfach. Da UDP nämlich verbindungslos ist, sind keine Operationen zum Verbindungsauf- und -abbau nötig, sondern lediglich zum Senden und Empfangen von Daten. Der zentrale Begriff ist der Socket. Einen Socket können Sie

sich als eine Steckdose vorstellen, über die wie über eine Telefonsteckdose Daten gesendet und empfangen werden können. An einen Socket kann eine Portnummer gebunden werden. Der Effekt davon ist, dass alle über diesen Socket gesendeten Datagramme die Portnummer dieses Sockets als Quellportnummer tragen. Außerdem können über diesen Socket nur Datagramme empfangen werden, die die Portnummer des Sockets als Zielportnummer enthalten.

Mit Hilfe der Socket-Schnittstelle lassen sich verteilte Anwendungen programmieren. Verteilte Anwendungen sind in der Regel *Client-Server-Anwendungen*, in denen einer der beiden kommunizierenden Partner die Rolle des Clients und der andere die Rolle des Servers übernimmt. Der *Client (Kunde, Auftragnehmer)* ist ein Prozess, der von einem Server eine bestimmte Dienstleistung anfordert. Die Initiative geht dabei immer vom Client aus. Bei der Benutzung von UDP bedeutet dies, dass der Client derjenige ist, der die Initiative ergreift und mit der Kommunikation anfängt. Der *Server (Diensterbringer)* ist ein Prozess, der typischerweise in einer Endlosschleife auf Aufträge von Kunden wartet, die ihm in Nachrichten übermittelt werden, diese Aufträge bearbeitet und eine Antwort an den Kunden zurücksendet.

Da der Client die Initiative ergreift, muss der Client die Adresse des Servers (d.h. dessen IP-Adresse und Portnummer) kennen. Die Portnummer ergibt sich aus der Art des in Anspruch zu nehmenden Dienstes. Für die verschiedenen Standarddienste sind nämlich internettweit gewisse Portnummern definiert, die als *wohlbekannte Portnummern (well-known port numbers)* bezeichnet werden. Der Client dagegen benötigt keine wohlbekannte Portnummer. Der Server kann dem Client antworten, denn mit der empfangenen Nachricht erfährt er die Absenderadresse (IP-Adresse und Portnummer). Der Client verwendet deshalb in der Regel eine beliebige Portnummer.

Ein Client-Programm hat damit typischerweise folgende Struktur:

```
erzeuge einen UDP-Socket s;
/* dabei besitzt der Socket irgendeine, im Moment noch nicht
   benutzte Portnummer
*/
wiederhole so oft wie nötig:
{
    sende über s eine Anfrage an eine vorgegebene IP-Adresse
        und Portnummer;
    warte auf eine Antwort am UDP-Socket s;
    analysiere die Antwort;
}
```

Ein Server-Programm ist in der Regel wie folgt aufgebaut:

```
erzeuge einen UDP-Socket s mit einer spezifischen Portnummer;
wiederhole immer wieder:
{
    warte auf eine Anfrage am UDP-Socket s;
    analysiere die Anfrage;
    führe entsprechend der Anfrage eine Aktion durch;
    erzeuge dabei eine Antwort;
    sende über s die Antwort an die IP-Adresse und Portnummer,
        von der die Anfrage kam;
}
```

## 5.2.2 Socket-Schnittstelle zu TCP

Wie bei UDP werden auch bei TCP Portnummern verwendet. TCP- und UDP-Portnummern sind unabhängig voneinander. Das bedeutet, dass dieselbe Portnummer sowohl für UDP als auch für TCP auf einem Rechner benutzt werden kann, ohne dass dies zu Problemen führt. An die TCP-Sockets können wie an die UDP-Sockets Portnummern gebunden werden. In einer *Client-Server-Anwendung* bindet der Server eine *wohlbekannte Portnummer* an seinen TCP-Socket, während der Client eine beliebige Portnummer verwenden kann. Der Client baut aktiv über seinen Socket eine Verbindung zu einem Server auf und kann erst nach erfolgreichem Verbindungsauflauf Daten über diesen Socket senden und empfangen. Der Server wartet an seinem Socket, bis ein Client eine Verbindung mit ihm aufbaut. Eine Besonderheit der Socket-Schnittstelle ist nun, dass in diesem Fall nach Annahme einer Verbindung ein neuer Socket entsteht. Der alte Socket kann für weitere Verbindungsannahmen benutzt werden, während der neue Socket zum Senden und Empfangen von Daten für diese neue Verbindung dient. Beachten Sie bitte, dass auf dem Server für jede passiv angenommene Verbindung ein neuer Socket erzeugt wird, während beim aktiven Verbindungsauflauf auf Client-Seite kein neuer Socket entsteht.

Ein Client könnte demnach in Pseudocode wie folgt programmiert werden:

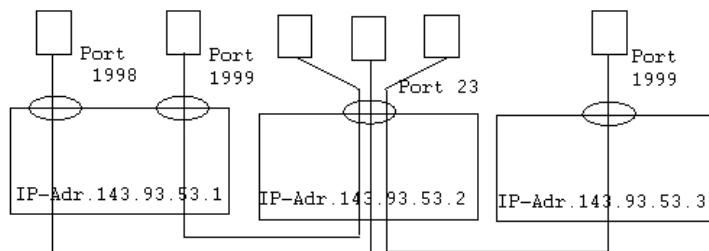
```
erzeuge einen TCP-Socket s;
/* dabei besitzt der Socket irgendeine, im Moment noch
   nicht benutzte Portnummer */
baue über s eine Verbindung zu einer vorgegebenen IP-Adresse
   und Portnummer auf;
/* dabei wird der Client so lange blockiert, bis die
   Gegenseite die Verbindung angenommen hat */
wiederhole so oft wie nötig:
{
    sende über s eine Anfrage;
    /* dabei ist die Angabe von IP-Adresse und Portnummer
       nicht möglich und auch nicht nötig
    */
    warte auf eine Antwort an s;
    analysiere die Antwort;
}
schließe die Verbindung über s;
```

Ein passendes Server-Programm dazu kann folgenden Aufbau haben:

```
erzeuge einen TCP-Socket sAnnahme mit einer spezifischen Portnummer;
wiederhole immer wieder:
{
    warte auf einen Verbindungsauflauf an sAnnahme
        und nimm die Verbindung an,
        dabei wird ein neuer Socket sVerb erzeugt;
    wiederhole, so lange die Verbindung besteht:
    {
        warte auf eine Anfrage an sVerb;
        falls eine Anfrage angekommen ist
        {
            analysiere die Anfrage;
            führe entsprechend der Anfrage eine Aktion durch;
            erzeuge dabei eine Antwort;
        }
    }
}
```

```
sende über sVerb die Antwort;
/* dabei ist die Angabe von IP-Adresse und Portnummer
   nicht möglich und auch nicht nötig
*/
}
andernfalls /* Verbindung wurde vom Client geschlossen */
{
    schließe die Verbindung über sVerb;
    verlasse die innere Schleife;
    /* d.h. es geht weiter bei
       "warte auf einen Verbindungsaufbau ..." */
}
}
```

In obigem Beispielprogramm gibt es zu einem Zeitpunkt höchstens zwei gültige Sockets. Ein alternatives Serverprogramm könnte aber auch während des Bestehens einer Verbindung über den Socket sAnnahme weitere Verbindungen annehmen. In diesem Fall gäbe es einen Socket zur Verbindungsannahme und für jede bestehende Verbindung je einen weiteren Socket. Es stellt sich jetzt die Frage, welche Portnummern an die Sockets auf der Serverseite gebunden sind. Die vielleicht für Sie verblüffende Antwort lautet: An alle Sockets ist dieselbe Portnummer gebunden, nämlich die bei der Erzeugung des Sockets sAnnahme angegebene. In den TCP-Segmenten (den in IP-Paketen verpackten TCP-Dateneinheiten) befindet sich wie bei UDP bzgl. der Adressierung lediglich eine Ziel- und Quellportnummer. Daraus ergibt sich das Problem, wie die ankommenden TCP-Daten auf die Sockets verteilt werden, da die TCP-Segmente ja offenbar alle dieselbe Portnummer beinhalten. Zum einen sind die Verbindungsaufbau-Segmente speziell gekennzeichnet, so dass sich diese von den reinen Daten-Segmenten unterscheiden lassen und somit eine Zuordnung zum sAnnahme-Socket oder einer der sVerb-Sockets vorgenommen werden kann. Ein Daten-Segment muss einer bestehenden Verbindung zugeordnet werden, damit es an den richtigen Socket verteilt werden kann. Bei TCP hat man sich dafür entschieden, keine Verbindungskennung in den TCP-Segmenten mitzuführen, sondern eine TCP-Verbindung ist gekennzeichnet durch die IP-Adressen und Portnummern der beiden Kommunikationspartner.



**Bild 5.3** TCP-Verbindungen

Wie Bild 5.3 zeigt, kann es somit höchstens eine einzige TCP-Verbindung von einem Port auf einem Rechner zu einem anderen Port auf einem anderen Rechner geben (z.B. von Portnummer 1998 auf dem Rechner mit der IP-Adresse 143.93.53.1 zu Portnummer 23 auf dem Rechner mit der IP-Adresse 143.93.53.2). Dagegen können durchaus mehrere Verbindun-

gen bestehen von einem Port auf einem Rechner zu unterschiedlichen Rechnern mit eventuell denselben Portnummern (z.B. von Port 23 auf dem Rechner mit der IP-Adresse 143.93.53.2 einerseits zu Portnummer 1999 auf dem Rechner mit der IP-Adresse 143.93.53.1 und andererseits zu derselben Portnummer 1999 auf dem Rechner mit der IP-Adresse 143.93.53.3) oder zu unterschiedlichen Ports auf demselben Rechner (z.B. von Port 23 auf dem Rechner mit der IP-Adresse 143.93.53.2 einerseits zu Portnummer 1998 auf dem Rechner mit der IP-Adresse 143.93.53.1 und andererseits zu Portnummer 1999 auf demselben Rechner). Anhand dieser Ausführungen sehen Sie, dass die ab und zu anzutreffende Behauptung, dass ein Socket einem Port entspricht, bei näherer Betrachtung nicht korrekt ist.

### 5.2.3 Socket-Schnittstelle für Java

Die Java-Klassenbibliothek enthält u.a. auch eine „objektorientierte Verkleidung“ der ursprünglich rein prozeduralen Socket-Schnittstelle. Die wichtigsten Klassen der Java-Socket-Schnittstelle, die sich im Package `java.net` befinden, sind:

- Die Klasse `InetAddress` repräsentiert den Namen eines Rechners sowie die dazugehörige IP-Adresse.
- Die Klassen `DatagramPacket` und `DatagramSocket` werden benötigt zur Kommunikation über UDP.
- Die Klassen `Socket` und `ServerSocket` braucht man zum TCP-Verbindungsauf- und -abbau (zur Kommunikation werden Klassen aus dem Package `java.io` verwendet).

Die wichtigsten Methoden der Klasse `InetAddress` sind:

```
public class InetAddress
{
    ...
    public String getHostName() {...}
    public String getHostAddress() {...}
    public static InetAddress getByName(String host)
        throws UnknownHostException {...}
    public static InetAddress[] getAllByName(String host)
        throws UnknownHostException {...}
    public static InetAddress getLocalHost()
        throws UnknownHostException {...}
    public boolean isReachable(int timeout)
        throws IOException {...}
    public boolean isReachable(NetworkInterface netif, int ttl,
        int timeout)
        throws IOException {...}
    ...
}
```

Die Klasse `InetAddress` stellt einen Rechnernamen und die dazugehörige IP-Adresse dar. Mit den Methoden `getHostName` kann der Rechnername und mit der Methode `getHostAddress` die IP-Adresse gelesen werden. Objekte der Klasse `InetAddress` werden in der Regel nicht mit `new` erzeugt. Stattdessen kann eine der angegebenen Static-Methoden verwendet werden:

Die Methode `getByName` liefert zu einem Rechnernamen, der als Parameter angegeben wird, ein entsprechendes InetAddress-Objekt. Als Parameter kann auch die IP-Adresse als String angegeben werden. Folgendes Programmfragment gibt die IP-Adresse des Rechners mit dem Namen „www.hochschule-trier.de“ aus:

```
try
{
    InetAddress ia =
        InetAddress.getByName("www.hochschule-trier.de");
    System.out.println("Zu " + ia.getHostName()
        + " gehört die IP-Adresse "
        + ia.getHostAddress());
}
catch(UnknownHostException e)
{
    System.out.println("Ausnahme: " + e);
}
```

Wenn es zu einem Rechnernamen mehrere IP-Adressen gibt (z. B. bei Rechnern mit mehreren Netzanschlüssen), dann kann man durch Verwendung der Methode `getAllByName` alle IP-Adressen erfragen, die in Form eines InetAddress-Feldes zurückgegeben werden. Mit der Methode `getLocalHost` erhält man ein InetAddress-Objekt, das den Namen und IP-Adresse des eigenen Rechners repräsentiert.

Interessant sind noch die beiden Varianten der Methode `isReachable`. Damit kann man wie mit dem Kommando `ping` überprüfen, ob ein Rechner erreichbar ist oder nicht. Mit dem Parameter `timeout` gibt man an, wie lange auf eine Antwort höchstens gewartet werden soll. Allerdings sagt die so überprüfte Erreichbarkeit eines Rechners nicht unbedingt etwas darüber aus, ob der Rechner auch über UDP oder TCP mit einer bestimmten Portnummer erreichbar ist. Eine *Firewall*, die zwischen Ihrem Rechner und dem überprüften Rechner steht, könnte nämlich zum Beispiel die Ping-Nachrichten durchlassen und die UDP-Datagramme bzw. TCP-Segmente verwerfen (oder umgekehrt).

## ■ 5.3 Kommunikation über UDP mit Java-Sockets

Wie im vorigen Abschnitt erwähnt wurde, sind die Klassen `DatagramSocket` und `DatagramPacket` die wichtigen Klassen zur Kommunikation über UDP. An ein Objekt der Klasse `DatagramSocket` kann eine Portnummer gebunden werden. Über diese „Kommunikationssteckdose“ können Objekte der Klasse `DatagramPacket` versendet und empfangen werden. Ein `DatagramPacket`-Objekt enthält neben den Daten die IP-Adresse und Portnummer des Partners. Für zu sendende Pakete ist dies dann die Zieladresse und Zielportnummer, für empfangene Pakete ist es die Quelladresse und Quellportnummer. In alle über ein `DatagramSocket`-Objekt gesendeten Datagramme wird als Quellportnummer die Portnummer des `DatagramSocket`-Objekts eingetragen. Entsprechend können von diesem `DatagramSocket`-Objekt nur solche Datagramme empfangen werden, die an diesen Port des Sockets adressiert sind.

Die Klasse *DatagramPacket* repräsentiert die Daten in Form eines Feldes des Typs byte und einer Länge. Die Länge kann höchstens so groß wie die Feldlänge sein. Ist die Länge kleiner als die Feldlänge, so wird nur der vordere Teil des angegebenen Feldes bis zur festgelegten Länge als relevant betrachtet. Weitere Attribute eines DatagramPacket-Objekts sind die IP-Adresse bzw. der Rechnername in Form eines InetSocketAddress-Objekts sowie die Portnummer. Die Klasse DatagramPacket besitzt entsprechende Setter- und Getter-Methoden für diese Attribute:

```
public class DatagramPacket
{
    public DatagramPacket(byte[] buf, int length) {...}
    public DatagramPacket(byte[] buf, int length,
                          InetAddress address, int port) {...}
    public byte[] getData() {...}
    public int getLength() {...}
    public InetAddress getAddress() {...}
    public int getPort() {...}
    public void setData(byte[] buf) {...}
    public void setLength(int length) {...}
    public void setAddress(InetAddress address) {...}
    public void setPort(int port) {...}
}
```

Die wichtigsten Methoden der Klasse *DatagramSocket* sind:

```
public class DatagramSocket implements Closeable
{
    public DatagramSocket() throws SocketException {...}
    public DatagramSocket(int port) throws SocketException {...}
    public void send(DatagramPacket p) throws IOException {...}
    public void receive(DatagramPacket p) throws IOException {...}
    public void setSoTimeout(int timeout) throws SocketException
    {...}
    public int getSoTimeout() throws SocketException {...}
    public InetAddress getLocalAddress() {...}
    public int getLocalPort() {...}
    public void close() {...}
}
```

Bei dem Konstruktor mit dem Int-Argument wird die Portnummer, die an den neuen Socket gebunden werden soll, vorgegeben. Dies funktioniert allerdings nur dann, falls die angegebene Portnummer im Moment noch frei ist (d. h. keinem anderen UDP-Socket auf diesem Rechner bereits zugewiesen ist). Dieser Konstruktor wird in der Regel von Servern benutzt. Bei der Benutzung des parameterlosen Konstruktors wird eine im Moment nicht benutzte Portnummer (größer oder gleich 1024) an den Socket gebunden. Dieser Konstruktor wird in der Regel von Clients benutzt. Wie die Namen *send* und *receive* andeuten, wird mit Hilfe dieser Methoden ein DatagramPacket-Objekt gesendet bzw. empfangen. Die Methode *receive* ist wie z. B. die gleichnamige Methode in unserer selbst implementierten Message-Queue blockierend. Das heißt, wenn im Moment kein DatagramPacket-Objekt vorliegt, wird so lange gewartet, bis ein Paket eintrifft. Die Wartezeit kann durch einen dem *receive* vorausgehenden Aufruf von *setSoTimeout* befristet werden. Die aktuell eingestellte Frist kann mit *getSoTimeout* gelesen werden. Die Methoden *getLocalAddress* und *getLocalPort* liefern die lokale IP-Adresse bzw. Portnummer des Sockets zurück. Mit *close* wird der Socket geschlossen; im Anschluss daran ist keine Kommunikation mehr über diesen Socket möglich.

Bei der erstmaligen Beschäftigung mit diesem Themenbereich wird man vermutlich der zuletzt genannten Methode zum Schließen des DatagramSockets keine besondere Beachtung schenken, insbesondere dann, wenn man nur kleinere Programme schreibt, in denen der erzeugte DatagramSocket ohnehin bis zum Ende des Programms benötigt wird; wird nämlich am Ende des Programms der DatagramSocket nicht geschlossen, so wird dies automatisch beim Beseitigen des Prozesses durch das Betriebssystem erledigt. Insofern sieht es so aus, als könnte man auf das Schließen verzichten. Wenn man diesen Programmteil dann aber später in ein größeres Programm einbaut, das längere Zeit läuft und in dem vielleicht sogar in einer Schleife wiederholt DatagramSocket-Objekte erzeugt werden, dann kann sich das fehlende Schließen nachteilig auswirken. Beim Erzeugen eines DatagramSocket-Objekts werden nämlich auch Funktionen des Betriebssystems aufgerufen, mit denen gewisse Ressourcen des Betriebssystems belegt werden (u. a. die Portnummern). Werden nicht mehr benötigte DatagramSockets nicht geschlossen, dann kann es auf Dauer zu einem Ressourcenengpass kommen. Es ist deshalb nicht nur guter Stil, sondern wichtig und nützlich, diese Sockets nach Gebrauch wieder zu schließen. Diese Bemerkung bezieht sich nicht nur auf die DatagramSockets im Zusammenhang mit UDP, sondern auch auf Sockets für TCP und allgemeiner auf alle Ein- und Ausgabeströme. Seit der Version 7 von Java gibt es eine spezielle Sprachunterstützung unter dem Namen *try-with-resources*, die für das automatische Schließen spezieller Ressourcen sorgt. Genaueres dazu folgt später.

Es mag auf den ersten Blick überraschend sein, dass die Klasse DatagramSocket auch eine Connect-Methode besitzt, obwohl UDP doch verbindungslos ist. Mit der Methode *connect* wird aber keine Verbindung im TCP-Sinn hergestellt, sondern der Aufruf dieser Methode bewirkt lediglich, dass ab sofort nur noch Datagramme an den im connect-Aufruf als Argument angegebenen Partner geschickt und von diesem Partner empfangen werden können. Wir werden diese Methode in diesem Buch nicht benutzen.

Als Daten können beliebige Byte-Folgen geschickt werden. Dies kann z. B. die Binärdarstellung einer Zahl des Typs int, float oder double, die Binärdarstellung eines Java-Objekts, eine Zeichenkette usw. oder eine beliebige Kombination davon sein. In den Beispielen dieses Buchs werden wir nur Zeichenketten (Strings) schicken. Die immer wiederkehrende Aufgabe, einen String vor dem Senden in ein Byte-Feld und nach dem Empfangen das Byte-Feld wieder in einen String zurück zu wandeln, kapseln wir in Methoden. Wir schreiben dazu die Klasse UDPSocket (s. Listing 5.1), die einen DatagramSocket benutzt. Mit Hilfe dieser Klasse können nur Strings gesendet und empfangen werden, aber in einfacherer Weise, als dies ohne diese Klasse der Fall wäre. Ferner merkt sich ein Objekt der Klasse UDPSocket die IP-Adresse und Portnummer des zuletzt empfangenen DatagramPackets. Mit der Methode *reply* kann dann ohne Angabe eines Empfängers demjenigen geantwortet werden, von dem zuletzt eine Nachricht empfangen wurde, indem die gemerkten Angaben verwendet werden. Diese gemerkten Angaben können durch entsprechende Get-Methoden auch erfragt werden. In der Send- und Receive-Methode wird die Wandlung von einem String in ein Feld des Typs byte bzw. umgekehrt durchgeführt. Die Begründung für die Protected-Sichtbarkeit des Socket-Attributs sowie für den Protected-Konstruktor wird erst im Zusammenhang mit Multicast (s. Abschnitt 5.4) deutlich werden.

**Listing 5.1**

```
import java.io.*;
import java.net.*;

public class UDPSocket implements AutoCloseable
{
    protected DatagramSocket socket;
    private InetAddress address;
    private int port;

    public UDPSocket() throws SocketException
    {
        this(new DatagramSocket());
    }

    public UDPSocket(int port) throws SocketException
    {
        this(new DatagramSocket(port));
    }

    protected UDPSocket(DatagramSocket socket)
    {
        this.socket = socket;
    }

    public void send(String s, InetAddress rcvrAddress,
                     int rcvrPort)
        throws IOException
    {
        byte[] outBuffer = s.getBytes();
        DatagramPacket outPacket = new DatagramPacket(outBuffer,
                                                       outBuffer.length,
                                                       rcvrAddress,
                                                       rcvrPort);
        socket.send(outPacket);
    }

    public String receive(int maxBytes) throws IOException
    {
        byte[] inBuffer = new byte[maxBytes];
        DatagramPacket inPacket = new DatagramPacket(inBuffer,
                                                      inBuffer.length);
        socket.receive(inPacket);
        address = inPacket.getAddress(); // addr for reply packet
        port = inPacket.getPort(); // port for reply packet

        return new String(inBuffer, 0, inPacket.getLength());
    }

    public void reply(String s) throws IOException
    {
        if(address == null)
        {
            throw new IOException("no one to reply");
        }
        send(s, address, port);
    }
}
```

```

public InetAddress getSenderAddress()
{
    return address;
}

public int getSenderPort()
{
    return port;
}

public void setTimeout(int timeout) throws SocketException
{
    socket.setSoTimeout(timeout);
}

public void close()
{
    socket.close();
}
}

```

Vielleicht haben die Leserinnen und Leser bemerkt, dass die Klasse `UDPSocket` die Schnittstelle `AutoCloseable` (aus dem Package `java.lang`) implementiert (`Closeable` ist eine aus `AutoCloseable` abgeleitete Schnittstelle, die von `DatagramSocket` implementiert wird, s. oben). `AutoCloseable` hat eine einzige parameterlose Void-Methode namens `close`. Wenn eine Klasse diese Schnittstelle implementiert, dann können deren Objekte in einer *Try-with-resources-Anweisung* verwendet werden. Zur Illustration betrachten wir die Klasse X:

```

class X implements AutoCloseable
{
    public X() {}
    public void m() {}
    public void close() {}
}

```

Da `AutoCloseable` von X implementiert wird, kann für X-Objekte das Try-with-resources benutzt werden:

```

try(X x = new X())
{
    x.m();
    ...
}

```

Der entscheidende Effekt dieser Try-with-resources-Anweisung ist, dass beim Verlassen des Try-Blocks (gleichgültig, ob regulär, durch eine Ausnahme, durch `return` oder auf andere Art) in jedem Fall noch die Methode `close` auf das X-Objekt angewendet wird. Wie Sie sehen, ist im Gegensatz zu einem normalen Try bei einem Try-with-resources nicht unbedingt ein Catch- oder Finally-Block notwendig. Wenn allerdings der Konstruktor der Klasse X, die Methode `m` oder `close` so deklariert wäre, dass sie eine Ausnahme werfen könnten (`throws` für eine Checked Exception), dann wäre auch in der obigen Try-with-resources-Anweisung ein Catch-Block notwendig. Angenommen, die Methode `m` würde tatsächlich eine solche Ausnahme werfen. Dann wird erst der Try-Block beendet, dabei wird `close` aufgerufen (was

selbst wieder eine Ausnahme auslösen könnte). Im Anschluss daran wird die von m geworfene Ausnahme behandelt.

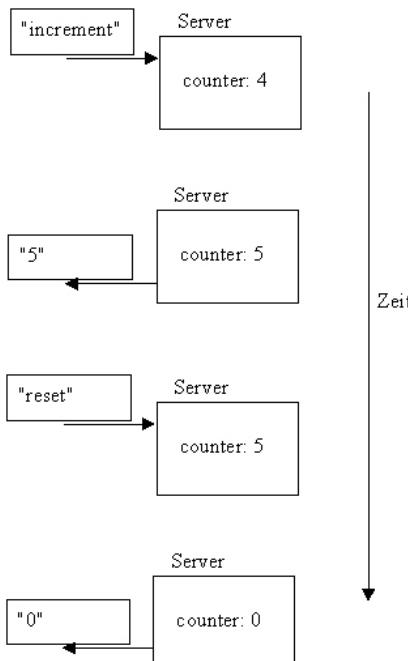
Selbstverständlich kann im Try-with-resources auch mehr als ein Objekt angelegt werden. Angenommen, die Klasse Y würde ebenfalls AutoCloseable implementieren, dann könnte das Try-with-resources beispielsweise auch so aussehen:

```
try(X x1 = new X(); X x2 = new X(); Y y = new Y())
{
    ...
}
```

Natürlich wird dann auf alle angegebenen Objekte x1, x2 und y beim Verlassen des Try-Blocks die Methode close angewendet.

Wenn übrigens AutoCloseable-Objekte weder mit Try-with-resources verwendet werden noch close auf sie angewendet wird, gibt der Compiler ab Java 7 eine Warnung aus.

Nach diesem kurzen Ausflug in das Sprachkonstrukt Try-with-resources kehren wir zu unserer Klasse UDPSocket zurück. Mit dieser Klasse wird es nun einfacher, eine Client-Server-Anwendung, die Strings über UDP versendet und empfängt, zu programmieren. Als Beispieldemonstration realisieren wir einen Server, der einen Zähler enthält. Wird dem Server ein Datagramm mit dem String „increment“ geschickt, so erhöht der Server den Zähler um eins und sendet den neuen Wert des Zählers als String zurück. Wird dagegen ein Datagramm mit „reset“ geschickt, so wird der Zähler auf 0 zurückgesetzt und dieser neue Zählerstand als Antwort dem Sender zurückgeschickt (s. Bild 5.4). Erhält der Server irgend etwas anderes, so bleibt der Zähler unverändert; als Antwort wird dennoch der aktuelle Zählerstand geschickt (d. h. jede Nachricht verschieden von „increment“ und „reset“ wirkt wie ein „get“). Damit haben wir unser eigenes Anwendungsprotokoll (Protokoll der Schicht 5) definiert.



**Bild 5.4**

Verhaltensweise des Beispiel-UDP-Servers

Das Java-Programm des Servers ist in Listing 5.2 dargestellt (vergleichen Sie dazu den Pseudocode aus Abschnitt 5.2.1):

### Listing 5.2

```
public class Server
{
    public static void main(String[] args)
    {
        int counter = 0;

        // create socket
        try(UDPSocket udpSocket = new UDPSocket(1250))
        {
            // wait for request packets
            System.out.println("Server wartet auf Kunden");

            // execute client requests
            while(true)
            {
                // receive request
                String request = udpSocket.receive(20);

                // perform increment operation
                if(request.equals("increment"))
                {
                    // perform increment
                    counter++;
                }
                else if(request.equals("reset"))
                {
                    // perform reset
                    counter = 0;
                    System.out.println("Zähler auf 0 gesetzt durch "
                        + udpSocket.getSenderAddress()
                        + ":"
                        + udpSocket.getSenderPort());
                }

                // generate answer
                String answer = String.valueOf(counter);

                // send answer
                udpSocket.reply(answer);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("DatagramSocket wurde geschlossen");
    }
}
```

Der Beispiel-Client sendet zunächst das Kommando „reset“ und dann mehrmals das Kommando „increment“. Der Client muss mit zwei Kommandozeilen-Argumenten gestartet werden: das erste Argument ist der Name des Rechners, auf dem der Server läuft, das zweite

Argument ist eine positive Zahl, die angibt, wie oft der Client das Kommando „increment“ senden soll. Auf zwei Besonderheiten des UDP-Clients, die im Pseudocode des Clients oben nicht enthalten sind, sei besonders hingewiesen:

- Die Zeit, wie lange das wiederholte Senden des Kommandos „increment“ sowie das Empfangen der entsprechenden Antwort dauert, wird gemessen. Dazu wird die schon bekannte Methode System.currentTimeMillis verwendet. Diese Zeit wird am Ende ausgegeben zusammen mit der Zeit, wie lange ein Schleifendurchlauf im Durchschnitt gedauert hat. Diese Durchschnittszeit wird bestimmt, indem die Gesamtzeit durch die Anzahl der Schleifendurchläufe dividiert wird.
- UDP ist unzuverlässig. Sollte das Kommando des Clients oder die Antwort des Servers verloren gehen, so würde in beiden Fällen keine Antwort beim Client eingehen. Der Client würde in diesem Fall „hängen bleiben“, falls er genauso programmiert wäre wie im Pseudocode. Das heißt: Falls keine Antwort kommt, würde das nächste Kommando nie mehr gesendet werden und der Client nicht zu Ende laufen. Als einfache Gegenmaßnahme wird im folgenden Programm eine Frist gesetzt, wie lange höchstens auf eine Antwort gewartet wird. Läuft diese Frist ab, so wird eine Ausnahme geworfen. Unser Client reagiert darauf so, dass er lediglich eine Fehlermeldung ausgibt. Der eigentliche Effekt ist aber, dass der Client aus dem Wartezustand geworfen wird und somit nicht endlos wartet. Dieses Vorgehen verhindert aber lediglich das Hängenbleiben. Wir können damit nicht garantieren, dass der Zähler des Servers genauso oft erhöht wird, wie es als Kommandozeilen-Argument angegeben wurde. Diese pragmatische Vorgehensweise hält das Programm aber überschaubar.

In Listing 5.3 ist nun das schon diskutierte Client-Programm dargestellt (vergleichen Sie dazu den Pseudocode aus Abschnitt 5.2.1):

#### **Listing 5.3**

```
import java.net.*;

public class Client
{
    private static final int TIMEOUT = 10000; // 10 seconds

    public static void main(String args[])
    {
        if(args.length != 2)
        {
            System.out.println("Notwendige Kommandozeilenargumente:"
                + " <Name des Server-Rechners>"
                + " <Anzahl>");
            return;
        }

        // create datagram socket
        try(UDPSocket udpSocket = new UDPSocket())
        {
            udpSocket.settimeout(TIMEOUT);

            // get inet addr of server
            InetAddress serverAddr = InetAddress.getByName(args[0]);
        }
    }
}
```

```
// set counter to zero
System.out.println("Zähler wird auf 0 gesetzt.");
udpSocket.send("reset", serverAddr, 1250);

String reply = null;
// receive reply
try
{
    reply = udpSocket.receive(20);
    System.out.println("Zähler: " + reply);
}
catch(Exception e)
{
    System.out.println(e);
}

// get count, initialize start time
System.out.println("Nun wird der Zähler erhöht.");
int count = new Integer(args[1]).intValue();
long startTime = System.currentTimeMillis();

// perform increment "count" number of times
for(int i = 0; i < count; i++)
{
    udpSocket.send("increment", serverAddr, 1250);
    try
    {
        reply = udpSocket.receive(20);
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}

// display statistics
long stopTime = System.currentTimeMillis();
long duration = stopTime - startTime;
System.out.println("Gesamtzeit = " + duration
                  + " msecs");
if(count > 0)
{
    System.out.println("Durchschnittszeit = "
                      + ((duration) / (float) count)
                      + " msecs");
}
System.out.println("Letzter Zählerstand: " + reply);
}
catch(Exception e)
{
    System.out.println(e);
}
System.out.println("DatagramSocket wurde geschlossen");
}
```

Wenn zum ersten Ausprobieren sowohl das Server- als auch das Client-Programm auf demselben Rechner ausgeführt wird, dann kann der Client z.B. mit den beiden Kommandozeilen-Argumenten localhost und 1000 gestartet werden.

## ■ 5.4 Multicast-Kommunikation mit Java-Sockets

IP-Multicast-Adressen sind spezielle Adressen, mit denen mehr als ein Rechner angesprochen werden kann. Wenn wir Multicast verwenden wollen, so müssen wir auf UDP zurückgreifen. TCP ist im Gegensatz zu UDP ein Protokoll, das nur für genau zwei Partner gedacht ist. Das Senden einer Multicast-Nachricht ist mit den bereits behandelten Klassen DatagramSocket und DatagramPacket möglich. Als Zieladresse ist einfach eine Multicast-IP-Adresse anzugeben. Zum Empfangen einer Multicast-Nachricht benötigt man dagegen ein Objekt der Klasse *MulticastSocket*. Diese Klasse ist aus DatagramSocket abgeleitet (d.h. alle in DatagramSocket vorhandenen Methoden wie send und receive werden auf MulticastSocket vererbt). Die wichtigsten zusätzlichen Methoden sind solche, mit denen man einer Multicast-Gruppe beitreten (*joinGroup*) und diese Gruppe wieder verlassen kann (*leaveGroup*). Beim Beitreten zu einer Multicast-Gruppe wird der eigene Rechner für die Multicast-IP-Adresse aktiviert, welche die Multicast-Gruppe repräsentiert. Damit werden alle IP-Pakete, die an diese Multicast-IP-Adresse adressiert sind, von diesem Rechner empfangen, was ohne diese Aktivierung nicht geschehen würde. Hier sind die wichtigsten Methoden der aus DatagramSocket abgeleiteten Klasse MulticastSocket dargestellt:

```
public class MulticastSocket extends DatagramSocket
{
    public MulticastSocket() throws IOException {...}
    public MulticastSocket(int port) throws IOException {...}
    public void joinGroup(InetAddress mcastaddr) throws IOException
    {...}
    public void leaveGroup(InetAddress mcastaddr) throws IOException
    {...}
    public int getTimeToLive() throws IOException {...}
    public void setTimeToLive(int ttl) throws IOException {...}
}
```

Der Wert *Time-To-Live* kann in ein IP-Paket eingetragen werden. Das Paket wird damit über höchstens so viele Router weitergeleitet, wie dieser Wert angibt. Da dieselbe Multicast-Adresse unter Umständen von mehreren voneinander unabhängigen Anwendungen gleichzeitig benutzt werden kann, und da häufig die über Multicast kommunizierenden Rechner nicht allzu weit voneinander entfernt sind, kann durch Angabe eines kleineren Time-To-Live-Werts die Reichweite der abgesendeten IP-Pakete eingeschränkt werden, um keine anderen Anwendungen zu stören. Außerdem wird dadurch das Netz nicht so sehr belastet. Will man also einen speziellen Time-To-Live-Wert vorgeben, so muss man auch zum Senden einen Multicast-Socket benutzen. Da dies im folgenden Beispiel nicht der Fall ist, benötigen wir den Multicast-Socket jedoch nur auf der Seite, auf der Multicast-Nachrichten empfangen werden.

Zum einfacheren Senden und Empfangen von Strings über einen MulticastSocket wird im Folgenden die Klasse UDPMulticastSocket vorgestellt (Listing 5.4), die die Klasse UDP-Socket des vorigen Abschnitts erweitert. Statt eines DatagramSockets wird ein Multicast-Socket benutzt, und die Klasse wird um die für Multicast spezifischen Funktionen erweitert:

#### **Listing 5.4**

```
import java.io.*;
import java.net.*;

public class UDPMulticastSocket extends UDPsocket
{
    public UDPMulticastSocket(int port) throws IOException
    {
        super(new MulticastSocket(port));
    }

    public void join(String mcAddress) throws IOException
    {
        InetAddress group = InetAddress.getByName(mcAddress);
        ((MulticastSocket) socket).joinGroup(group);
    }

    public void leave(String mcAddress) throws IOException
    {
        InetAddress group = InetAddress.getByName(mcAddress);
        ((MulticastSocket) socket).leaveGroup(group);
    }
}
```

Der Konstruktor der Klasse UDPMulticastSocket verwendet den Protected-Konstruktor der Basisklasse UDPsocket, der einen DatagramSocket als Parameter hat. Da MulticastSocket aus DatagramSocket abgeleitet ist, kann diesem Konstruktor der Klasse UDPsocket auch ein MulticastSocket-Objekt als Parameter übergeben werden. Dadurch wird dem Attribut der Klasse UDPsocket, das vom Typ DatagramSocket ist, eine Referenz auf ein Multicast-Socket-Objekt zugewiesen. Dieses Attribut wird in den Methoden join und leave verwendet. Dies ist möglich, da das DatagramSocket-Attribut vorsorglich mit Protected-Sichtbarkeit ausgestattet wurde. Allerdings muss es auf MulticastSocket gecastet werden, damit die Methoden joinGroup und leaveGroup darauf anwendbar sind. Damit können wir nun einen einfachen Multicast-Echo-Server programmieren (Listing 5.5). Der Server schließt sich zu Beginn der Multicast-Gruppe an, deren Multicast-IP-Adresse als einziges Kommandozeilen-Argument beim Starten angegeben wird. Der Server sendet jede empfangene Nachricht wie ein Echo an den Sender zurück. Beim Empfang der Nachricht „exit“ verlässt er die Multicast-Gruppe, der er anfangs beigetreten ist, und beendet sich.

#### **Listing 5.5**

```
public class Server
{
    public static void main(String[] args)
    {
        if(args.length != 1)
        {
            System.out.println("Notwendiges Kommandozeilenargument:"
```

```

        + " <Multicast-IP-Adresse>");

    return;
}

try(UDPMulticastSocket multiSocket =
            new UDPMulticastSocket(1250))
{
    System.out.println("MulticastSocket erzeugt");
    multiSocket.join(args[0]);
    System.out.println("Multicast-Gruppe beigetreten");

    while(true)
    {
        String request = multiSocket.receive(200);
        System.out.println("Nachricht erhalten: "
                + multiSocket.getSenderAddress()
                + ":"
                + multiSocket.getSenderPort()
                + ":" +
                + request);
        multiSocket.reply(request);
        if(request.equals("exit"))
        {
            break;
        }
    }
    multiSocket.leave(args[0]);
    System.out.println("Multicast-Gruppe verlassen");
}
catch(Exception e)
{
    System.out.println("Ausnahme '" + e + "'");
}
System.out.println("MulticastSocket wurde geschlossen");
}
}

```

Der dazu passende Client soll wie folgt arbeiten: Er soll mit mehreren Kommandozeilen-Argumenten aufgerufen werden. Das erste Argument gibt dabei die Multicast-IP-Adresse an, an die gesendet werden soll. Die weiteren Argumente in beliebiger Anzahl sind die Strings, die der Reihe nach an die Multicast-Gruppe gesendet werden. Der Client soll ferner zu jeder gesendeten Nachricht alle Antworten anzeigen. Dabei ergibt sich das Problem, dass man nicht weiß, wie viele Antworten kommen, und dass sich diese Zahl auch ändern kann. Dies hat folgende Gründe:

- Zum einen wird UDP benutzt. Wie im vorigen Abschnitt wiederholt wurde, ist UDP unzuverlässig. Deshalb können die Nachricht des Clients oder die Antworten der Server verloren gehen.
  - Da man nicht an eine Liste von Unicast-Adressen, sondern an eine Multicast-Adresse sendet, weiß man nicht, wie viele Mitglieder diese Gruppe im Moment hat. Ferner können zu jeder Zeit neue Mitglieder der Gruppe beitreten oder die Gruppe verlassen.

Als Lösung wird wiederum wie im vorigen Abschnitt eine Frist (Timeout) verwendet (s. Listing 5.6). Der Client geht davon aus, dass er alle Nachrichten empfangen hat, wenn eine gewisse Zeit lang (im Beispiel sind es 2 Sekunden) keine Antwort mehr eintrifft. Programm-

technisch wird dies so umgesetzt, dass die Antworten in einer Endlosschleife (while(true)) entgegengenommen werden. Beim Ablaufen der Frist wird von der Receive-Methode eine Ausnahme geworfen. Der entsprechende Try-Catch-Block wird außerhalb der Schleife eingerichtet, so dass beim Werfen der Ausnahme die Schleife automatisch beendet wird.

**Listing 5.6**

```
import java.net.*;  
  
public class Client  
{  
    private static final int TIMEOUT = 2000; // 2 seconds  
  
    public static void main(String[] args)  
    {  
        if(args.length < 2)  
        {  
            System.out.println("Notwendige Kommandozeilenargumente:"  
                + " <Multicast-IP-Adresse>"  
                + " <Nachricht 1> ..." + " < Nachricht N>");  
            return;  
        }  
  
        // create datagram socket  
        try(UDPSocket udpSocket = new UDPSocket())  
        {  
            udpSocket.settimeout(TIMEOUT);  
  
            // get inet addr of server  
            InetAddress serverAddr = InetAddress.getByName(args[0]);  
  
            for(int i = 1; i < args.length; i++)  
            {  
                udpSocket.send(args[i], serverAddr, 1250);  
  
                try  
                {  
                    while(true)  
                    {  
                        String reply = udpSocket.receive(200);  
                        System.out.println("Nachricht erhalten: "  
                            + udpSocket.getSenderAddress()  
                            + ":"  
                            + udpSocket.getSenderPort()  
                            + ":"  
                            + reply);  
                    }  
                }  
                catch(Exception e)  
                {  
                    System.out.println("Ausnahme '" + e + "'");  
                }  
            } // for  
        } catch(Exception e)  
        {
```

```

        System.out.println("Ausnahme '" + e + "'");
    }
    System.out.println("DatagramSocket wurde geschlossen");
}
}

```

Beim Ausführen des vorgestellten Multicast-Beispiels ist Folgendes zu beachten:

- Im Gegensatz zum vorigen UDP-Beispiel können Sie mehrere Server auf einem einzigen Rechner gleichzeitig starten. Sie können dann auch Ihren Client auf demselben Rechner starten und können somit ein bisschen Multicast-Gefühl auch auf einem einzigen Rechner erleben.
- Sollten Sie mehrere Rechner benutzen, so sollte es keine Probleme geben, falls sich alle Rechner im selben lokalen Netz (d. h. im selben IP-Subnetz) befinden. Befinden sich Client und Server aber in unterschiedlichen Netzen, ist eine Kommunikation nur dann möglich, falls der oder die dazwischen liegenden Router multicast-fähig und entsprechend konfiguriert sind. Dies dürfte in der Mehrzahl der Fälle eher nicht der Fall sein.

## ■ 5.5 Kommunikation über TCP mit Java-Sockets

Zur Kommunikation über TCP mit Java-Sockets werden die Klassen *Socket* und *ServerSocket* eingesetzt. Diese Klassen dienen lediglich zum Auf- und Abbau von Verbindungen, nicht aber zum Senden und Empfangen der Daten. Zu diesem Zweck werden Klassen aus dem Ein-/Ausgabe-Package `java.io` herangezogen (s. u.).

Die Klasse *Socket* wird vom Client und vom Server verwendet, während die Klasse *ServerSocket* nur auf Server-Seite eingesetzt wird. Ein Objekt der Klasse *Socket* repräsentiert eine TCP-Verbindung. Der Client erzeugt ein solches *Socket*-Objekt explizit (mit `new`). Als Parameter für den Konstruktor wird der Server in Form von Rechnername oder IP-Adresse sowie Portnummer angegeben. Beim expliziten Erzeugen eines *Socket*-Objekts wird eine TCP-Verbindung aufgebaut, so dass ein zusätzliches Kommando zum Aufbauen der Verbindung nicht benötigt wird (hier haben wir also eine kleine Vereinfachung gegenüber dem Pseudocode aus Abschnitt 5.2.2).

Auf dem Server wird ein Objekt der Klasse *ServerSocket* explizit erzeugt. Als Parameter ist die Portnummer, an dem der Server lauscht, anzugeben. Diese ist eine so genannte wohl bekannte Portnummer, die den Clients bekannt sein muss. Die wichtigste Methode der Klasse *ServerSocket* ist *accept*. Damit wartet ein Server so lange, bis ein Client eine Verbindung zu ihm aufbaut. Der Rückgabetyp dieser Methode ist *Socket*. Das heißt, dass der Server von der Methode *accept* ein *Socket*-Objekt zurückbekommt, welches die neu aufgebaute Verbindung repräsentiert. Der Server erzeugt *Socket*-Objekte also nicht explizit mit `new` wie der Client, sondern lässt sich diese über die Methode *accept* generieren. Die Funktion der weiteren Methoden *close*, *getInetAddress* und *getLocalPort* sind an ihren Namen jeweils ablesbar:

```
public class ServerSocket implements Closeable
{
    public ServerSocket(int port) throws IOException {...}
    public Socket accept() throws IOException {...}
    public void close() throws IOException {...}
    public InetAddress getInetAddress() {...}
    public int getLocalPort() {...}
}
```

Die wichtigsten Methoden der Klasse *Socket* sind:

```
public class Socket implements Closeable
{
    public Socket(String host, int port)
        throws UnknownHostException, IOException;
    public Socket(InetAddress address, int port)
        throws IOException;
    public void close() throws IOException;
    public void shutdownInput() throws IOException;
    public void shutdownOutput() throws IOException;
    public InetAddress getInetAddress();
    public int getPort();
    public InetAddress getLocalAddress();
    public int getLocalPort();
    public InputStream getInputStream() throws IOException;
    public OutputStream getOutputStream() throws IOException;
}
```

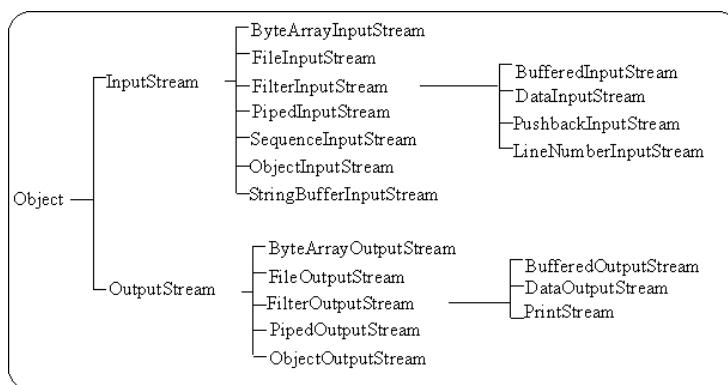
Beim Erzeugen eines *Socket*-Objekts wird unmittelbar eine TCP-Verbindung aufgebaut. Der Partner, zu dem die Verbindung aufgebaut wird, kann auf unterschiedliche Arten angegeben werden. Oben wurden zwei Konstruktoren angegeben: Bei dem einen gibt man den Namen des Rechners oder dessen IP-Adresse als String an, bei dem anderen als *InetAddress*. In beiden Fällen wird als zweiter Parameter die Portnummer des Partners als Int-Wert angegeben. Das Abbauen der Verbindung kann auf mehrere Arten erfolgen: Falls mit *close* die Verbindung geschlossen wird, kann man danach weder senden noch empfangen. Da eine TCP-Verbindung aber als zwei unidirektionale Verbindungen gesehen werden kann, kann jede Richtung auch separat geschlossen werden: Mit *shutdownInput* schließt man die Eingangsverbindung; folglich kann man nach dem Aufruf dieser Methode noch weiter senden, aber nicht mehr empfangen. Das Umgekehrte gilt für *shutdownOutput*. Mit den angegebenen Getter-Methoden lassen sich sowohl die eigene Adresse und Portnummer als auch diejenigen des Partners erfragen.

Wie Sie sehen, gehören zu den wichtigsten Methoden der Klasse *Socket* keine Methoden zum Schreiben und Lesen bzw. Senden und Empfangen von Daten. Stattdessen existieren die beiden Methoden *getInputStream* und *getOutputStream*, mit denen man sich einen Eingabe- bzw. Ausgabestrom (*InputStream/OutputStream*) geben lassen kann. Von einem Eingabestrom kann (durch mehrere überladene Methoden *read*) gelesen und auf einen Ausgabestrom kann (durch mehrere überladene Methoden *write*) geschrieben werden. Mit dem Schreiben auf dem Ausgabestrom werden Daten über die entsprechende TCP-Verbindung geschickt, die durch das *Socket*-Objekt repräsentiert wird. Durch das Lesen des Eingabestroms können Daten von der TCP-Verbindung empfangen werden. Da wir aber in unseren Beispielen nie direkt auf diesen vom *Socket* zurückgelieferten Strömen lesen und schreiben, müssen wir einen kurzen Einblick in das Package *java.io* geben, zu dem auch die Klassen *InputStream* und *OutputStream* gehören.

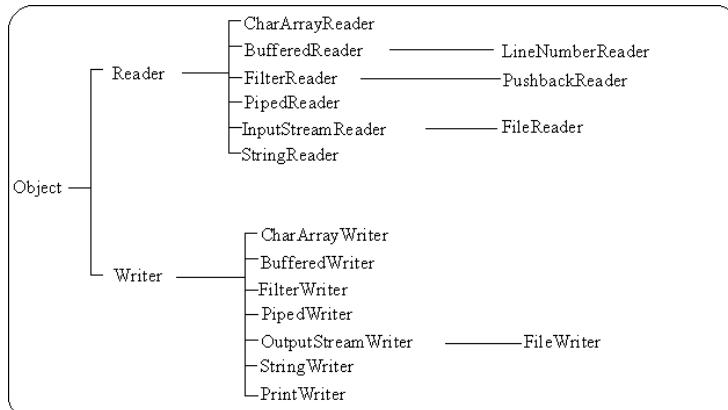
In Java gibt es unterschiedliche Klassen für die Eingabe und die Ausgabe einerseits und unterschiedliche Klassen für die byteweise und zeichenweise Ein-/Ausgabe andererseits (ein Zeichen [character] ist in Java bekanntlich zwei Bytes lang). Dies ergibt also vier Kombinationsmöglichkeiten. Zu jeder Kombinationsmöglichkeit gibt es eine entsprechende Java-Klasse:

- *InputStream*: byteweise Eingabe
- *OutputStream*: byteweise Ausgabe
- *Reader*: zeichenweise Eingabe
- *Writer*: zeichenweise Ausgabe

Diese vier Java-Klassen sind allerdings alle abstrakt (zur Erinnerung: von abstrakten Klassen kann es keine Objekte geben, sondern nur von daraus abgeleiteten nicht abstrakten Klassen). Zu jeder dieser vier Klassen gibt es eine ganze Reihe von daraus abgeleiteten Klassen. In Bild 5.5 und Bild 5.6 ist die Vererbungshierarchie für die Klassen zur byteweisen bzw. zeichenweisen Ein-/Ausgabe dargestellt. Dabei stellt eine Verbindungslinie von links nach rechts eine Vererbungsbeziehung dar. So ist beispielsweise in Bild 5.5 zu sehen, dass die Klasse *BufferedInputStream* aus *FilterInputStream* abgeleitet ist, diese aus *InputStream* und *Object* selbst aus *Object*.



**Bild 5.5** Aus *InputStream* und *OutputStream* abgeleitete Klassen



**Bild 5.6** Aus *Reader* und *Writer* abgeleitete Klassen

Alle diese Klassen implementieren die Schnittstelle AutoCloseable und können deshalb in einem Try-with-resources verwendet werden.

Man mag sich nun fragen, warum es so viele Klassen gibt. Zum einen gibt es unterschiedliche Klassen für unterschiedliche Datensenken und Datenquellen. So gibt es z.B. Klassen für das Schreiben und Lesen von Dateien (*FileInputStream*, *FileOutputStream*, *FileReader*, *FileWriter*) und für das Schreiben und Lesen von Feldern (*ByteArrayInputStream*, *ByteArrayOutputStream*, *CharacterArrayReader*, *CharacterArrayWriter*). Daneben gibt es Klassen, die gewisse Zusatzfunktionalitäten erbringen.

Objekte von Klassen, welche Zusatzfunktionalitäten erbringen, können dabei wie unterschiedliche Verarbeitungseinheiten zu einer Art Fließband zusammengesteckt werden. Eine solche Klasse mit Zusatzfunktionalität aus dem InputStream-Bereich hat dazu in der Regel einen Konstruktor mit einem Parameter des Typs InputStream. Da InputStream abstrakt ist, kann es keine Objekte davon geben, aber jedes Objekt einer aus InputStream abgeleiteten Klasse kann somit als Parameter verwendet werden. Der Effekt ist nun der Folgende: Immer wenn von dem Objekt der Klasse mit der Zusatzfunktionalität gelesen wird, dann werden die Daten von dem Eingabestrom des Parameter-Objekts gelesen und entsprechend behandelt, bevor sie weitergegeben werden. Entsprechendes gilt für OutputStream, Reader und Writer.

Zum besseren Verständnis soll das Prinzip an einem Beispiel illustriert werden. Mit der Klasse *DataOutputStream* können Werte z.B. der Typen int, float oder double in Binärdarstellung geschrieben werden. Beim Erzeugen eines Objekts der Klasse DataOutputStream wird dieses mit einem anderen OutputStream-Objekt verknüpft, das als Parameter angegeben wird. Wenn nun mit der Methode *writeInt* ein Int-Wert über das DataOutputStream-Objekt ausgegeben werden soll, dann wird der Int-Wert von diesem Objekt in eine Bytefolge gewandelt und diese Bytefolge wird an den als Konstruktor-Parameter übergebenen Ausgabestrom ausgegeben. Ist dieser Ausgabestrom z.B. ein Dateiausgabestrom, so können damit Int-Werte in Binärdarstellung in eine Datei geschrieben werden. Das folgende Programmfragment zeigt diese Möglichkeit (der Einfachheit halber ohne Try-with-resources):

```
/* Dateiausgabestrom in die Datei "x.y";
falls die Datei noch nicht existiert, wird sie erzeugt;
falls sie schon existiert, werden alle bisherigen Inhalte
gelöscht */
FileOutputStream fout = new FileOutputStream("x.y");

/* Datenausgabestrom wird auf Dateiausgabestrom aufgesetzt */
DataOutputStream dout = new DataOutputStream(fout);

/* Int-Wert wird geschrieben */
dout.writeInt(4711);
```

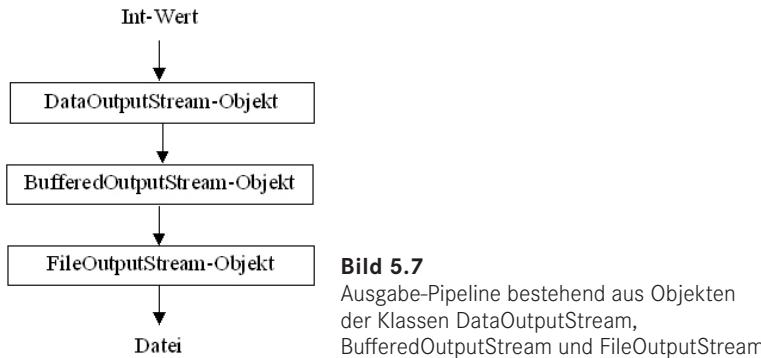
Entsprechend kann dieser in der Datei x.y gespeicherte Int-Wert wieder gelesen werden, entweder byteweise oder direkt als Int-Wert über *DataInputStream*.

Eine weitere wichtige Zusatzfunktion wird durch Pufferung erbracht (Klassen *BufferedInputStream*, *BufferedOutputStream*, *BufferedReader* und *BufferedWriter*). Bei der Ausgabe werden hier die zu schreibenden Daten zunächst in einem Feld (Puffer) abgelegt. Erst wenn dieses Feld voll ist, werden die Daten weitergereicht. Entsprechendes gilt für die Eingabe. Damit wird die Anzahl der darunter liegenden Betriebssystemaufrufe reduziert. Bei der Ein-

und Ausgabe einer großen Datenmenge kann dadurch die Laufzeit um mehrere Größenordnungen verbessert werden. Daher ist zu empfehlen, die gepufferten Klassen immer zu verwenden, wenn nicht andere Gründe gegen deren Verwendung sprechen. Wenn wir diese Pufferfunktionalität in obiges Beispiel beim Schreiben von Int-Werten in eine Datei integrieren möchten, dann ist das einfach möglich, indem wir ein BufferedOutputStream-Objekt zwischen das FileOutputStream- und das DataOutputStream-Objekt stecken:

```
FileOutputStream fout = new FileOutputStream("x.y");
BufferedOutputStream bout = new BufferedOutputStream(fout);
DataOutputStream dout = new DataOutputStream(bout);
dout.writeInt(4711);
```

Die entstandene Ausgabe-Pipeline ist in Bild 5.7 dargestellt.



Außerhalb des Packages `java.io` gibt es weitere Ein-/Ausgabeklassen mit Zusatzfunktionalität. So gibt es Klassen zum *Komprimieren* und *Dekomprimieren* (wie beim bekannten ZIP-Programm) oder zum *Verschlüsseln* und *Entschlüsseln* von Daten. Darauf wollen wir aber an dieser Stelle nicht näher eingehen.

Kommen wir nun zu unserem eigentlichen Thema Sockets zurück. Wie im UDP-Beispiel wollen wir auch im TCP-Beispiel Strings schicken. Strings bestehen aus Zeichen. Deshalb benötigen wir Klassen zum Lesen und Schreiben von Zeichenströmen. Mit den Methoden `getInputStream` und `getOutputStream` der Klasse `Socket` erhalten wir aber Byteströme. Die Klassen `InputStreamReader` und `OutputStreamWriter` helfen an dieser Stelle weiter; sie stellen nämlich Adapter-Klassen dar, die einen Byte-Eingabestrom in einen Zeichen-Eingabestrom bzw. einen Zeichen-Ausgabestrom in einen Byte-Ausgabestrom wandeln. Da wir noch zusätzlich die Pufferungsfunktion einsetzen wollen, ergibt sich z.B. für die Ausgabe (d.h. das Senden von Strings) folgender Programmcode:

```
Socket s = ...;
OutputStream os = s.getOutputStream();
OutputStreamWriter osw = new OutputStreamWriter(os);
BufferedWriter bw = new BufferedWriter(osw);
```

Etwas kompakter lässt sich das auch so schreiben:

```
Socket s = ...;
BufferedWriter bw = new BufferedWriter(
    new OutputStreamWriter(
        s.getOutputStream()));
```

Die Klasse *BufferedWriter* hat u. a. eine Write-Methode zum Schreiben eines Strings. Mit der folgenden Anweisung kann ein String über eine TCP-Verbindung gesendet werden:

```
bw.write("Hallo Welt");
```

Bei der Kommunikation über TCP ist nun aber noch zu beachten, dass TCP datenstromorientiert ist. Das heißt, es kann passieren, dass mehrere hintereinander gesendete Strings beim Empfangen zu einem einzigen, langen String zusammengeklebt werden. Oder es ist umgekehrt möglich, dass ein sehr langer String, der mit einem einzigen write-Aufruf abgesendet wird, häppchenweise auf der anderen Seite ankommt. Beim Empfangen wird somit zuerst eventuell nur der vordere Teil zurückgeliefert, so dass man den gesamten String durch mehrmaliges Aufrufen der read-Methode wieder „zusammenbasteln“ muss. Dadurch entstehen zwar keine Fehler, aber die Anwendung hat gewisse Mühe, die gelesenen Daten richtig zu interpretieren. Zur Verbesserung dieser Situation gibt es mehrere Möglichkeiten. Eine ist z. B., zuerst immer die Länge des Strings zu senden und beim Lesen dann genauso viele Zeichen zu lesen. Eine andere Möglichkeit ist die Benutzung von Trennzeichen. Davon wird im folgenden Beispielprogramm Gebrauch gemacht. Und zwar verwenden wir wie andere bekannte *ASCII-Protokolle* (z. B. *HTTP*, *SMTP*, *POP*) als Trennzeichen das Newline-Symbol. Die Klasse *BufferedWriter* besitzt sogar eine spezielle Methode namens *newLine*, um dieses Zeichen zu schreiben. Ferner wenden wir die Methode *flush* auf unser *BufferedWriter*-Objekt an. Damit stellen wir sicher, dass die gepufferten Daten an das darunter liegende Ausgabe-Objekt ausgegeben werden und damit sofort über das Internet verschickt werden. Das Senden eines Strings mit abschließendem Newline-Symbol geschieht also wie folgt:

```
bw.write("Hallo Welt");
bw.newLine();
bw.flush();
```

Das Lesen ist einfacher. Zunächst wird analog zur Ausgabe eine Eingabe-Pipeline erzeugt:

```
Socket s = ...;
BufferedReader br = new BufferedReader(
    new InputStreamReader(
        s.getInputStream()));
```

Die Klasse *BufferedReader* stellt die Methode *readLine* bereit, mit der aus einem Zeichenstrom so lange gelesen wird, bis das Newline-Symbol erkannt wird. Die gelesenen Zeichen werden dann als String ohne das Newline-Symbol als Rückgabewert zurückgeliefert (das Newline-Symbol wird somit von dieser Klasse „verschluckt“).

```
String message = br.readLine();
```

Das Ende eines Datenstroms wird durch den Rückgabewert null angezeigt. Wird eine Datei gelesen, so bedeutet dies das Ende der Datei. In unserem Fall, wo es um eine TCP-Verbin-

dung geht, bedeutet der Rückgabewert null, dass der Partner die TCP-Verbindung geschlossen hat.

Bitte beachten Sie, dass das Lesen von einem Socket den Aufrufer solange blockiert, bis Daten angekommen sind. Dies gilt aber nur, solange die Verbindung noch geöffnet ist und folglich die Möglichkeit besteht, dass noch Daten ankommen. Wenn die Verbindung geschlossen wurde, dann erfolgt keine Blockierung mehr beim Lesen (in welcher Form die Lesemethode das Schließen der Verbindung anzeigt, hängt von der Lesemethode der entsprechenden Klasse ab; für readLine wird es – wie oben besprochen – durch null als Rückgabewert angezeigt).

Ähnlich wie bei UDP kapseln wir die immer wiederkehrenden Aktionen in einer selbst geschriebenen Klasse. Diese Klasse nennen wir TCPSocket (s. Listing 5.7). Wie die Klasse UDPSocket hat TCPSocket zwei Konstruktoren, einen für den Client und einen für den Server. Der Client-Konstruktor erzeugt ein Socket-Objekt explizit. Die dazu benötigten Parameter, nämlich der Name oder die IP-Adresse des Server-Rechners und die Portnummer des Servers, werden als Parameter im Konstruktor übergeben. Auf der Server-Seite wird das Socket-Objekt von der accept-Methode der Klasse ServerSocket geliefert. Deshalb hat unsere Klasse TCPSocket für den Server einen zweiten Konstruktor, in dem ein bereits vorhandenes Socket-Objekt übergeben und das dann im Folgenden verwendet wird. Mit diesen Ausführungen sollten Sie nun eigentlich keine Probleme mehr beim Lesen der Klasse TCPSocket haben:

**Listing 5.7**

```
import java.io.*;
import java.net.*;

public class TCPSocket implements AutoCloseable
{
    private Socket socket;
    private BufferedReader istream;
    private BufferedWriter ostream;

    public TCPSocket(String serverAddress, int serverPort)
        throws UnknownHostException, IOException
    {
        socket = new Socket(serverAddress, serverPort);
        initializeStreams();
    }

    public TCPSocket(Socket socket) throws IOException
    {
        this.socket = socket;
        initializeStreams();
    }

    public void sendLine(String s) throws IOException
    {
        ostream.write(s);
        ostream.newLine();
        ostream.flush();
    }
}
```

```

public String receiveLine() throws IOException
{
    return istream.readLine();
}

public void close() throws IOException
{
    socket.close();
}

private void initializeStreams() throws IOException
{
    ostream = new BufferedWriter(
        new OutputStreamWriter(
            socket.getOutputStream()));
    istream = new BufferedReader(
        new InputStreamReader(
            socket.getInputStream()));
}
}

```

Wie im UDP-Beispiel implementiert auch diese Klasse die Schnittstelle AutoCloseable. Ebenfalls wie im UDP-Beispiel ist es nun leichter, unseren Server und Client zu programmieren. Die folgende Client-Server-Anwendung ist dieselbe wie für UDP. Der Server besitzt einen Zähler, der durch die Kommandos „increment“ und „reset“ erhöht bzw. auf 0 zurückgesetzt werden kann (s. Bild 5.4). Anders als im Pseudocode schließt der Client seine Verbindung nicht explizit, sondern verlässt seinen Try-with-resources-Block, wodurch die Verbindung geschlossen wird. An dem Rückgabewert null der Empfangsmethode erkennt der Server, dass der Client die Verbindung geschlossen hat und somit nichts mehr senden wird. Anders als im Pseudocode (s. Abschnitt 5.2.2) schließt auch der Server seine Verbindung zum Client nicht explizit, sondern verlässt lediglich mit break seine innere Schleife und gelangt damit an das Ende des inneren Try-with-resources-Block, wodurch nun auch implizit die Verbindung von der Server-Seite aus geschlossen wird. Danach findet ein erneuter Durchlauf der äußeren Schleife statt, der mit einem Warten auf den nächsten Client beginnt. Der Java-Code des Servers ist in Listing 5.8 enthalten:

### **Listing 5.8**

```

import java.net.*;

public class Server
{
    public static void main(String[] args)
    {
        int counter = 0;

        // create socket
        try(ServerSocket serverSocket = new ServerSocket(1250))
        {
            while(true)
            {
                // wait for connection then create streams
                System.out.println("Warten auf Verbindungsauftbau");
                try(TCPSocket tcpSocket =

```

```
new TCPSocket(serverSocket.accept()))
{
    // execute client requests
    while(true)
    {
        String request = tcpSocket.receiveLine();
        if(request != null)
        {
            if(request.equals("increment"))
            {
                // perform increment operation
                counter++;
            }
            else if(request.equals("reset"))
            {
                // perform reset operation
                counter = 0;
                System.out.println("Der Zähler "
                    + "wurde "
                    + "zurückgesetzt");
            }
            String result = String.valueOf(counter);
            tcpSocket.sendLine(result);
        }
        else
        {
            System.out.println("Schließen der "
                + "Verbindung");
            break;
        }
    }
    catch(Exception e)
    {
        System.out.println(e);
        System.out.println(">= Verbindung geschlossen");
    }
}
catch(Exception e)
{
    System.out.println("Fehler beim Erzeugen oder Nutzen "
        + "des ServerSockets");
    return;
}
}
```

Der Client ist ebenfalls analog zum UDP-Beispiel gestaltet. Er muss mit zwei Kommandozeilen-Argumenten, nämlich dem Rechnernamen oder der IP-Adresse des Servers und der Anzahl der zu sendenden „increment“-Kommandos, gestartet werden. Ebenfalls misst auch dieser Client die Ausführungszeit. Anders als beim UDP-Client müssen wir uns aber nicht überlegen, wie wir auf den Verlust von Nachrichten reagieren sollen, da TCP zuverlässig ist. Die ersten beiden Aktionen des Client-Pseudocodes (Socket erzeugen und Verbindung aufbauen, s. Abschnitt 5.2.2) können in Java mit einer einzigen Anweisung durchgeführt werden. Der Code des TCP-Clients ist in Listing 5.9 enthalten:

**Listing 5.9**

```
public class Client
{
    public static void main(String args[])
    {
        if(args.length != 2)
        {
            System.out.println("Notwendige Kommandozeilenargumente:"
                               + " <Name des Server-Rechners>"
                               + " <Anzahl>");
            return;
        }

        // create socket connection
        System.out.println("Aufbau der Verbindung");
        try(TCPSocket tcpSocket = new TCPSocket(args[0], 1250))
        {
            // set counter to zero
            System.out.println("Zähler zurücksetzen");
            tcpSocket.sendLine("reset");
            String reply = tcpSocket.receiveLine();

            // get count, initialize start time
            System.out.println("Zähler erhöhen");
            int count = new Integer(args[1]).intValue();
            long startTime = System.currentTimeMillis();

            // perform increment "count" number of times
            for(int i = 0; i < count; i++)
            {
                tcpSocket.sendLine("increment");
                reply = tcpSocket.receiveLine();
            }

            // display statistics
            long stopTime = System.currentTimeMillis();
            long duration = stopTime - startTime;
            System.out.println("Gesamtzeit = " + duration
                               + " msecs");
            if(count > 0)
            {
                System.out.println("Durchschnittszeit = "
                                   + ((duration) / (float) count)
                                   + " msecs");
            }
            System.out.println("Letzter Zählerstand: " + reply);
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("TCP-Verbindung wurde geschlossen");
    }
}
```

Da das von uns definierte und von der Anwendung verwendete Anwendungsprotokoll ein ASCII-Protokoll ist, kann unser TCP-Server auch über einen TELNET-Client angesprochen werden. Starten Sie zu diesem Zweck das TELNET-Programm mit einem optionalen zweiten Argument, nämlich der Portnummer des Servers (in unserem Fall 1250). Wenn auf Ihrem Rechner ein Server läuft, dann können Sie z. B. folgendes Kommando eingeben:

```
telnet localhost 1250
```

Damit wird eine Verbindung zu Ihrem Server aufgebaut. Alles, was Sie nun eintippen, wird zum Server geschickt. Wenn Sie also „increment“ oder „reset“ eingeben und dann die Return-Taste drücken, dann sollte die entsprechende Aktion auf dem Server ausgeführt werden, und Sie sollten eine entsprechende Antwort von Ihrem Server auf dem Bildschirm sehen. Wenn Sie irgendetwas anderes eingeben (z. B. eine Leerzeile durch Drücken der Return-Taste), dann können Sie damit den aktuellen Zählerstand vom Server abfragen, ohne dass Änderungen am Zähler vorgenommen werden.

## ■ 5.6 Sequenzielle und parallele Server

Sinn und Zweck von Client-Server-Anwendungen ist u. a. die Nutzung eines Servers durch mehrere Clients. Dabei wird es in der Regel vorkommen, dass mehrere Clients einen Server sogar gleichzeitig nutzen. Was bedeutet das nun für unsere UDP- und TCP-Beispiele?

- Im UDP-Fall kann zwar ein Server das folgende Kommando erst dann ausführen, wenn er die Arbeit am aktuell bearbeiteten Kommando beendet und die Antwort zurückgeschickt hat. Aber immerhin werden mehrere Clients „verzahnt“ bedient. Das heißt im Fall von z. B. zwei Clients, dass erst ein Kommando des ersten Clients, dann eines des zweiten Clients, dann wieder eines des ersten usw. ausgeführt wird. Dies entspricht unserem intuitiven Verständnis von Fairness.
- Im TCP-Fall ist dies anders. Hier werden erst alle Kommandos eines einzigen Clients bearbeitet, bevor der zweite bedient wird. Dies liegt an der Programmstruktur unseres TCP-Servers. Nach der Annahme einer TCP-Verbindung liest der Server nur von dieser Verbindung. Erst wenn der Client die Verbindung geschlossen hat, nimmt der Server die nächste Verbindung an und wendet sich damit dem nächsten Client zu. Dieses Verhalten ist besonders ärgerlich in dem Fall, dass ein Server über einen interaktiv bedienbaren Client wie z. B. den TELNET-Client benutzt wird. Selbst wenn der Benutzer des TELNET-Clients keine Kommandos eingibt, sondern für eine halbe Stunde Kaffee trinken gegangen ist, ist der Server blockiert. Er kann keine Kommandos für andere Clients ausführen, obwohl er untätig ist, da er auf Kommandos eines bestimmten Clients wartet. Die meisten Leser stimmen mir wohl zu, wenn ich ein solches Verhalten als unfair beurteile. Im Internet eingesetzte Server wie Web-Server, Mail-Server usw. würden von den Anwendern wenig geschätzt werden, wenn sie sich so verhalten würden wie unser Beispiel-Server.

Beide Server-Arten (UDP und TCP) arbeiten sequenziell (d. h. ohne Parallelität). Es ist möglich, für beide Server-Arten parallele Versionen zu entwickeln. Wie die vorigen Erläuterungen gezeigt haben, ist dies vor allem für den TCP-Fall notwendig. Deshalb wollen wir in

diesem Abschnitt parallele Versionen nur für TCP-Server vorstellen, obwohl man UDP-Server ebenfalls parallelisieren könnte. Es sei noch angemerkt, dass das als unfair bezeichnete Verhalten unseres bisherigen TCP-Servers auch ohne Einführung von Parallelität verbessert werden kann. Auf diese Realisierungsmöglichkeit gehen wir am Ende dieses Abschnitts näher ein.

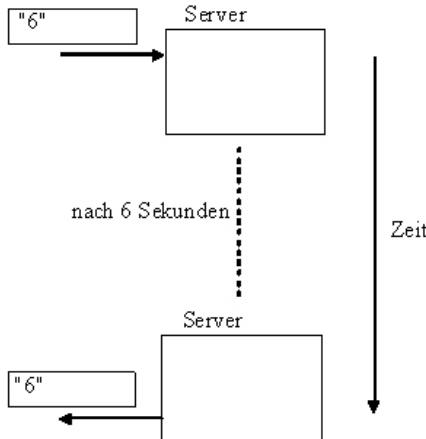
Für die Realisierung eines parallelen TCP-Servers gibt es mehrere Alternativen:

- *Statische Parallelität*: Es werden mehrere parallel ablaufende Threads zu Beginn gestartet, die sich die Arbeit teilen. Die Anzahl der Threads bleibt somit immer gleich (statisch). Diese Arbeitsweise kennen wir aus dem täglichen Leben. Stellen Sie sich dazu z.B. einen Raum vor, in dem an mehreren Schaltern Kunden bedient werden (Post, Fahrkartenschalter der Deutschen Bahn usw.). Immer wenn ein Schalter frei wird, geht der nächste Kunde, der an der Reihe ist, an den freien Schalter und wird bedient.
- *Dynamische Parallelität*: Hier wird für jeden neuen Kunden ein neuer Thread gestartet, der den Kunden betreut. Nachdem die Verbindung zu dem Kunden geschlossen wird, endet der Thread. Die Anzahl der Threads ändert sich daher im Lauf der Zeit ständig (dynamisch). Dies würde für das Beispiel aus dem täglichen Leben bedeuten, dass ein Kunde zunächst von einem Empfangschef begrüßt wird. Dieser Empfangschef zaubert dann für diesen Kunden einen neuen Schalter samt einer neuen Bedienperson, die den Kunden bedient. Sobald der Kunde den Schalter verlässt, lösen sich Schalter und Bedienperson in nichts auf. Dieses Verhalten ist uns aus dem täglichen Leben eher weniger geläufig. In Software ist es aber leicht realisierbar.
- Mischformen: Sowohl die statische als auch die dynamische Form der Parallelität hat Vor- und Nachteile. Deshalb gibt es auch Mischformen. So können z.B. zu Beginn eine bestimmte Anzahl paralleler Threads gestartet werden. Sobald die Anzahl der Kunden die Anzahl der Threads übersteigt, können (evtl. bis zu einem vorgegebenen Maximum) weitere Threads dazugeschaltet werden.

In diesem Abschnitt werden wir je ein Beispiel für die statische und die dynamische Form der Parallelität realisieren. Als Beispiel der dynamischen Parallelitätsform wird ein neues Beispiel besprochen. Anhand dieses Beispiels lassen sich besonders gut die Verhaltensunterschiede einer sequenziellen und parallelen Version eines Servers demonstrieren. Als Beispiel für die statische Form der Parallelität greifen wir wieder auf unser ursprüngliches Beispiel mit einem Zähler zurück und verwenden zur Realisierung der Parallelität einen Thread-Pool.

### 5.6.1 TCP-Server mit dynamischer Parallelität

Der TCP-Server arbeitet wie ein Echo-Server. Das heißt, jede empfangene Zeile wird zurückgesendet. Zusätzlich besitzt der Server die Funktionalität, dass er beim Empfang einer Eingabezeile, die aus einer ganzen Zahl besteht, seine Echo-Antwort um so viele Sekunden, wie die Zahl angibt, verzögert. Dieses Verhalten wird in Bild 5.8 veranschaulicht. Damit wird es möglich, den Server so lange zu beschäftigen, wie man will.

**Bild 5.8**

Verhaltensweise des TCP-Echo-Servers

Um den Vorteil, den die parallele Version bringen wird, besser einschätzen zu können, wird zuerst eine sequentielle Server-Version (Listing 5.10) gezeigt. Da dieses Programm dem TCP-Server aus dem vorigen Abschnitt sehr ähnlich ist, sind keine weiteren Erläuterungen nötig.

**Listing 5.10**

```
import java.net.*;

public class SequentialServer
{
    public static void main(String[] args)
    {
        // create socket
        try(ServerSocket serverSocket = new ServerSocket(1250))
        {
            while(true)
            {
                // wait for connection then create streams
                System.out.println("Warten auf Verbindungsauflaufbau");
                try(TCPSocket tcpSocket =
                    new TCPSocket(serverSocket.accept()))
                {
                    // execute client requests
                    while(true)
                    {
                        String request = tcpSocket.receiveLine();
                        if(request != null)
                        {
                            // perform sleep
                            try
                            {
                                int secs =
                                    Integer.parseInt(request);
                                Thread.sleep(secs * 1000);
                            }
                            catch(Exception e)
                            {

```

```
        System.out.println(e);
    }
    tcpSocket.sendLine(request);
}
else
{
    System.out.println("Schließen der "
                       + "Verbindung");
    break;
}
}
catch(Exception e)
{
    System.out.println(e);
    System.out.println(">=> Schließen der "
                       + "Verbindung");
}
}
catch(Exception e)
{
    System.out.println("Fehler beim Erzeugen oder Nutzen "
                       + "des ServerSockets");
}
}
}
```

Die parallele Version unseres TCP-Servers lässt sich wie in Listing 5.11 realisieren:

#### Listing 5.11

```
import java.net.*;

public class ParallelDynamicServer
{
    public static void main(String[] args)
    {
        // create socket
        try(ServerSocket serverSocket = new ServerSocket(1250))
        {
            while(true)
            {
                // wait for connection then create streams
                System.out.println("Warten auf Verbindungsaufbau");
                try
                {
                    TCPSocket tcpSocket =
                        new TCPSocket(serverSocket.accept());
                    new Slave(tcpSocket);
                }
                catch(Exception e)
                {
                    System.out.println(e);
                }
            }
        }
        catch(Exception e)
```

```

        {
            System.out.println("Fehler beim Erzeugen oder Nutzen "
                               + "des ServerSockets");
        }
    }

class Slave extends Thread
{
    private TCPSocket socket;

    public Slave(TCPSocket socket)
    {
        this.socket = socket;
        this.start();
    }

    public void run()
    {
        try(TCPSocket s = socket)
        {
            while(true)
            {
                String request = s.receiveLine();
                // execute client requests
                if(request != null)
                {
                    try
                    {
                        int secs = Integer.parseInt(request);
                        Thread.sleep(secs * 1000);
                    }
                    catch(InterruptedException e)
                    {
                        System.out.println(e);
                    }
                    s.sendLine(request);
                }
                else
                {
                    break;
                }
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("Verbindung geschlossen");
    }
}

```

Bitte beachten Sie, dass in der Main-Methode das Try innerhalb der While-Schleife kein Try-with-resources sein darf, da sonst sofort nach dem Starten des Slave-Threads der Socket geschlossen werden würde. Somit könnte der Thread dann die Verbindung gar nicht nutzen. Stattdessen wird in der Run-Methode der Klasse Slave ein Try-with-resources verwendet. Damit dies möglich ist, muss eine neue Variable eines AutoCloseable-Typs deklariert wer-

den. Diesem Zweck dient die Variable s vom Typ TCPSocket, wobei in diesem Fall der TCP-Socket nicht erzeugt werden muss, da dieser schon im Konstruktor übergeben wurde.

Der dazu passende Client wird mit denselben Kommandozeilen-Argumenten wie der UDP-Client aus dem Multicast-Beispiel gestartet: Das erste Argument ist der Name oder die IP-Adresse des Servers. Die folgenden Argumente, die in beliebiger Anzahl folgen können, werden als Strings der Reihe nach an den Server geschickt. Der Client ist in Listing 5.12 implementiert:

#### **Listing 5.12**

```
public class Client
{
    public static void main(String args[])
    {
        if(args.length < 2)
        {
            System.out.println("Notwendige Kommandozeilenargumente:"
                + " <Name des Server-Rechners>"
                + " <Sekundenzahl1> ..."
                + " < SekundenzahlN>");
            return;
        }

        // create socket connection
        System.out.println("Aufbau der Verbindung");
        try(TCPSocket tcpSocket = new TCPSocket(args[0], 1250))
        {
            // perform sleep operations
            for(int i = 1; i < args.length; i++)
            {
                tcpSocket.sendLine(args[i]);
                String result = tcpSocket.receiveLine();
                System.out.println(result);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
        }
        System.out.println("Verbindung geschlossen");
    }
}
```

Wie im vorigen Beispiel können Sie beim Ausprobieren des sequenziellen und parallelen TCP-Servers neben dem selbst geschriebenen Client auch einen TELNET-Client verwenden.

#### **5.6.2 TCP-Server mit statischer Parallelität**

Die dynamische Form der Parallelität birgt die Gefahr, dass damit bei einem Denial-of-Service-Angriff so viele Threads auf dem Server erzeugt werden, dass der Rechner vollkommen überlastet wird. Ein solcher Angriff kann leicht programmiert werden, indem man sehr viele Threads erzeugt, von denen jeder eine Verbindung zum Server öffnet, ohne diese wieder zu schließen. Der Effekt kann noch verstärkt werden, wenn man diesen Angriff

gleichzeitig von mehreren Rechnern aus startet. Wir werden deshalb im folgenden Beispiel die Anzahl der Threads beschränken.

Als Beispiel verwenden wir wieder unser ursprünglich eingeführtes Zählerbeispiel. Da der Zähler nun von mehreren Threads gleichzeitig benutzt wird, wird er nicht mehr als lokale Variable der Main-Methode, sondern als Attribut einer Klasse realisiert. In dieser Klasse existieren Methoden zum Lesen und Ändern (Erhöhen und Zurücksetzen) des Zählers. Da ein Objekt dieser Klasse von mehreren Threads benutzt wird, müssen alle lesenden und schreibenden Methoden synchronized sein. Die statische Parallelität wird dadurch erreicht, dass zu Beginn eine feste Anzahl von Threads erzeugt wird. Jeder dieser Threads wartet mit accept auf demselben ServerSocket-Objekt auf eine neue Verbindung. Falls eine Verbindung von einem Client aufgebaut wird, dann nimmt irgendeiner der bereiten Threads die Verbindung an; die gerade beschäftigten Threads führen keinen Aufruf von accept durch und kommen deshalb zur Betreuung der neuen Verbindung nicht in Frage. Listing 5.13 zeigt das Programm des Servers mit statischer Parallelität.

**Listing 5.13**

```
import java.net.*;  
  
class Counter  
{  
    private int counter;  
  
    public synchronized int increment()  
    {  
        counter++;  
        return counter;  
    }  
  
    public synchronized int reset()  
    {  
        counter = 0;  
        return counter;  
    }  
  
    public synchronized int getCounter()  
    {  
        return counter;  
    }  
}  
  
public class ParallelStaticServer  
{  
    private static final int NUMBER_OF_SLAVES = 20;  
  
    public static void main(String[] args)  
    {  
        ServerSocket serverSocket = null;  
        Counter counter = new Counter();  
        try  
        {  
            // create socket  
            serverSocket = new ServerSocket(1250);  
        }
```

```
        catch(Exception e)
        {
            System.out.println("Fehler beim Erzeugen des "
                               + "ServerSockets");
            return;
        }
        for(int i = 0; i < NUMBER_OF_SLAVES; i++)
        {
            Thread t = new StaticSlave(serverSocket, counter);
            t.start();
        }
    }

class StaticSlave extends Thread
{
    private ServerSocket serverSocket;
    private Counter counter;

    public StaticSlave(ServerSocket serverSocket, Counter counter)
    {
        this.serverSocket = serverSocket;
        this.counter = counter;
    }

    public void run()
    {
        while(true)
        {
            // wait for connection then create streams
            try(TCPSocket tcpSocket =
                new TCPSocket(serverSocket.accept()))
            {
                while(true)
                {
                    String request = tcpSocket.receiveLine();
                    // execute client requests
                    int result;
                    if(request != null)
                    {
                        if(request.equals("increment"))
                        {
                            // perform increment operation
                            result = counter.increment();
                        }
                        else if(request.equals("reset"))
                        {
                            // perform reset operation
                            result = counter.reset();
                            System.out.println("Zähler wurde "
                                              + "zurückgesetzt");
                        }
                        else
                        {
                            result = counter.getCounter();
                        }
                        tcpSocket.sendLine("" + result);
                    }
                }
            }
        }
    }
}
```

```

            else
            {
                System.out.println("Schließen der "
                    + "Verbindung");
                break;
            }
        }
    catch(Exception e)
    {
        System.out.println(e);
        System.out.println(">=> Schließen der Verbindung");
    }
}
}
}
}
}
```

In diesem Fall kann für den ServerSocket kein Try-with-resources benutzt werden. Der Grund dafür sollte offensichtlich sein.

Alternativ kann zur Realisierung der statischen Parallelität auch ein Thread-Pool (s. Abschnitt 3.7.1) verwendet werden (s. Listing 5.14). Der von einem Thread durchgeföhrte Programmcode befindet sich in der Methode run der Klasse Task. Objekte der Klasse Task werden an den Thread-Pool zur Ausführung übergeben. Deshalb muss die Klasse Task die Schnittstelle Runnable implementieren. Im Code unseres Servers wird kein Thread explizit gestartet.

#### Listing 5.14

```

import java.net.*;
import java.util.concurrent.*;

class Counter
{
    //wie zuvor
    ...
}

class Task implements Runnable
{
    private TCPsocket socket;
    private Counter counter;

    public Task(TCPsocket socket, Counter counter)
    {
        this.socket = socket;
        this.counter = counter;
    }

    public void run()
    {
        try(TCPSocket s = socket)
        {
            while(true)
            {
                String request = socket.receiveLine();
                int result;
```

```
        if(request != null)
        {
            if(request.equals("increment"))
            {
                // perform increment operation
                result = counter.increment();
            }
            else if(request.equals("reset"))
            {
                // perform reset operation
                result = counter.reset();
            }
            else
            {
                result = counter.getCounter();
            }
            socket.sendLine("" + result);
        }
        else
        {
            break;
        }
    }
}
catch(Exception e)
{
    System.out.println(e);
}
}

public class ParallelStaticServerWithThreadPool
{
    public static void main(String[] args)
    {
        Counter counter = new Counter();
        ThreadPoolExecutor pool =
            new ThreadPoolExecutor(3, 3,
                                  0L, TimeUnit.SECONDS,
                                  new LinkedBlockingQueue<Runnable>());

        try(ServerSocket serverSocket = new ServerSocket(1250))
        {
            while(true)
            {
                try
                {
                    TCPsocket tcpSocket =
                        new TCPsocket(serverSocket.accept());
                    Task task = new Task(tcpSocket, counter);
                    pool.execute(task);
                }
                catch(Exception e)
                {
                    System.out.println(e);
                }
            }
        }
    }
}
```

```
        catch(Exception e)
    {
        System.out.println("Fehler beim Erzeugen oder Nutzen "
                           + "des ServerSockets");
    }
}
```

Eine vergleichende Betrachtung dieser beiden parallelen Varianten mit der sequenziellen Version des Servers (Listing 5.8) wird den Leserinnen und Lesern empfohlen. Im Abschnitt 5.5 befindet sich das Client-Programm (Listing 5.9), das auch für die beiden parallelen Server-Versionen benutzt werden kann. Alternativ kann auch wieder TELNET als Client verwendet werden.

Wenn wir den Konstruktor des ThreadPoolExecutors in Listing 5.14 mit anderen Parametern versorgen, dann erhalten wir auf einfache Art und Weise einen Server, der eine Mischform zwischen statischer und dynamischer Parallelität realisiert. Dazu müssen wir lediglich für den zweiten Parameter (maximumPoolSize) einen größeren Wert wählen als für den ersten Parameter (corePoolSize) und als letzten Parameter eine Queue-Typ verwenden, der im Gegensatz zu LinkedBlockingQueue ein begrenztes Fassungsvermögen hat (vgl. dazu Abschnitt 3.7.1). In diesem Fall werden dann zusätzliche Threads erzeugt, wenn die Warteschlange voll ist und weitere Aufträge ankommen. Diese zusätzlichen Threads werden wieder beendet, sobald die Belastung des Servers sinkt.

### 5.6.3 Sequenzieller, „verzahnt“ arbeitender TCP-Server

Wie am Anfang dieses Abschnitts erwähnt wurde, arbeitet der UDP-Server „verzahnt“. Das heißt, dass er zwar nicht mehrere Aufträge gleichzeitig ausführen, aber abwechselnd die Aufträge mehrerer gleichzeitig aktiver Kunden bearbeiten kann (also beispielsweise einen Auftrag des ersten Kunden, dann einen Auftrag eines anderen Kunden, danach den nächsten Auftrag des ersten Kunden usw.). Im Gegensatz dazu muss der sequenzielle TCP-Server alle Aufträge eines einzigen Kunden erst vollständig bearbeiten, bevor er den nächsten Kunden bedienen kann. Wenn also die einzelnen Aufträge der Kunden sehr schnell ausführbar sind und es nicht allzu viele Kunden gleichzeitig gibt, ein Kunde aber relativ viele Aufträge erteilt und die Zeit zwischen je zwei Aufträgen eines Kunden relativ lang ist, dann kommt der sequenzielle UDP-Server mit dieser Situation sehr gut zurecht, während es beim sequenziellen TCP-Server zu unnötig langen Wartzeiten kommt. In einem solchen Fall muss allerdings auch ein TCP-Server nicht unbedingt parallel arbeiten. Es genügt, wenn er wie der UDP-Server in die Lage versetzt wird, „verzahnt“ zu arbeiten.

Zu diesem Zweck muss er mehrere Verbindungen zu unterschiedlichen Clients gleichzeitig unterhalten. Er darf nun aber nicht zyklisch Daten von den Verbindungen in blockierender Weise lesen, denn dann könnte es passieren, dass er auf den Auftrag eines Clients wartet, der längere Zeit pausiert, während die Aufträge anderer Clients nicht bedient werden. Der TCP-Server muss also sozusagen alle Verbindungen gleichzeitig im Auge behalten und immer von derjenigen lesen, auf der gerade Daten anliegen. Dabei darf er den ServerSocket nicht vergessen, über den neue Verbindungen eintreffen können. Eine Realisierung, bei der periodisch der ServerSocket und alle Verbindungen abgeklappert werden, um zu prüfen, ob

etwas anliegt, würde aber aktives Warten bedeuten, was wir nicht verwenden wollen. Wir brauchen eine Art ODER-Warten (man wartet, bis dieses oder das oder jenes eintritt).

In der *Java-NIO-Bibliothek (New Input/Output)* ist ein solches ODER-Warten in Form eines *Selectors* realisiert. An einem Selector können sich sogenannte *Channels* anmelden. Sowohl für ServerSockets als auch für Sockets gibt es dazugehörige Channels. Ein Channel kann sich an einem Selector registrieren. Dabei muss auch angegeben werden, auf welche möglichen Ereignisse des Channels der Selector das ODER-Warten anwenden soll.

In Listing 5.15 ist der Programmcode eines sequenziellen, „verzahnt“ arbeitenden TCP-Servers zu sehen. Als Beispiel realisieren wir jetzt wieder den Echo-Server mit sleep aus Abschnitt 5.6.1. Ohne auf die Details einzugehen sei nur das wesentliche Prinzip des Programms erwähnt:

Zu Beginn wird über einen *ServerSocketChannel* ein *ServerSocket* erzeugt. Außerdem wird ein Selector angelegt und der *ServerSocketChannel* meldet sich am Selector für Accept-Ereignisse an (d.h. man kehrt beim Warten über den Selector zurück, falls auf dem *ServerSocket* *accept* ohne Blockierung ausgeführt werden kann). In einer Endlosschleife wendet man dann *select* auf den Selector an. Die Methode *select* ist eine Realisierung des ODER-Wartens, die den Aufrufer blockieren kann. Wenn man aus *select* zurückkehrt, lässt man sich alle Elemente zurückgeben, die das Warten beendet haben. In einer inneren Schleife untersucht man alle diese Elemente. Wenn ein Element anzeigt, dass ein *Accept*-Aufruf jetzt ohne Blockade möglich ist, dann wird die Methode *accept* aufgerufen, wobei *accept* nicht auf den *ServerSocket*, sondern auf den *ServerSocketChannel* angewendet wird, was wieder einen Channel, nämlich einen *SocketChannel*, zurückliefert. In diesem Beispiel befindet sich im Selector immer genau ein *ServerSocketChannel*, so dass man sich für den *Accept*-Fall diesen nicht beschaffen müsste, sondern es könnte der Channel aus der ersten Anweisung der Main-Methode verwendet werden. Das Beispiel zeigt aber damit auch, wie man auf Verbindungsaufbauwünsche unterschiedlicher *ServerSockets* gleichzeitig warten und wie man dann damit umgehen könnte. Der von *accept* zurückgelieferte *SocketChannel* registriert sich am Selector für Read-Ereignisse (d.h. man kehrt beim Warten über den Selector zurück, falls auf dem *Socket* *read* ohne Blockierung ausgeführt werden kann). Zeigt das in der inneren Schleife untersuchte Element an, dass *read* möglich ist, wird der Auftrag gelesen, nach Möglichkeit als Zahl interpretiert und eine Antwort gesendet. Bitte beachten Sie, dass das Programm an dieser Stelle eigentlich in unzulässiger Weise vereinfacht wurde; die Datenstromorientierung von TCP, die bisher immer so deutlich herausgestellt wurde, wird jetzt der Einfachheit halber in Listing 5.15 vollkommen vernachlässigt. Es wird stattdessen immer davon ausgegangen, dass die Daten eines Kommandos vollständig vorhanden sind, dass sie komplett in den bereitgestellten Puffer passen und dass die letzten zwei Bytes die Codierung für Newline enthalten. Bei unseren kurzen Kommandos wird dies mit hoher Wahrscheinlichkeit die meiste Zeit funktionieren; seriös ist es aber nicht. Übrigens müsste man auch das Zurücksenden der Antwort aus Zuverlässigen Gründen etwas aufwändiger programmieren, worauf ebenfalls verzichtet wurde. Beim Schließen einer Verbindung muss der dazugehörige Channel übrigens nicht aus dem Selector entfernt werden; dies geschieht automatisch, wie die Ausgabe zu Beginn der äußeren Schleife zeigt.

**Listing 5.15**

```

import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.util.*;

public class SequentialServerWithInterleaving
{
    public static void main(String[] args) throws Exception
    {
        try(ServerSocketChannel serverSocketChannel =
                ServerSocketChannel.open();
            ServerSocket serverSocket =
                serverSocketChannel.socket();
            Selector selector = Selector.open())
        {
            serverSocket.bind(new InetSocketAddress(1250));
            serverSocketChannel.configureBlocking(false);
            serverSocketChannel.register(selector,
                                         SelectionKey.OP_ACCEPT);
            while(true)
            {
                System.out.println("Anzahl der Elemente im "
                                   + "Selector: "
                                   + selector.keys().size());
                selector.select(); // warte auf Ereignis
                Set<SelectionKey> readyKeys =
                    selector.selectedKeys();
                Iterator<SelectionKey> iterator =
                    readyKeys.iterator();
                while(iterator.hasNext())
                {
                    // bearbeite Keys mit Ereignissen
                    SelectionKey key = iterator.next();
                    if(key.isAcceptable())
                    {
                        // Accept-Ereignis:
                        ServerSocketChannel ssc =
                            (ServerSocketChannel) key.channel();
                        SocketChannel sc = ssc.accept();
                        System.out.println("Verbindung angenommen "
                                           + "von " + sc);
                        sc.configureBlocking(false);
                        SelectionKey newKey = sc.register(selector,
                                                       SelectionKey.OP_READ
                                                       /* | SelectionKey.OP_WRITE */ );
                        ByteBuffer buffer =
                            ByteBuffer.allocate(128);
                        newKey.attach(buffer);
                    }
                    else if(key.isReadable())
                    {
                        // Read-Ereignis:
                        SocketChannel sc =
                            (SocketChannel) key.channel();
                        ByteBuffer buffer =
                            (ByteBuffer) key.attachment();
                    }
                }
            }
        }
    }
}

```

```
        buffer.clear();
        int bytes = sc.read(buffer);
        if(bytes >= 0)
        {
            System.out.print(bytes + " Byte(s) "
                + "gelesen: ");
            byte[] array =
                new byte[buffer.capacity()];
            buffer.flip();
            buffer.get(array, 0, bytes);
            String message = new String(array, 0,
                bytes-2);
            System.out.println(message);
            try
            {
                int secs =
                    Integer.parseInt(message);
                Thread.sleep(secs * 1000);
            }
            catch(Exception e)
            {
                System.out.println("sleep nicht "
                    + "möglich");
            }
            buffer.flip();
            sc.write(buffer);
        }
        else
        {
            sc.close();
            System.out.println("Verbindung "
                + "geschlossen.");
        }
    }
    /*
    else if(key.isWritable())
    {
        // Write-Ereignis:
    }
    */
    iterator.remove();
}
}
catch(Exception e)
{
    System.out.println(e);
}
}
}
```

Das Programm aus Listing 5.12 oder alternativ TELNET kann als Client für diesen Server eingesetzt werden. Bitte beachten Sie noch, dass wir hier wie in Abschnitt 5.6.2 auch das Zählerbeispiel als „verzahnt“ arbeitenden TCP-Server realisieren hätten können. Im Gegensatz zu Listing 5.13 und Listing 5.14 wäre dann aber eine Synchronisation beim Zugriff auf den Zähler nicht nötig (aber natürlich auch nicht verboten) gewesen, da hier alle Zugriffe auf den Zähler durch denselben Thread erfolgen.

## ■ 5.7 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie Threads über Prozess- und sogar Rechnergrenzen hinweg miteinander kommunizieren können. Als Transportprotokolle können dabei wahlweise UDP oder TCP verwendet werden. Die unterschiedlichen Eigenschaften der beiden Transportprotokolle schlagen dabei auf die Programmierschnittstelle durch. Für die Nutzung von UDP wurde eine Client-Server-Anwendung entwickelt, bei der ein Zähler auf einem Server von Clients erhöht und zurückgesetzt werden kann. Zur Demonstration von Multicast wurde eine weitere Anwendung mit UDP realisiert, bei der ein einfacher Echo-Server realisiert wurde. Das Zählerbeispiel wurde anschließend auch auf die Nutzung des TCP-Protokolls übertragen. Zum Abschluss dieses Kapitels wurde dann die Nützlichkeit von Parallelität für die Server-Seite diskutiert, die für das angegebene TCP-Server-Beispiel dringlicher war. Es wurden deshalb parallele Versionen von TCP-Servern vorgestellt, wobei einmal die dynamische und einmal die statische Form der Parallelität eingesetzt wurde. Außerdem wurde darauf hingewiesen, wie leicht eine Mischform zwischen statischer und dynamischer Parallelität umgesetzt werden kann. Ganz am Ende wurde gezeigt, wie das Verhalten des ursprünglich sequenziell arbeitenden TCP-Servers durch „verzahnte“ Abarbeitung der Kundenaufträge auch ohne Parallelität verbessert werden kann.

Mit der hier beschriebenen Java-Socket-Schnittstelle ist es nun möglich, im Internet gebräuchliche Anwendungsprotokolle zu realisieren (z.B. HTTP, SMTP, POP3 und IMAP basierend auf TCP und DNS sowie SNMP basierend auf UDP). Damit können entsprechende Client- oder Server-Programme geschrieben werden. In der Java-Klassenbibliothek finden sich bereits Klassen, die solche Entwicklungen erleichtern. So können z.B. mit den Klassen *URL* und *URLConnection* bequem Webseiten von einem Web-Server mit Hilfe des HTTP-Protokolls abgerufen werden, ohne dass man dazu dieses Protokoll kennen muss. Ebenso existieren Klassen, die das Versenden von elektronischer Post über SMTP oder das Abrufen elektronischer Post über POP3 oder IMAP vereinfachen. Im Internet findet man weitere Klassen zur einfacheren Nutzung weiterer Protokolle wie z.B. SNMP.

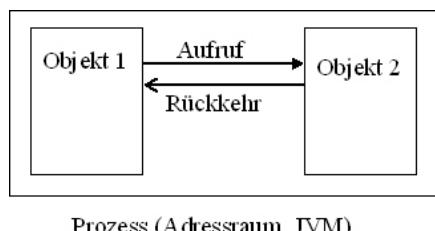
# 6

# Verteilte Anwendungen mit RMI

Wenn man die Aufgabe hat, eine verteilte Java-Anwendung von Grund auf neu zu entwickeln, insbesondere wenn dabei die zu nutzenden Kommunikationsprotokolle nicht vorgegeben sind, dann stellt RMI die bessere Alternative gegenüber den Sockets dar. Davon soll Sie dieses Kapitel überzeugen.

## ■ 6.1 Prinzip von RMI

Auch mit *RMI* (*Remote Method Invocation*) kann man verteilte Anwendungen realisieren. Der Ansatz bei den Sockets ist der, dass man eine Schnittstelle zur Nutzung der Internet-Transportprotokolle UDP und TCP bereitstellt. Bei RMI geht man davon aus, dass man heutzutage objektorientiert entwirft und entwickelt und in dieser Gedanken- und Programmierwelt auch verteilte Anwendungen realisiert. Dabei will man möglichst wenig mit der Welt der Rechnernetze zu tun haben. Anders formuliert: Mit RMI kann man verteilte Anwendungen entwickeln, wobei der Aspekt der Verteilung für die Programmiererinnen möglichst *transparent* (d. h. unsichtbar) ist.

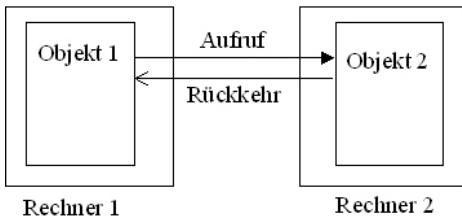


**Bild 6.1**

Lokaler Methodenauftrag: Beide Objekte befinden sich auf demselben Rechner, sogar im selben Adressraum (d. h. in demselben Prozess und derselben Java Virtual Machine [JVM])

In Bild 6.1 und Bild 6.2 ist der Unterschied zwischen einem lokalen Methodenauftrag und RMI bildlich dargestellt. Bild 6.1 illustriert einen lokalen Methodenauftrag, wobei aus einer Methode des Objekts 1 ein Methodenauftrag erfolgt, der auf ein Objekt 2 angewendet wird. Dabei befinden sich die beiden Objekte nicht nur auf einem Rechner, sondern auch im selben Adressraum (d. h. in derselben Java Virtual Machine). Bild 6.2 zeigt dagegen den Fall für einen *Fern-Methodenauftrag* (deutsche Übersetzung für Remote Method Invocation

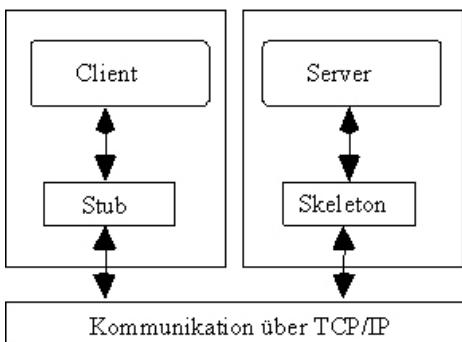
[RMI]), in dem sich die Objekte auf unterschiedlichen Rechnern und damit in unterschiedlichen Adressräumen befinden.



**Bild 6.2**

Fern-Methodeaufruf: Beide Objekte befinden sich in unterschiedlichen Prozessen (und damit in unterschiedlichen Adressräumen bzw. Java Virtual Machines), in der Regel auch auf unterschiedlichen Rechnern

Zur Realisierung der *Verteilungstransparenz* (d.h. zum Verbergen des Aspekts Verteilung vor dem Programmierer) erfolgt der Fern-Methodeaufruf über so genannte *Stubs* und *Skeletons* (s. Bild 6.3).



**Bild 6.3**

Prinzip des Fern-Methodeaufrufs über Stubs und Skeletons

Um die in Bild 6.3 gezeigte, relativ abstrakte Struktur besser zu verstehen, wird das Prinzip von RMI anhand einer Metapher aus dem täglichen Leben erläutert, das jeder von Ihnen kennen dürfte. Stellen Sie sich ein Fernsehgerät vor, das an seinem Gehäuse eine Menge von Druckknöpfen hat zur Programmwahl, zur Regelung der Lautstärke, der Helligkeit usw. Stellen Sie sich weiter eine Fernbedienung vor, die genau dieselben Knöpfe mit denselben Wirkungen und in derselben Anordnung hat (in Wirklichkeit ist dies in der Regel nicht so; lassen Sie uns das aber für den Vergleich einmal annehmen). Die Fernbedienung erlaubt Ihnen also, Ihr Fernsehgerät aus der Ferne so zu bedienen, als würden Sie direkt vor dem Fernseher sitzen und direkt auf die Knöpfe, die am Fernsehergehäuse angebracht sind, drücken. Ähnlich verhält es sich mit RMI. Das Fernsehgerät entspricht dem Server. Der Server enthält ein Objekt, das aus der Ferne benutzt werden kann. Der menschliche Benutzer der Fernbedienung entspricht dem Client, der das Objekt des Servers nutzen möchte, indem Methoden auf dieses Objekt in der Ferne angewendet werden. Die Fernbedienung entspricht dem *Stub* (ausgesprochen: Stabb). Dies ist ein Objekt, das dieselbe Java-Schnittstelle implementiert wie das Objekt, das auf dem Server von der Ferne aus benutzt werden soll (Metapher: dieselben Knöpfe in derselben Anordnung). Das heißt: Mit dem Stub-Objekt kann der Client so umgehen, als wäre es das Objekt, das sich auf dem Server befindet. Der Client kann dieselben Methoden aufrufen. Ähnlich wie bei der Fernbedienung wird aber dadurch nur ein entsprechendes Signal über eine TCP-Verbindung an den Server gesendet und so lange gewartet, bis von der anderen Seite eine Bestätigung über den Abschluss des

Methodenaufrufs eintritt. Dann kehrt die aufgerufene Methode zum Client zurück. Das Empfangsteil am Fernsehgehäuse, das die von der Fernbedienung gesendeten Infrarotsignale empfängt und in Steuerimpulse für das Fernsehgerät umsetzt, entspricht schließlich dem *Skeleton*. Der Skeleton nimmt die Nachrichten, die vom Stub über die aufgebaute TCP-Verbindung gesendet werden, entgegen, interpretiert diese und ruft die entsprechenden Methoden auf dem entsprechenden Server-Objekt auf. Nachdem der Methodenaufruf zu Ende ist, wird eine Erfolgsmeldung zusammen mit dem Rückgabewert der Methode, falls die Methode nicht void ist, an den Stub zurückgesendet oder alternativ die geworfene Ausnahme, falls die Methode durch das Werfen einer Ausnahme vorzeitig beendet wurde.

Der Aspekt *Transparenz* kommt bei RMI nun dadurch zustande, dass der Entwickler nur Client und Server, nicht aber Stub und Skeleton programmieren muss. Der Stub wird für jede Anwendung automatisch erzeugt. Der Skeleton ist ein Programmteil, das in der RMI-Implementierung dabei ist und für alle RMI-Anwendungen verwendet werden kann.

Bei der Entwicklung einer RMI-Anwendung werden wir in den folgenden Beispielen immer in folgenden Schritten vorgehen:

**1. Schnittstelle definieren:** Dies ist die Java-Schnittstelle zwischen Client und Server, die natürlich in Abhängigkeit von der konkreten Anwendung definiert werden muss. Die Schnittstelle führt alle Methoden (mit ihren Parameter- und Rückgabetypen) auf, die über RMI in der Ferne aufgerufen werden können. Bei der Definition der Schnittstelle sind zwei Besonderheiten zu beachten:

- Die Schnittstelle muss aus der Schnittstelle *Remote* aus dem Package `java.rmi` abgeleitet werden.
- Alle Methoden der Schnittstelle müssen mit „throws  *RemoteException*“ gekennzeichnet werden.

**2. Schnittstelle implementieren:** Es wird eine Klasse geschrieben, die die in Schritt 1 definierte Schnittstelle implementiert. Auch dabei gibt es zwei Besonderheiten zu beachten:

- Einem Objekt muss die Benutzung von außen über RMI eingeräumt werden (d. h. das Objekt muss exportiert werden). Dafür gibt es mehrere Möglichkeiten. Eine einfache Lösung besteht darin, dass die Klasse aus der Klasse *UnicastRemoteObject* abgeleitet wird.
- Wenn die implementierende Klasse aus *UnicastRemoteObject* abgeleitet wird, muss es einen expliziten Konstruktor geben, der mit „throws  *RemoteException*“ gekennzeichnet ist. Wird kein Konstruktor angegeben, so gibt es ja bekanntlich den Standardkonstruktor. Da dieser aber nicht mit „throws  *RemoteException*“ markiert ist, würde dies zu einem Syntaxfehler führen.

**3. Server programmieren:** Im 2. Schritt haben wir lediglich eine Klasse geschrieben, aber wir benötigen natürlich Objekte. Deshalb brauchen wir noch den Server selbst. Dieser wird in unseren Beispielen immer nur aus einer kurzen Main-Methode bestehen, in der eines oder mehrere Objekte der Klasse aus Schritt 2 (RMI-Objekte) erzeugt und unter einem frei wählbaren Namen bei einer Art *Auskunftsdiest* angemeldet werden.

**4. Client programmieren:** In diesem Schritt schreiben wir ein Programm, das sich über die Auskunft einen Stub für das angemeldete RMI-Objekt beschafft. Dazu muss der Client den Rechner kennen, auf dem der Server läuft, sowie den Namen, unter dem das Objekt in Schritt 3 vom Server angemeldet wurde. Dieser Stub kann dann so verwendet werden, als

wäre er das RMI-Objekt selbst. Das heißt: Der Client benutzt die Methoden der Schnittstelle aus Schritt 1, wobei dadurch die in Schritt 2 implementierten Methoden auf dem RMI-Objekt des Servers ausgeführt werden.

**5. Anwendung übersetzen und ausführen:** Im letzten Schritt schließlich werden alle Java-Dateien übersetzt. Beim Ausführen des Servers ist darauf zu achten, dass vor dem Starten des Servers die *RMI-Registry* gestartet wird, falls dies nicht aus dem Programm des Servers heraus erfolgt.

In den folgenden Abschnitten entwickeln wir RMI-Beispiele. Das erste Beispiel ist ein einführendes Beispiel. Das zweite Beispiel erläutert die vorhandene Parallelität bei RMI-Methodenaufrufen. Während in den ersten beiden Beispielen die Methoden parameterlos sind oder die Typen der Methoden-Parameter und -Rückgabewerte primitive Datentypen (im konkreten Fall int) sind, beschäftigen sich die nächsten zwei Beispiele mit dem Fall, dass Parameter oder Rückgabewerte Objekte sind. Diese können durch ihren Wert oder ihre Referenz übergeben werden. Zu jedem Fall werden wir ein Beispiel zur Illustration betrachten.

## ■ 6.2 Einführendes RMI-Beispiel

### 6.2.1 Basisprogramm

In diesem Beispiel greifen wir das Zähler-Beispiel wieder auf, das im vorigen Kapitel mit UDP und mit TCP realisiert wurde. Nun realisieren wir es mit RMI und gehen dabei so vor, wie am Ende des vorigen Abschnitts beschrieben wurde.

**1. Schnittstelle definieren:** Wenn wir die in Bild 6.4 dargestellte Kommunikation möglichst genau in RMI übertragen wollen, benötigen wir zwei Methoden mit den Namen increment und reset. Beide sind parameterlos und haben einen Int-Wert als Rückgabewert, denn auch im UDP- und TCP-Beispiel wurde der neue Zählerzustand als Antwort an den Client zurückgeschickt. Wenn wir dann die im vorigen Abschnitt erläuterten Besonderheiten beachten (Ableitung der Schnittstelle aus der Schnittstelle Remote, alle Methoden mit „throws RemoteException“ kennzeichnen), dann sieht unsere RMI-Schnittstelle wie in Listing 6.1 aus:

#### Listing 6.1

```
import java.rmi.*;  
  
public interface Counter extends Remote  
{  
    public int reset() throws RemoteException;  
    public int increment() throws RemoteException;  
}
```

**2. Schnittstelle implementieren:** Die Implementierung der Schnittstelle ist im Prinzip einfach (s. Listing 6.2). Auf drei Besonderheiten sei aber hingewiesen:

- Zum einen leiten wir aus der Klasse UnicastRemoteObject ab. Dies ist eine Möglichkeit (unter anderen), die bewirkt, dass Objekte dieser Klasse exportiert werden und somit über RMI benutzt werden können.
- Zum anderen brauchen wir einen Konstruktor, der mit „throws RemoteException“ gekennzeichnet ist, und den wir daher explizit angeben müssen, auch wenn er keine Anweisungen enthält.
- Schließlich sind die beiden Methoden reset und increment als synchronized gekennzeichnet. Damit sind wir darauf vorbereitet, dass Objekte dieser Klasse von mehreren Clients gleichzeitig benutzt werden können.

### **Listing 6.2**

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class CounterImpl extends UnicastRemoteObject
    implements Counter
{
    private int counter;

    public CounterImpl() throws RemoteException
    {
        /* super(); counter = 0;
        */
    }

    public synchronized int reset() throws RemoteException
    {
        System.out.println("Methode reset wurde aufgerufen");
        counter = 0;
        return counter;
    }

    public synchronized int increment() throws RemoteException
    {
        counter++;
        return counter;
    }
}
```

**3. Server programmieren:** Der Server (s. Listing 6.3) besteht aus einer Main-Methode, in der ein Objekt der Klasse CounterImpl erzeugt und unter dem Namen „Counter“ bei der Auskunft angemeldet wird. Diese Anmeldung erfolgt durch Aufruf der Static-Methode *rebind* der Klasse *Naming*.

### **Listing 6.3**

```
import java.rmi.*;

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

```

        CounterImpl myCounter = new CounterImpl();
        Naming.rebind("Counter", myCounter);
        System.out.println("Zähler-Server ist gestartet");
    }
    catch(Exception e)
    {
        System.out.println("Ausnahme: " + e.getMessage());
        e.printStackTrace();
    }
}
}

```

Obwohl die Main-Methode zu Ende läuft, ist der das Programm ausführende Prozess nicht zu Ende. Dies liegt daran, dass „heimlich“ ein Thread gestartet wird. Dieser Thread ist der Skeleton, der wie unser im vorigen Kapitel selbst entwickelter TCP-Server in einer Schleife auf ankommende TCP-Verbindungen wartet.

**4. Client programmieren:** Als Client benötigen wir ein Beispiel, das die Methoden reset und increment aufruft. Unser Client-Programm (Listing 6.4) gestalten wir so wie die Client-Programme für UDP und TCP des vorigen Kapitels. Das heißt: Der Client wird mit zwei Kommandozeilenargumenten gestartet: das erste Argument ist der Name des Rechners, auf dem der Server läuft, das zweite Argument ist eine Zahl, die angibt, wie oft der Zähler erhöht werden soll. Um die Methoden increment und reset anwenden zu können, benötigt man eine Referenz auf ein entsprechendes Objekt. Diese beschafft man sich über eine Anfrage bei der Auskunft, an der das Objekt vom Server angemeldet wurde. Dazu benutzt man die Static-Methode *lookup* der Klasse *Naming* (diese ist das Gegenstück zu *rebind*). Diese Methode besitzt als Argument einen String, der eine URL-artige Form hat:

```
rmi://<Rechnername>/<Objektname>
```

Die in <>-Klammern angegebenen Bestandteile sind dabei durch die entsprechenden Namen zu ersetzen. Die von *lookup* zurückgelieferte Referenz kann vom Client so benutzt werden, als wäre es eine Referenz auf das Objekt, das sich in der Ferne auf dem Server befindet. Auf dieses Objekt kann dann einmal die Methode *reset* und so oft wie als Kommandozeilenargument angegeben die Methode *increment* aufgerufen werden. Wie schon im vorigen Kapitel wird die Zeit für alle Increment-Aufrufe gemessen und zusammen mit der Durchschnittszeit für einen RMI-Methodenaufruf ausgegeben.

#### Listing 6.4

```

import java.rmi.*;

public class Client
{
    public static void main(String args[])
    {
        if(args.length != 2)
        {
            System.out.println("Notwendige Kommandozeilenargumente:"
                + " <Name des Server-Rechners>"
                + " <Anzahl>");
            return;
        }
    }
}

```

```
try
{
    Counter myCounter = (Counter) Naming.lookup("rmi://" +
                                                + args[0] +
                                                + "/Counter");

    // set counter to initial value of 0
    System.out.println("Zähler wird auf 0 gesetzt.");
    myCounter.reset();
    System.out.println("Nun wird der Zähler erhöht.");

    // calculate start time
    int count = new Integer(args[1]).intValue();
    long startTime = System.currentTimeMillis();

    // increment count times
    int result = 0;
    for(int i = 0; i < count; i++)
    {
        result = myCounter.increment();
    }

    // calculate stop time; print out statistics
    long stopTime = System.currentTimeMillis();
    long duration = stopTime - startTime;
    System.out.println("Gesamtzeit = " + duration
                       + " msec");
    if(count > 0)
    {
        System.out.println("Durchschnittszeit = "
                           + ((duration) / (float) count)
                           + " msec");
    }
    System.out.println("Letzter Zählerstand: " + result);
}
catch(Exception e)
{
    System.out.println("Ausnahme: " + e);
}
```

Beachten Sie den Aufruf von `Naming.lookup` im Client-Programm. Die Methode `lookup` ist so definiert, dass ein Objekt des Typs `Remote` zurückgegeben wird. Da diese Schnittstelle natürlich die von uns definierten Methoden `increment` und `reset` nicht enthält (genauer: `Remote` ist eine leere Schnittstelle, die keine einzige Methode enthält), müssen wir das zurückgegebene Objekt entsprechend casten. Da wir das Objekt danach so benutzen können, als wäre es das Objekt auf dem Server, das der Klasse `CounterImpl` angehört, könnte man eventuell versucht sein, auf `CounterImpl` zu casten. Dies würde aber zu einer `ClassCastException` führen, denn die von `lookup` zurückgegebene Referenz verweist nicht auf ein Objekt des Typs `CounterImpl`, sondern auf das entsprechende Stub-Objekt. Da wir aber den Typ des Stubs in der Regel nicht kennen, casten wir auf die Schnittstelle `Counter`, die natürlich vom Stub implementiert wird. Das Casten gelingt deshalb und erlaubt uns damit die Benutzung der Methoden der Schnittstelle.

**5. Anwendung übersetzen und ausführen:** Beim Erzeugen des Programmcodes musste in früheren Zeiten der Code für die Stub- und Skeleton-Klasse durch Aufruf des RMI-Compilers (Kommando *rmic*) explizit erzeugt werden. Seit der Java-Version 1.2 ist die Erzeugung des Skeletons und seit der Java-Version 5 auch die des Stubs nicht mehr nötig. Wir werden in Abschnitt 6.9 sehen, warum das so ist.

Bevor der Server gestartet wird, muss die Auskunft gestartet werden, bei der der Server das Objekt anmelden wird. Dies geschieht durch folgendes Kommando:

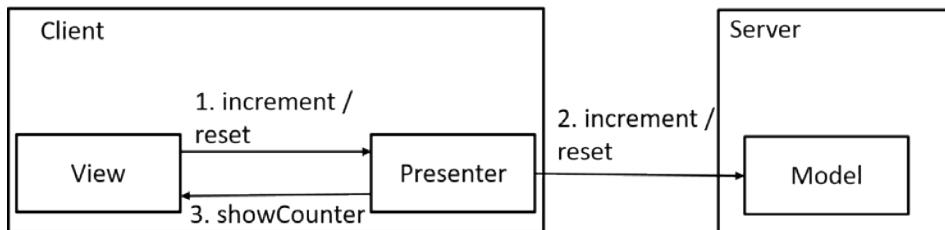
```
rmiregistry
```

Es ist ganz wichtig, Folgendes zu beachten: Wenn Sie dieses Kommando ausführen, dann muss der CLASSPATH so eingestellt sein, dass Sie durch Eingabe des Java-Kommandos auch den Server starten könnten. Dies erreicht man z. B. dadurch, dass im CLASSPATH das aktuelle Verzeichnis(.) enthalten ist, und dass das aktuelle Verzeichnis dasjenige ist, in dem die Pfadstruktur beginnt. Angenommen, der Server würde sich im Package chapter6.example1 befinden, dann muss sich im aktuellen Ordner der Ordner chapter6 befinden (und darin der Ordner example1, und darin die Class-Dateien Counter.class, CounterImpl.class und Server.class). Das Programm rmiregistry erzeugt nach dem Start keine Ausgabe und läuft in einer Endlosschleife. Das heißt: Es muss manuell z. B. durch Eingabe von Strg-C abgebrochen werden.

Sollten Sie Client und Server auf unterschiedlichen Rechnern ausführen wollen, so genügt es, die Auskunft RMI-Registry auf dem Rechner zu starten, auf dem der Server läuft; auf dem Client-Rechner ist keine RMI-Registry nötig.

## 6.2.2 RMI-Client mit grafischer Benutzeroberfläche

Damit liegt uns die erste lauffähige RMI-Anwendung vor. Wie schon die Client-Programme für die Socket-Beispiele hat auch dieser RMI-Client keine grafische Benutzeroberfläche. Dies ist heute in der Regel eher unüblich. Deshalb erweitern wir den Client aus Listing 6.4 um eine grafische Benutzeroberfläche, was natürlich auch bei den Socket-Beispielen möglich gewesen wäre. Eine lokale Zähleranwendung wurde bereits in Listing 4.3 vorgestellt. Das von dem Programm erzeugte Fenster war in Bild 4.2 bzw. mit etwas „Luft“ um die Interaktionselemente in Bild 4.5 zu sehen und soll für die verteilte Anwendung genauso aussehen. Das Zähler-Beispiel mit JavaFX wurde vorgestellt, bevor das MVP-Architekturmuster erläutert wurde und war deshalb noch nicht nach diesem Muster „gestrickt“. Es ist aber sehr naheliegend, daraus eine MVP-Anwendung zu machen, die in diesem Fall sogar noch verteilt ist. Die Modellkomponente befindet sich nun auf dem Server, während der Client die Komponenten für die Darstellung (View) und für die Ablaufsteuerung (Presenter) enthält (s. Bild 6.4).



**Bild 6.4** Prinzip der verteilten MVP-Zähler-Anwendung

Die Modellkomponente besteht aus der Zählerschnittstelle (Listing 6.1), deren Implementierung (Listing 6.2) und dem Server (Listing 6.3); sie liegt bereits fertig vor und muss nicht geändert werden. Die Darstellungskomponente baut wie üblich die Oberfläche auf und besitzt eine Methode namens `showCounter` zum Anzeigen des geänderten Zählerwerts. Je nachdem, welcher Button gedrückt wurde, wird die Methode `increment` oder `reset` des Presenters aus der View heraus aufgerufen. Die Implementierung dieser Presenter-Methoden wäre im lokalen Fall sehr einfach. Nehmen Sie an, dass wie üblich `model` und `view` Attribute der Presenter-Klasse sind. Dann würde beispielsweise die Methode `increment` wie folgt aussehen (`reset` analog):

```

public void increment()
{
    int c = model.increment();
    view.showCounter(c);
}

```

Nun haben wir aber das Problem, dass die Ausführung von `model.increment` eine Kommunikation mit einem Server beinhaltet und somit unter Umständen länger dauern kann. Also müssen wir den Inhalt der Methode `increment` in einen Thread auslagern und darin den Zugriff auf die View über `Platform.runLater` durchführen. Das ist sehr einfach zu programmieren. Da dies für `increment` und `reset` sehr ähnlich ist, wollen wir die Aufgabe etwas allgemeingültiger lösen. Wir stellen uns vor, dass wir eine allgemeine Methode `asyncCall` hätten, in der wir als ersten Parameter den Code angeben könnten, der länger dauert (hier in diesem Fall der RMI-Aufruf) und als zweiten Parameter den Code, der nach dem RMI-Aufruf über `Platform.runLater` durchgeführt werden soll (in diesem Fall `view.showCounter`). Der Rückgabewert des RMI-Aufrufs soll dabei als Parameter in die Aktion, die über `Platform.runLater` erfolgt, einfließen. Wenn soeben vom Angeben von Code die Rede war, dann soll das bedeuten, dass man dies über Lambda-Ausdrücke realisieren kann. Wir wünschen uns also, dass wir die Methode `increment` so schreiben können:

```

public void increment()
{
    asyncCall(()->model.increment(), (c)->view.showCounter(c));
}

```

Damit ein solcher Aufruf von `asyncCall` möglich ist, muss `asyncCall` ganz offensichtlich zwei Parameter besitzen, deren Typ jeweils eine funktionale Schnittstelle ist. Die erste Schnittstelle muss eine parameterlose Methode haben und sollte etwas zurückgeben (am besten man wählt dafür einen generischen Typ für die Rückgabe). Eine solche Schnittstelle

gibt es bereits in Form der Schnittstelle Supplier im Package java.util.function. Wenn wir diese Schnittstelle verwenden würden, hätten wir allerdings das Problem, dass der Aufruf von model.increment nicht passt, da model.increment eine RemoteException werfen kann. Deshalb wird im folgenden Client-Programm eine eigene Schnittstelle namens RMISupplier definiert, deren Methode mit throws RemoteException versehen ist. Für den zweiten Parameter können wir die vorhandene Schnittstelle Consumer aus dem Package java.util.function benutzen, die eine Methode ohne Rückgabewert und mit einem Argument eines generischen Typs besitzt:

```
public interface Consumer<T>
{
    public void accept(T t);
}
```

Mit diesen beiden Schnittstellen RMISupplier und Consumer lässt sich dann die Methode asyncCall implementieren. In Listing 6.5 findet sich der komplette Code für einen JavaFX-Zähler-Client, der im Wesentlichen aus View und Presenter besteht einschließlich der Implementierung von asyncCall.

#### **Listing 6.5**

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.List;
import java.util.function.Consumer;
import javafx.application.*;
import javafx.geometry.Insets;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;

class ClientGuiView
{
    private ClientGuiPresenter presenter;
    private VBox root;
    private Label label;

    public ClientGuiView(ClientGuiPresenter presenter)
    {
        this.presenter = presenter;
        initView();
    }

    private void initView()
    {
        root = new VBox(10);
        Insets ins = new Insets(10);
        root.setPadding(ins);
        label = new Label();
        label.setMaxWidth(Double.MAX_VALUE);
        root.getChildren().add(label);
        Button bInc = new Button("Erhöhen");
        bInc.setMaxWidth(Double.MAX_VALUE);
        bInc.setOnAction(e -> presenter.increment());
    }
}
```

```
root.getChildren().add(bInc);
Button bReset = new Button("Zurücksetzen");
bReset.setMaxWidth(Double.MAX_VALUE);
bReset.setOnAction(e -> presenter.reset());
root.getChildren().add(bReset);
}

public Pane getUI()
{
    return root;
}

public void showCounter(int counter)
{
    label.setText("" + counter);
}
}

interface RMISupplier<T>
{
    public T execute() throws RemoteException;
}

class ClientGuiPresenter
{
    private Counter model;
    private ClientGuiView view;

    public void setModelAndView(Counter model, ClientGuiView view)
    {
        this.model = model;
        this.view = view;
    }

    public void increment()
    {
        asyncCall(() -> model.increment(), (c) -> view.showCounter(c));
    }

    public void reset()
    {
        asyncCall(() -> model.reset(), (c) -> view.showCounter(c));
    }

    private <T> void asyncCall(RMISupplier<T> rmiCall, Consumer<T> fxCall)
    {
        new Thread(() -> doInThread(rmiCall, fxCall)).start();
    }

    private <T> void doInThread(RMISupplier<T> rmiCall, Consumer<T> fxCall)
    {
        try
        {
            T t = rmiCall.execute();
            Platform.runLater(() -> fxCall.accept(t));
        }
        catch(RemoteException e)
        {
        }
    }
}
```

```

        System.err.println("Es gibt Probleme mit RMI!");
    }
}
}

public class ClientGui extends Application
{
    public void start(Stage primaryStage)
    {
        List<String> args = getParameters().getUnnamed();
        Counter m = null;
        try
        {
            m = (Counter) Naming.lookup("rmi://" +
                args.get(0) + "/" +
                args.get(1));
        }
        catch(Exception e)
        {
            System.out.println("Verbindung zu Server nicht möglich");
            Platform.exit();
        }

        ClientGuiPresenter p = new ClientGuiPresenter();
        ClientGuiView v = new ClientGuiView(p);
        p.setModelAndView(m, v);

        Scene scene = new Scene(v.getUI());
        primaryStage.setScene(scene);
        primaryStage.setTitle("RMI-Zähler");
        primaryStage.show();
    }

    public static void main(String[] args)
    {
        if(args.length != 2)
        {
            System.out.println("Argumente: Server RMI-Objekt");
            Platform.exit();
            return;
        }
        launch(args);
    }
}
}

```

Wie in In Listing 6.5 zu sehen ist, veranlasst `asyncCall`, dass die Methode `doInThread` in einem Thread ausgeführt wird. Darin findet der RMI-Aufruf statt. Der Rückgabewert des RMI-Aufrufs wird dann dem Methodenaufruf übergeben, der mit `Platform.runLater` ausgeführt wird. Sollte während des RMI-Aufrufs etwas schiefgehen, dann gibt `doInThread` fest codiert eine einfache Fehlermeldung auf `System.err` aus. Die Aktion, die im Falle einer Ausnahme passieren soll, hätte man als dritten Parameter in `asyncCall` angeben können, wodurch `asyncCall` dann noch etwas allgemeingültiger geworden wäre.

Im Client-Code ist nicht vorgegeben, auf welchem Rechner der Server läuft und unter welchem Namen das RMI-Objekt registriert wurde. Dies wird erst über Kommandozeilenargumente beim Starten des Client-Programms festgelegt. Die Kommandozeilenargumente wer-

den an main bekanntlich in Form eines String-Arrays übergeben und beim Aufruf von lauch weitergegeben. In der Methode start kann man an diese Kommandozeilenargumente über getParameters().getUnnamed() in Form einer String-Liste herankommen. Die Elemente der Liste werden dann für den Aufruf von Naming.lookup verwendet. Wenn man statt des Aufrufs von Naming.lookup lokal ein Counter-Objekt erzeugen würde, dann hätte man eine lokale Variante des Zähler-Beispiels in MVP-Form.

Abschließend wollen wir noch auf einen kleinen subtilen Punkt hinweisen, der einmal mehr zeigt, wie tückisch Parallelität sein kann. Zwar wird man nicht benachrichtigt, wenn der Zähler von einem anderen Client verändert wird, aber man wird vermutlich davon ausgehen, dass der angezeigte Zählerwert immer derjenige ist, der aus der eigenen zuletzt angestoßenen Aktion resultiert. Aber dies muss nicht in jedem Fall so sein. Wenn wir annehmen, dass zwei (oder mehr) Button-Klicks sehr schnell hintereinander ausgeführt werden, dann wird jedes Mal ein neuer Thread gestartet. Natürlich können diese unterschiedlich schnell laufen. So kann es passieren, dass ein zuerst gestarteter Thread den Aufruf von Platform.runLater als Letzter durchführt. Damit wird im Fenster ein Zählerstand angezeigt, der nicht aus der zuletzt angestoßenen Aktion herröhrt. Für den Anwender könnte es somit irritierend sein, wenn beispielsweise der Erhöhen- und dann der Zurücksetzen-Button schnell hintereinander geklickt werden, am Ende aber nicht 0 angezeigt wird. In diesem Fall wäre das Problem dadurch zu lösen, dass alle Hintergrundaktionen durch einen einzigen Thread ausgeführt werden, der alle Aufträge der Reihe nach abarbeitet.

### 6.2.3 RMI-Registry

Schauen wir uns im Folgenden die im Client- und Server-Programm benutzte Klasse Naming, die einen Zugriff auf die RMI-Registry bietet, noch etwas genauer an:

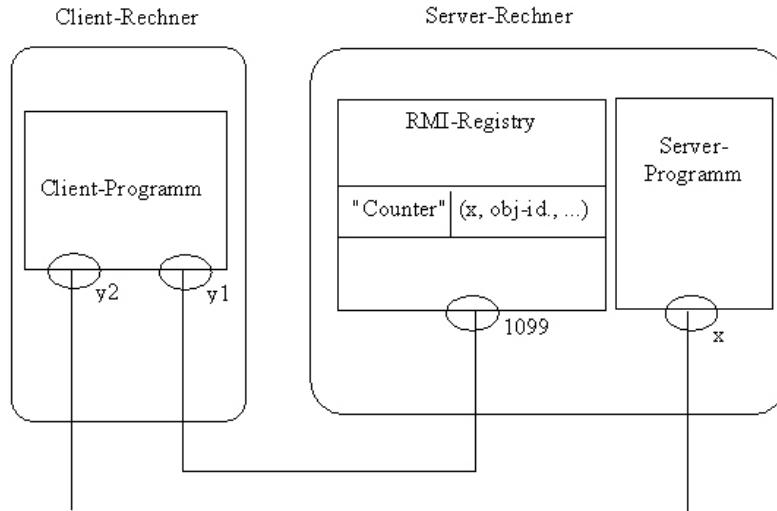
```
public final class Naming
{
    public static void bind(String name, Remote obj)
        throws AlreadyBoundException, MalformedURLException,
               RemoteException
    {...}

    public static void rebind(String name, Remote obj)
        throws MalformedURLException, RemoteException
    {...}
    public static void unbind(String name)
        throws NotBoundException, MalformedURLException,
               RemoteException
    {...}
    public static Remote lookup(String name)
        throws NotBoundException, MalformedURLException,
               RemoteException
    {...}
    public static String[] list(String name)
        throws MalformedURLException, RemoteException
    {...}
}
```

Neben den bereits behandelten Static-Methoden rebind und lookup besitzt die Klasse Naming drei weitere Static-Methoden. Eine davon ist *bind*. Diese ist sehr ähnlich wie rebind. Es wird allerdings die Ausnahme *AlreadyBoundException* ausgelöst, falls in der Auskunft RMI-Registry bereits ein Eintrag mit diesem Namen existiert. Wir haben in unserer Anwendung allerdings rebind verwendet, wodurch Einträge mit demselben Namen überschrieben werden. Der Grund für die Verwendung von rebind ist: Wenn man eine RMI-Anwendung entwickelt, dann wird es häufiger kommen, dass man den Server abbricht (z.B. mit Strg-C), am Programmcode etwas ändert, neu übersetzt und neu startet. Würde man dabei bind verwenden, so müsste man auch die RMI-Registry abbrechen und neu starten. Verwendet man aber rebind, so hat dies den Vorteil, dass die RMI-Registry nicht neu gestartet werden muss, denn der Server überschreibt einfach seinen Eintrag vom vorhergehenden Lauf. Eine weitere Static-Methode der Klasse Naming ist *unbind*, mit der ein Eintrag wieder aus der RMI-Registry gelöscht werden kann. Schließlich gibt es noch die Methode *list*, die ein String-Feld zurückgibt, das alle Namen aller aktuell vorhandenen Einträge der RMI-Registry enthält. Die Methoden bind, rebind und unbind werden typischerweise vom Server aufgerufen, wobei sich die RMI-Registry auf demselben Rechner befinden muss wie der Aufrufer dieser Methoden. Die Methoden lookup und list werden in der Regel vom Client aufgerufen. Der Client kann auf einem anderen Rechner laufen.

Die Kommunikation zwischen Client und RMI-Registry, Server und RMI-Registry sowie Client und Server erfolgt über TCP mit Sockets. Die Frage dabei ist, welche Portnummern verwendet werden und wie der Client von den verwendeten Portnummern erfährt.

In Bild 6.5 ist die Antwort auf diese Frage dargestellt. Der Server verwendet eine beliebige Portnummer, die in der Abbildung mit x bezeichnet wird. Beim Anmelden des RMI-Objekts an der RMI-Registry werden dem Namen (in unserem Beispiel Counter) Informationen zugeordnet, mit denen man das entsprechende Objekt erreichen kann. Diese Informationen bestehen u.a. aus der Portnummer und einer Kennung obj-id für das Objekt (der Server kann mehrere Objekte beherbergen, die alle über den Port x erreicht werden können). Der Client ruft die Methode Naming.lookup auf. Aus dem angegebenen String kann der Name des Rechners extrahiert werden. Da die RMI-Registry auf einem Socket mit wohl bekannter Portnummer, nämlich 1099, lauscht, kann der Client damit eine TCP-Verbindung zur RMI-Registry des Servers herstellen. Dort wird eine Anfrage nach dem Namen des Objekts (also „Counter“) gestellt. Der Client erhält die diesem Objekt zugeordneten Erreichbarkeitsinformationen und kann nun im zweiten Schritt eine Verbindung aufbauen zu dem Rechner, von dem er diese Information bekommen hat, und zu der in diesen Informationen enthaltenen Portnummer. Über die aufgebaute Verbindung sendet er die Kennung für das Objekt, damit der Server weiß, mit welchem Objekt der Client arbeiten möchte. Wie üblich verwendet der Client stets beliebige Portnummern (in Bild 6.5 mit y1 und y2 bezeichnet).



**Bild 6.5** Kommunikation zwischen Client und Server bei RMI

Falls die Portnummer 1099 aus irgendeinem Grund nicht für die RMI-Registry verwendet werden kann oder soll, so kann die RMI-Registry auch einen anderen Port benutzen. Dazu gibt man beim Start der RMI-Registry die Portnummer als zusätzliches Kommandozeilenargument an:

```
rmiregistry 5000
```

In diesem Fall lauscht die RMI-Registry auf die Portnummer 5000. Ähnlich wie bei HTTP muss aber dann in diesem Fall die Portnummer der RMI-Registry auf Client-Seite angegeben werden. Der in Naming.lookup angegebene String hat im Allgemeinen folgende Struktur:

```
rmi://<Rechnername>:<Portnummer der RMI-Registry>/<Objektname>
```

In unserem Client-Programm müsste dann in diesem konkreten Fall die Naming.lookup-Anweisung so aussehen (Änderung fett hervorgehoben):

```
Counter myCounter = (Counter) Naming.lookup("rmi://" +
+ args[0] +
+ ":5000/Counter");
```

Die RMI-Registry kann übrigens auch aus dem Programm heraus gestartet werden. Dazu wird aus dem Package `java.rmi.registry` die Static-Methode `createRegistry` der Klasse `LocateRegistry` benötigt:

```
public final class LocateRegistry
{
    public static Registry createRegistry(int port)
        throws RemoteException
    {...}
    ...
}
```

Das Argument gibt die Portnummer an. Will man den Standardport verwenden, muss die Methode mit dem Argument 1099 aufgerufen werden. Der Rückgabetyp *Registry* ist eine Schnittstelle und besitzt genau dieselben Methoden wie die Klasse Naming (Naming kann aber Registry nicht implementieren, da die Methoden von Naming alle static sind). Man kann wahlweise die von *createRegistry* zurückgelieferte Referenz benutzen, um auf die RMI-Registry zuzugreifen, oder man benutzt weiterhin die Static-Methoden aus der Klasse Naming. Eine dritte Alternative besteht darin, dass man sich eine Referenz auf eine lokale oder ferne RMI-Registry mit Hilfe einer der überladenen Methoden *getRegistry* der Klasse *LocateRegistry* beschafft.

```
public final class LocateRegistry
{
    //registry on local host, port 1099
    public static Registry getRegistry()
        throws RemoteException
    {...}

    //registry on local host, specified port
    public static Registry getRegistry(int port)
        throws RemoteException
    {...}

    //registry on specified host, port 1099
    public static Registry getRegistry(String host)
        throws RemoteException
    {...}

    //registry on specified host, specified port
    public static Registry getRegistry(String host, int port)
        throws RemoteException
    {...}
    ...
}
```

Typischerweise würde man das Starten einer Registry in das Programm eines Servers einbauen (der Client benötigt ja keine Registry). In Listing 6.6 wird diejenige Alternative gezeigt, bei der die von *createRegistry* zurückgegebene Referenz benutzt wird, um die Methode *rebind* aufzurufen (Änderungen gegenüber Listing 6.3 sind fett gedruckt).

#### **Listing 6.6**

```
import java.rmi.registry.*;

public class Server
{
    public static void main(String args[])
    {
        try
        {
            Registry registry = LocateRegistry.createRegistry(1099);
            registry.rebind("Counter", new CounterImpl());
            System.out.println("Zähler-Server ist gestartet");
        }
        catch(Exception e)
        {
```

```
        System.out.println("Ausnahme: " + e.getMessage());
        e.printStackTrace();
    }
}
```

Beachten Sie, dass Sie nicht mehrere Server gleichzeitig ausführen können, die alle eine Registry auf demselben Port starten. Auch ist es nicht möglich, das Kommando rmiregistry auszuführen und während dieser Zeit eine Registry auf demselben Port aus einem Programm heraus zu starten. Sie dürfen die Registry entweder nur in einem der Server starten, oder Sie müssen für jedes Starten einer Registry eine neue, noch nicht benutzte Portnummer verwenden.

## ■ 6.3 Parallelität bei RMI-Methodenaufrufen

In diesem Abschnitt wollen wir erforschen, was die gleichzeitige Nutzung eines RMI-Objekts durch mehrere Clients bedeutet und worauf man dabei zu achten hat. Um die gleichzeitige Nutzung besser beobachten zu können, verwenden wir eine RMI-Methode, deren Ausführung längere Zeit dauern soll. Wie in Abschnitt 5.6.1 zur Demonstration des Unterschieds zwischen einem sequenziellen und einem parallelen TCP-Server verwenden wir Thread.sleep.

**1. Schnittstelle definieren:** Die Schnittstelle besteht in diesem Fall lediglich aus einer Void-Methode namens sleep, der eine Int-Zahl als Wert übergeben wird (s. Listing 6.7). Diese Zahl gibt an, wie viele Sekunden lang der Methodenaufruf dauern soll:

### Listing 6.7

```
public interface Sleep extends java.rmi.Remote
{
    public void sleep(int secs) throws java.rmi.RemoteException;
}
```

**2. Schnittstelle implementieren:** Die Implementierung der Schnittstelle (s. Listing 6.8) wird mit Thread.sleep realisiert:

### Listing 6.8

```
import java.rmi.*;
import java.rmi.server.*;

public class SleepImpl extends UnicastRemoteObject implements Sleep
{
    public SleepImpl() throws RemoteException
    {
    }

    public synchronized void sleep(int secs) throws RemoteException
    {
        System.out.println("Beginn von sleep(" + secs + ")");
        try
        {
            Thread.sleep(secs * 1000);
        }
        catch (InterruptedException e)
        {
            System.out.println("Fehler beim Warten: " + e.getMessage());
        }
    }
}
```

```

        try
        {
            Thread.sleep(secs * 1000);
        }
        catch(InterruptedException e)
        {
            System.out.println(e);
        }
        System.out.println("Ende von sleep(" + secs + ")");
    }
}

```

Die Methode sleep müsste nicht synchronized sein. Wir werden in diesem Abschnitt aber nochmals demonstrieren, welchen Effekt synchronized hat, indem wir das Programm mit und ohne synchronized ausführen.

**3. Server programmieren:** Der Server erzeugt in diesem Fall zwei Objekte der Klasse Sleep-Impl und meldet diese unter unterschiedlichen Namen bei der RMI-Registry an (s. Listing 6.9). Auch dies erfolgt nur zu Demonstrationszwecken. Wir wollen zeigen, welchen Unterschied es macht, wenn mehrere Clients gleichzeitig auf dasselbe oder auf unterschiedliche RMI-Objekte zugreifen:

#### Listing 6.9

```

import java.rmi.*;

public class Server
{
    public static void main(String[] args)
    {
        try
        {
            SleepImpl server;
            server = new SleepImpl();
            Naming.rebind("SleepServer1", server);
            server = new SleepImpl();
            Naming.rebind("SleepServer2", server);
        }
        catch(Exception e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
    }
}

```

**4. Client programmieren:** Der Anwender soll wählen können, mit welchem RMI-Objekt das Client-Programm arbeiten soll. Deshalb muss nach dem Namen des Server-Rechners als weiteres Kommandozeilenargument der Name des RMI-Objekts angegeben werden. Danach können beliebig viele weitere Kommandozeilenargumente folgen. Diese werden als Zahlen interpretiert, die der Reihe nach als Argumente bei wiederholtem Aufruf der Sleep-Methode übergeben werden (s. Listing 6.10):

**Listing 6.10**

```

import java.rmi.*;

public class Client
{
    public static void main(String[] args)
    {
        if(args.length < 3)
        {
            System.out.println("Benötigte Kommandozeilenargumente:"
                + " <Name des Servers>"
                + " <Name des Objekts>"
                + " <Sekunden 1> ... <Sekunden N>");
            return;
        }

        try
        {
            Sleep server = (Sleep) Naming.lookup("rmi://" + args[0]
                + "/"
                + args[1]);
            System.out.println("Kontakt zu Server hergestellt");
            for(int i = 2; i < args.length; i++)
            {
                int secs = Integer.parseInt(args[i]);
                System.out.println("Aufruf von sleep(" + args[i]
                    + ")");
                server.sleep(secs);
            }
        }
        catch(Exception e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
    }
}

```

**5. Anwendung übersetzen und ausführen:** Bei der Ausführung dieser Anwendung stellen sich interessante Effekte nur dann ein, wenn wir mindestens zwei Clients gleichzeitig starten. Angenommen, der erste Client wird mit der Argumentliste „localhost SleepServer1 60“ gestartet. So lange dieser Client noch läuft (also innerhalb einer Minute), sollte es uns nun gelingen, einen zweiten Client zu starten, der mit dem anderen RMI-Objekt arbeitet (also z. B. mit der Argumentliste „localhost SleepServer2 5“). Wir beobachten, dass der zweite Client nach 5 Sekunden zu Ende läuft. Der erste Client kann zu diesem Zeitpunkt immer noch aktiv sein. Dies bedeutet, dass der zweite Client nicht warten musste, bis der erste Client bedient war. Dies wird auch deutlich, wenn wir die Ausgabe des Sleep-Servers betrachten:

```

Beginn von sleep(60)
Beginn von sleep(5)
Ende von sleep(5)
Ende von sleep(60)

```

Auch an dieser Ausgabe erkennt man, dass ein zweiter Aufruf von sleep erfolgte, bevor der erste zu Ende war. Offenbar werden die Methodenaufrufe der beiden Clients in unterschiedlichen Threads abgewickelt. Das heißt, dass wir zur parallelen Nutzung des RMI-Servers durch mehrere Clients nichts dazu programmieren müssen, sondern dass diese Möglichkeit in RMI bereits „eingebaut“ ist. Anders als beim parallelen TCP-Server müssen wir in unserer Anwendung also keine Threads erzeugen und starten, da dies bereits automatisch gemacht wird. Dieser Komfort beinhaltet aber auch Gefahren, denn dadurch kann unter Umständen übersehen werden, dass man es hier mit Parallelität zu tun hat. Wie aus Kapitel 2 bekannt ist, kann die Ausführung mit synchronized sequenzialisiert werden, wenn mehrere Threads auf demselben Objekt arbeiten. Wir wiederholen das Starten zweier Clients, wobei der erste Client wie zuvor gestartet wird, während der zweite Client nun auch mit dem Argument „SleepServer1“ statt „SleepServer2“ aufgerufen wird. Falls die Sleep-Methode der Klasse SleepImpl wie in Listing 6.8 mit synchronized gekennzeichnet ist, dann ist das Verhalten dieses Mal deutlich anders. Auf das Ende des zweiten Clients muss nun länger gewartet werden; erst 5 Sekunden nach dem Ende des ersten Clients ist der zweite Client zu Ende. Dies zeigt, dass die Sleep-Aufrufe dieses Mal nicht parallel, sondern hintereinander ausgeführt wurden. Dies zeigt auch ein Blick auf die Ausgabe des Servers:

```
Beginn von sleep(60)
Ende von sleep(60)
Beginn von sleep(5)
Ende von sleep(5)
```

Wird das Schlüsselwort synchronized in der Klasse SleepImpl entfernt und werden die beiden Experimente wiederholt, so ist das Verhalten jetzt in beiden Fällen so wie bei unserem ersten Experiment. Da es keinerlei Einschränkungen der Parallelität gibt, werden die Sleep-Aufrufe parallel abgearbeitet.

Durch Veränderung dieses Beispiels kann man übrigens noch weitere Eigenschaften von RMI erforschen:

- Identität und Gleichheit von Stubs: Führt man in einem RMI-Client wie z. B. dem in Listing 6.10 zwei Mal Naming.lookup mit exakt demselben String als Argument aus, so erhält man zwei unterschiedliche Stub-Objekte für dasselbe RMI-Objekt (d. h. die beiden Stub-Objekte sind nicht identisch – ein Vergleich der beiden Stub-Objekte mit == ergibt false). Wenn wir wieder unsere Metapher vom Beginn des Kapitels zur Hand nehmen, dann entsprechen die Stubs zwei unterschiedliche Fernbedienungen, mit denen sich aber derselbe Fernseher bedienen lässt. Vergleicht man die beiden Stub-Objekte jedoch mit Hilfe der Methode equals, so erhält man true (d. h. die beiden Stub-Objekte sind gleich). Nimmt man dagegen zwei Stubs, die auf unterschiedliche RMI-Objekte zeigen, ergibt ein Vergleich mit equals den Rückgabewert false. Daran kann man erkennen, dass die Methode equals für Stubs offenbar so definiert ist, dass sie genau dann true zurückgibt, wenn die beiden verglichenen Stubs auf dasselbe RMI-Objekt zeigen.
- Parallele Nutzung eines Stubs: Hat man sich in einem Client unterschiedliche Stubs für dasselbe oder unterschiedliche RMI-Objekte besorgt, so ist die Situation bei paralleler Nutzung dieser Stubs nicht anders, als wenn diese unterschiedlichen Stub-Objekte in unterschiedlichen Client-Instanzen parallel genutzt werden. Die spannende Frage ist nun, was passiert, wenn ein Stub-Objekt in einer Client-Instanz durch mehrere Threads parallel benutzt wird. Dies kann durch Variation des RMI-Clients aus Listing 6.10 einfach

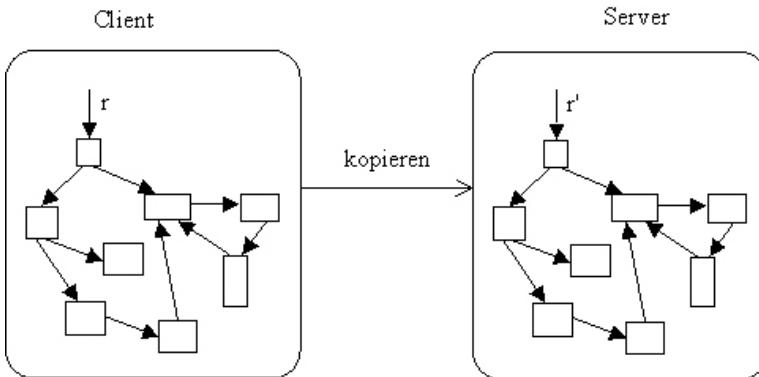
ausprobiert werden. Dazu sollte aber die Methode sleep in Listing 6.8 nicht mehr synchronized sein. Es zeigt sich, dass sich der Stub „vernünftig“, sogar „schlau“ verhält. Wird er nämlich sequenziell benutzt, so baut er nur eine Verbindung zum Server auf. Wird er dagegen parallel benutzt, so baut er mehrere Verbindungen zum Server auf, wodurch auf Server-Seite mehrere Threads erzeugt werden. Im konkreten Fall folgt daraus, dass die Methode sleep auf dem Server von den parallel laufenden Threads einer Client-Instanz auch über einen einzigen Stub parallel aufgerufen werden kann.

In den ersten beiden RMI-Beispielen dieses und des vorigen Abschnitts wurden Werte des Typs int als Parameter vom Client zum Server und als Rückgabewerte vom Server zum Client übertragen. Wie in Java üblich erfolgt die Übergabe von Int-Werten als Parameter oder als Rückgabewerte in Form einer Kopie („Call by Value“/„Return by Value“). Das heißt, dass eine Änderung des Parameters durch den Server keine Auswirkungen hat auf die Variable des Clients, die beim Methodenaufruf übergeben wurde, da der Server mit einer Kopie arbeitet. Gleiches gilt für die anderen primitiven Datentypen boolean, char, short, long, float oder double. Objektparameeter oder von Methoden zurückgegebene Objekte werden in Java dagegen immer in Form von Referenzen auf Objekte („Call by Reference“/„Return by Reference“) realisiert. Eine Besonderheit von RMI ist, dass auch Objektparameeter oder -rückgabewerte als Wert oder als Referenz übergeben werden können. Für beide Fälle folgt jeweils ein Beispiel in den beiden folgenden Abschnitten.

## ■ 6.4 Wertübergabe für Parameter und Rückgabewerte

Wie am Ende des vorigen Abschnitts beschrieben wurde, besitzt RMI die Besonderheit, dass Objektparameeter oder Objektrückgabewerte als Wert oder als Referenz übergeben werden können. In diesem Abschnitt entwickeln wir ein Beispiel für die *Wertübergabe* („Call by Value“).

In Bild 6.6 wird die Wertübergabe grafisch veranschaulicht. Angenommen, es wird eine RMI-Methode vom Client auf dem Server aufgerufen und dabei als Parameter die Referenz r übergeben. Falls Wertübergabe vorliegt, wird eine Kopie des Objekts, auf das die Referenz r zeigt, vom Client auf den Server übertragen. Enthält das Objekt dabei Referenzen auf weitere Objekte, so wird der Kopiervorgang rekursiv fortgesetzt, bis alle Objekte, die man über die Referenz r erreichen kann, auf den Server übertragen wurden und auf dem Server eine entsprechende Verzeigerung der Kopien wiederhergestellt wurde. Die RMI-Methode wird dann auf dem Server mit der Referenz r' aufgerufen. Die RMI-Methode arbeitet also mit einer kompletten Kopie. Entsprechendes gilt für von Methoden zurückgegebene Objekte, die als Werte übertragen werden.



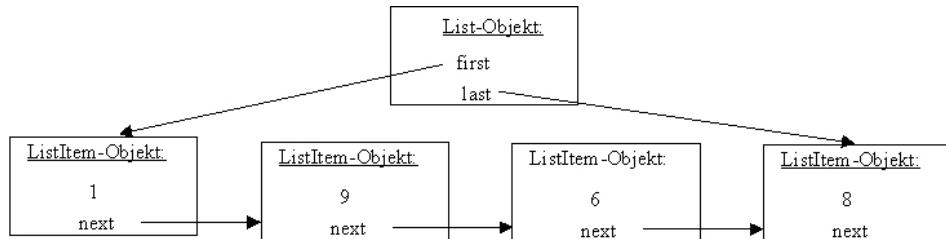
**Bild 6.6** Wertübergabe von Objektparametern bei RMI

Eine Übergabe eines Parameters als Wert ist z. B. dann sinnvoll, wenn ein Client eine Datenstruktur aufgebaut hat und diese zur Auswertung oder Speicherung einem Server übergeben möchte, also bei rein lesender Verwendung des Parameters. Als Werte können beliebig komplexe Datenstrukturen übergeben werden. Wir verwenden im Folgenden als Beispiel lediglich eine simple und überschaubare Datenstruktur, nämlich eine einfache verkettete Liste.

#### 6.4.1 Serialisierung und Deserialisierung von Objekten

Es mag auf den ersten Blick vielleicht sehr kompliziert erscheinen, das Kopieren einer Datenstruktur wie in Bild 6.6 zu programmieren. Glücklicherweise müssen wir dazu aber (fast) nichts tun, denn das Kopieren von solchen komplexen Datenstrukturen ist in Java schon „eingebaut“. Es handelt sich dabei um das Konzept der *Serialisierung*. Mit Serialisierung ist gemeint, dass eine Datenstruktur, die wie in Bild 6.6 zweidimensional dargestellt ist, „flach gedrückt“ und in eine Folge von Bytes (Serie von Bytes) gewandelt wird. Ebenso kann dann auf einfache Weise aus dieser Bytefolge eine exakte Kopie der serialisierten Datenstruktur hergestellt werden. Diesen Vorgang bezeichnet man als *Deserialisierung*. Damit können aber nicht nur Kopien von Datenstrukturen über das Netz übertragen werden, sondern z. B. auch in eine Datei abgespeichert und zu einem späteren Zeitpunkt wieder geladen werden. Damit dies funktioniert, muss der Programmierer lediglich bei allen Klassen der Objekte, die serialisiert werden sollen, die Schnittstelle *Serializable* aus dem Package `java.io` implementieren. Da diese Schnittstelle leer ist (d. h. keine Methoden enthält), ist außer der Kennzeichnung der Klassen durch „implements Serializable“ nichts zu tun.

Im Folgenden entwickeln wir eine einfach verkettete Liste mit Int-Werten, die als Ganzes *serialisierbar* sein soll. Eine Liste besteht aus einem Anker-Objekt der Klasse `List`, das auf das erste und letzte Element der Liste vom Typ `ListItem` zeigt. Ein Beispiel einer solchen Liste zeigt Bild 6.7.



**Bild 6.7** Beispiel einer Liste

Um die Liste *serialisieren* zu können, genügt es nicht, lediglich die Klasse List serialisierbar zu machen. Da Objekte der Klasse nämlich Referenzen auf Objekte der Klasse ListItem haben, muss die Klasse ListItem ebenfalls serialisierbar sein. Entsprechend sind im folgenden Programmtext in Listing 6.11 beide Klassen List und ListItem mit „implements Serializable“ gekennzeichnet. Als Methoden für die Klasse List sehen wir lediglich eine Methode append zum Anhängen eines Elements an das Ende der Liste sowie die Methode print zur Ausgabe der Liste auf dem Bildschirm vor:

#### **Listing 6.11**

```

import java.io.Serializable;

class ListItem implements Serializable
{
    private int value;
    private ListItem next;

    public ListItem(int v)
    {
        value = v;
        next = null;
    }

    public int getValue()
    {
        return value;
    }

    public ListItem getNext()
    {
        return next;
    }

    public void setNext(ListItem n)
    {
        next = n;
    }
}

public class List implements Serializable
{
    private ListItem first, last;

    public void append(int i)
}
  
```

```

    {
        if(first == null)
        {
            first = new ListItem(i);
            last = first;
        }
        else
        {
            last.setNext(new ListItem(i));
            last = last.getNext();
        }
    }

    public void print()
    {
        ListItem item = first;
        while(item != null)
        {
            System.out.print(item.getValue() + " ");
            item = item.getNext();
        }
        System.out.println();
    }
}

```

Die Implementierung der Liste dürfte ohne weitere Erklärungen verständlich sein. Bevor wir zu unserem RMI-Beispiel zurückkehren, wird an einem Beispiel gezeigt, dass eine serialisierbare Datenstruktur auch in eine Datei abgespeichert werden kann. Anders als bei RMI, wo die Serialisierung und Deserialisierung ohne explizite Programmierung erfolgt, muss dieser Fall, wo die Liste in eine Datei gespeichert und daraus wieder gelesen werden soll, explizit ausprogrammiert werden. Zum Serialisieren und Deserialisieren werden die Klassen *ObjectOutputStream* bzw. *ObjectInputStream* aus dem Package `java.io` benutzt (s. Bild 5.5 in Abschnitt 5.5). Objekte dieser Klassen können mit einem `FileOutputStream`- bzw. `FileInputStream`-Objekt zusammengeschaltet werden, wenn das Ziel bzw. die Quelle eine Datei sein soll. Erfreulich ist, dass das Serialisieren durch eine einzige Anweisung, die aus dem Aufruf der Methode *writeObject* der Klasse *ObjectOutputStream* besteht, erfolgt. Das zu serialisierende Objekt muss dabei als Argument übergeben werden. Es werden dann alle Referenzen verfolgt und die entsprechenden Objekte werden dann ebenfalls rekursiv serialisiert. Ist ein Objekt nicht serialisierbar (d.h. die entsprechende Klasse implementiert nicht `Serializable`), dann wird eine Ausnahme ausgelöst und die Serialisierung abgebrochen. Zyklen werden erkannt. Das heißt, es gibt weder eine endlose Rekursion noch wird ein Objekt, das schon serialisiert wurde, nochmals serialisiert, wenn man über eine Folge von Referenzen zu einem Objekt kommt, das man schon behandelt hat. Entsprechend erfolgt die Deserialisierung durch Aufruf der Methode *readObject* der Klasse *ObjectInputStream*. Der Rückgabetyp von *readObject* ist `Object`. Es kann dann auf den Typ gecastet werden, der vorliegt.

Das Programm in Listing 6.12 kann eine Liste serialisieren und in eine Datei schreiben sowie die Liste aus der Datei durch Deserialisierung wieder rekonstruieren. Das erste Kommandozeilenargument des Programms sollte „lesen“ oder „schreiben“ sein. Wenn es „schreiben“ ist, dann erwartet das Programm den Namen einer Datei, in die geschrieben werden soll, sowie eine beliebige Anzahl weiterer Argumente, die alle ganzen Zahlen sein

sollten, und die den Inhalt der aufzubauenden und der zu serialisierenden Liste vorgeben (die Liste aus Listing 6.11 besteht aus Int-Werten). Das Programm könnte also z.B. mit den Parametern „schreiben beispiel.ser 5 4 3 2 1“ gestartet werden. Ist das erste Kommandozeilenargument „lesen“, dann erwartet das Programm lediglich ein weiteres Argument, nämlich den Namen einer Datei, aus der eine Liste eingelesen und ausgegeben werden soll. Ohne weitere Erläuterungen sollten Sie nun in der Lage sein, das Programm aus Listing 6.12, in dem der Aufruf der wichtigen Methoden `readObject` und `writeObject` fett gedruckt ist, zu verstehen:

### Listing 6.12

```
import java.io.*;

public class SerializeExample
{
    public static void main(String[] args)
    {
        if(args.length < 2)
        {
            usage();
        }
        else if(args[0].equals("lesen"))
        {
            read(args);
        }
        else if(args[0].equals("schreiben"))
        {
            write(args);
        }
        else
        {
            usage();
        }
    }

    private static void usage()
    {
        System.out.println("Nötige Kommandozeilenargumente: ");
        System.out.println("lesen <Name der Datei>");
        System.out.println("ODER");
        System.out.println("schreiben <Name der Datei> "
                           + "<Zahl 1> <Zahl 2> ... <Zahl N>");
    }

    private static void write(String[] args)
    {
        List l = new List();
        for(int i = 2; i < args.length; i++)
        {
            int value = Integer.parseInt(args[i]);
            l.append(value);
        }
        try(FileOutputStream fOutput =
                new FileOutputStream(args[1]));
            ObjectOutputStream output =
                new ObjectOutputStream(fOutput))

```

```

    {
        output.writeObject(l);
        output.flush();
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}

private static void read(String[] args)
{
    try(FileInputStream fInput = new FileInputStream(args[1]);
        ObjectInputStream input = new ObjectInputStream(fInput))
    {
        List l = (List) input.readObject();
        l.print();
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}
}

```

Bei der Serialisierung eines Objekts wird (neben dem Namen der Klasse des Objekts) im Wesentlichen der Zustand des Objekts in eine Byte-Folge gewandelt. Der Zustand eines Objekts besteht aus den aktuellen Werten seiner Attribute, die nicht static und nicht *transient* sind. Das heißt zum einen, dass man mit dem Schlüsselwort transient ein Attribut kennzeichnen kann, das nicht serialisiert werden soll:

```

public class X
{
    private transient String description;
    private int x;
    private int y;
    ...
}

```

Zum anderen bedeutet dies, dass die Daten und nicht der Programmcode (also nicht die Methoden, die in übersetzter Form als Byte-Code in einer Class-Datei stehen) serialisiert werden.

Das bei der angegebenen Serialisierung und Deserialisierung verwendete Format ist ein spezielles Binärformat, das von Menschen nur mühsam beispielsweise mit Hilfe eines Hex-Editors entziffert werden kann, was aber in der Regel zum Glück nicht nötig ist. Alternativ gibt es auch Klassen zur Serialisierung und Deserialisierung, bei denen die für Menschen leichter lesbaren Formate XML (eXtended Markup Language) oder JSON (JavaScript Object Notation) verwendet werden. Im Zusammenhang mit RESTful WebServices geht Unterabschnitt 7.9.2 auf die Serialisierung und Deserialisierung mit dem JSON-Format näher ein.

## 6.4.2 Serialisierung und Deserialisierung bei RMI

In dem folgenden RMI-Beispiel wird eine Liste als Parameter in einer RMI-Methode verwendet. Da die Liste serialisierbar ist, wird sie als Wert übergeben (diese Aussage ist nicht ganz korrekt; richtig muss es heißen: Da die Liste serialisierbar und nicht exportiert ist, wird sie als Wert übergeben; die Bedeutung des Exportierens ist aber momentan noch nicht zu verstehen). Wie wir zuvor gesehen haben, wäre eine typische Anwendung für die Wertübergabe ein rein lesender Zugriff auf die Liste auf der Server-Seite. Da wir aber die Wertübergabe deutlich machen wollen, werden wir eine (relativ unsinnige) RMI-Methode entwickeln, die an die Liste ein Element anhängt. Sie sollen dann sehen, dass zwar dadurch die Kopie auf dem Server verändert wird, das Original auf dem Client aber vor und nach dem RMI-Aufruf unverändert bleibt. Bitte beachten Sie, dass sich diese Situation von einem lokalen Methodenaufruf deutlich unterscheidet. Bei der Entwicklung der Anwendung wird wieder das „Kochrezept“ aus Abschnitt 6.1 verwendet.

**1. Schnittstelle definieren:** Wie soeben erläutert wurde, schreiben wir eine Anwendung zum Anhängen eines Elements an eine Liste. Diese Anwendung ist nicht besonders sinnvoll, dient aber zur Erläuterung des Prinzips Wertübergabe. Die Schnittstelle wird in Listing 6.13 gezeigt.

### Listing 6.13

```
public interface Append extends java.rmi.Remote
{
    public void tryToAppend(List l) throws java.rmi.RemoteException;
}
```

**2. Schnittstelle implementieren:** Die Implementierung der Schnittstelle ist sehr einfach, wie in Listing 6.14 zu sehen ist. Die Liste wird vor und nach der Veränderung am Bildschirm ausgegeben:

### Listing 6.14

```
import java.rmi.*;
import java.rmi.server.*;

public class AppendImpl extends UnicastRemoteObject
    implements Append
{
    public AppendImpl() throws RemoteException
    {
    }

    public void tryToAppend(List l) throws RemoteException
    {
        System.out.print("erhaltene Liste: ");
        l.print();

        l.append(4711);
        System.out.print("manipulierte Liste: ");
        l.print();
    }
}
```

Die Methode tryToAppend muss nicht synchronized sein, da selbst bei paralleler Nutzung jeder Methodenaufruf mit seiner eigenen Liste arbeitet.

**3. Server programmieren:** Der Server besteht aus einer Main-Methode, in der ein Objekt der Klasse AppendImpl erzeugt und unter dem Namen „AppendServer“ bei der Auskunft angemeldet wird (s. Listing 6.15).

#### Listing 6.15

```
import java.rmi.*;  
  
public class Server  
{  
    public static void main(String[] args)  
    {  
        try  
        {  
            AppendImpl server = new AppendImpl();  
            Naming.rebind("AppendServer", server);  
        }  
        catch(Exception e)  
        {  
            System.out.println(e);  
            e.printStackTrace();  
        }  
    }  
}
```

**4. Client programmieren:** Der Client (s. Listing 6.16) wird mit Kommandozeilenargumenten gestartet: das erste Argument ist der Name des Rechners, auf dem der Server läuft, die beliebig vielen weiteren Argumente sind die Elemente, die die als Parameter übergebene Liste enthalten soll.

#### Listing 6.16

```
import java.rmi.*;  
  
public class Client  
{  
    public static void main(String[] args)  
    {  
        if(args.length < 2)  
        {  
            System.out.println("Nötige Kommandozeilenargumente:"  
                + " <Name der Servers>"  
                + " <Zahl 1> <Zahl 2> ... <Zahl N>");  
            return;  
        }  
  
        try  
        {  
            Append server = (Append) Naming.lookup("rmi://" + args[0] + "/AppendServer");  
            System.out.println("Kontakt zu Server hergestellt");  
            List l = new List();  
            for(int i = 1; i < args.length; i++)  
                l.add(args[i]);  
            server.append(l);  
        }  
    }  
}
```

```
        {
            int value = Integer.parseInt(args[i]);
            l.append(value);
        }
        System.out.print("Liste vor RMI-Aufruf: ");
        l.print();
        server.tryToAppend(l);
        System.out.print("Liste nach RMI-Aufruf: ");
        l.print();
    }
} catch(Exception e)
{
    System.out.println(e);
    e.printStackTrace();
}
}
```

**5. Anwendung übersetzen und ausführen:** Nehmen wir an, Client und Server laufen auf demselben Rechner und wir starten den Client mit der Argumentliste „localhost 5 4 3 2 1“, dann produziert der Server die folgende Ausgabe:

```
erhaltene Liste: 5 4 3 2 1
manipulierte Liste: 5 4 3 2 1 4711
```

Beim Client ist dagegen folgende Ausgabe zu sehen:

```
Kontakt zu Server hergestellt
Liste vor RMI-Aufruf: 5 4 3 2 1
Liste nach RMI-Aufruf: 5 4 3 2 1
```

Man sieht also, dass auf dem Server die als Parameter übergebene Liste in der Methode verändert wird, dass dies aber keine Auswirkungen für den Client hat. Bei einem Aufruf einer Methode innerhalb einer JVM (einem lokalen Methodenaufruf) kann ein solches Verhalten nicht vorkommen.

Wie zu Beginn der Besprechung dieses Beispiels erläutert, ist das Beispiel nicht besonders sinnvoll. Wir können aber ein etwas sinnvollereres Beispiel daraus machen, wenn wir die geänderte Liste vom Server wieder auf den Client zurückkopieren. Dies kann leicht durch eine Rückgabe des Typs `List` bewerkstelligt werden. Die veränderte RMI-Schnittstelle ist in Listing 6.17 zu finden. Da der Client jetzt etwas von den Änderungen des Servers mitbekommt, nennen wir die Methode nun nicht mehr `tryToAppend`, sondern `append` (alle Änderungen gegenüber Listing 6.13 sind in Listing 6.17 fett gedruckt).

#### **Listing 6.17**

```
public interface Append extends java.rmi.Remote
{
    public List append(List l) throws java.rmi.RemoteException;
}
```

Die Implementierung aus Listing 6.14 muss dann entsprechend angepasst werden (s. Listing 6.18, Änderungen fett gedruckt).

**Listing 6.18**

```

import java.rmi.*;
import java.rmi.server.*;

public class AppendImpl extends UnicastRemoteObject
    implements Append
{
    public AppendImpl() throws RemoteException
    {
    }

    public List append(List l) throws RemoteException
    {
        System.out.print("erhaltene Liste: ");
        l.print();

        l.append(4711);
        System.out.print("manipulierte Liste: ");
        l.print();
        return l;
    }
}

```

Der Server kann unverändert bleiben. Der Client sollte den zurückgegebenen Wert in eine Variable übernehmen und zu Demonstrationszwecken ausgeben. Wenn wie in Listing 6.19 eine neue Variable dafür verwendet und die Parametervariable somit nicht überschrieben wird, dann hat man nach dem Aufruf zwei Listen zur Verfügung: eine, die so als Parameter übergeben wurde, und eine, die man vom Server zurückbekommen hat. Die zweite Liste hat am Ende ein Element mehr.

**Listing 6.19**

```

import java.rmi.*;
public class Client
{
    public static void main(String[] args)
    {
        if(args.length < 2)
        {
            System.out.println("Nötige Kommandozeilenargumente:"
                + " <Name der Servers>"
                + " <Zahl 1> <Zahl 2> ... <Zahl N>");
            return;
        }

        try
        {
            Append server = (Append) Naming.lookup("rmi://"
                + args[0]
                + "/AppendServer");
            System.out.println("Kontakt zu Server hergestellt");
            List l = new List();
            for(int i = 1; i < args.length; i++)
            {
                int value = Integer.parseInt(args[i]);
                l.append(value);
            }
        }
    }
}

```

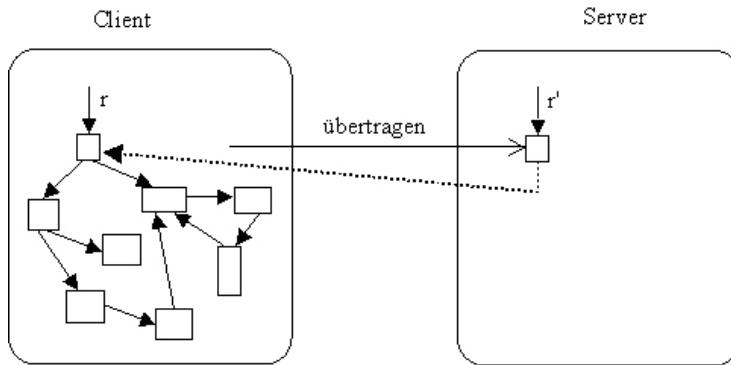
```
        }
        List lReturned = server.append(l);
        System.out.print("Nach dem RMI-Aufruf noch vorhandene"
            + " Liste, die bei der Parameterübergabe"
            + " an den Server als Wert übergeben"
            + " wurde: ");
        l.print();
        System.out.print("Vom RMI-Aufruf zurückgelieferte"
            + " Liste: ");
        lReturned.print();
    }
} catch(Exception e)
{
    System.out.println(e);
    e.printStackTrace();
}
}
```

## ■ 6.5 Referenzübergabe für Parameter und Rückgabewerte

Wir beschäftigen uns in diesem Abschnitt mit der *Referenzübergabe* („*Call by Reference*“) bei Parametern und Rückgabewerten von Methoden. Die Referenzübergabe ist ja die einzige Übergabeart bei lokalen Methodenaufrufen. Was bedeutet aber Referenzübergabe im Fall eines Fern-Methodenaufrufs? Das Objekt, dessen Referenz als Parameter übergeben wird, befindet sich auf dem Client und kann später vom Server aus benutzt werden. Diese Nutzung bedeutet, dass Methoden auf dem Client-Objekt vom Server aufgerufen werden können, und dies natürlich über RMI. Folglich muss das Objekt, dessen Referenz als Parameter übergeben wird, auch ein RMI-Objekt sein. Das heißt, dass die Klasse des Objekts eine Remote-Schnittstelle implementieren und selbst aus UnicastRemoteObject abgeleitet sein muss. Die Tatsache, dass der Server den Client „zurückruft“, wird auch als *Callback* bezeichnet.

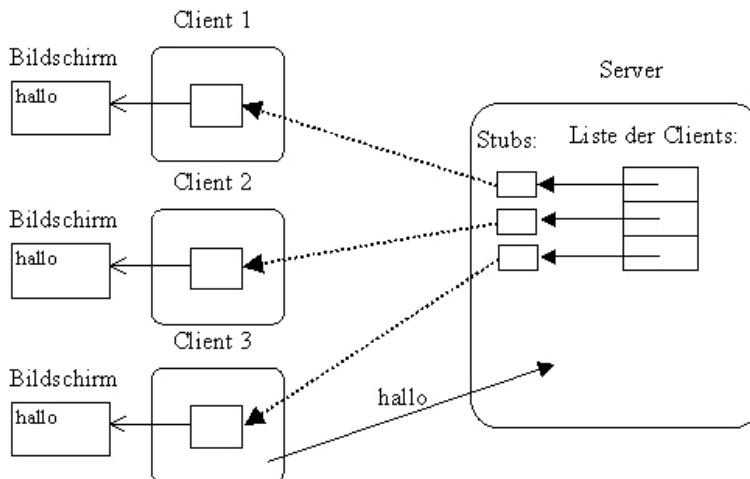
In Bild 6.8 wird die Referenzübergabe für einen Fern-Methodenaufruf dargestellt. Wenn eine RMI-Methode vom Client auf dem Server aufgerufen und dabei als Parameter die Referenz r übergeben wird, dann wird der entsprechenden Methode auf dem Server die Referenz r' übergeben. Mit dieser Referenz kann der Server auf das Objekt, das sich auf dem Client befindet, über RMI zugreifen. Mit anderen Worten: Die Referenz r' zeigt auf einen Stub für das Client-Objekt. Das heißt also, dass beim Methodenaufruf mit Referenzübergabe als Parameter ein Stub übertragen wird. Entsprechendes gilt für von Methoden zurückgegebene Objekte, die als Referenz übertragen werden.

Eine Übergabe eines Parameters als Referenz ist notwendig, wenn der Server die Datenstruktur auf dem Client ändern muss oder wenn die auf dem Client ausgeführten Methoden Seiteneffekte haben sollen, die sich gezielt auf dem Client manifestieren sollen. Beispiele für solche Seiteneffekte sind z. B. Ausgaben auf dem Bildschirm oder in Dateien des lokalen Dateisystems des Clients.



**Bild 6.8** Referenzübergabe von Objektparametern bei RMI

Das Beispiel dieses Abschnitts ist ein Chat-Beispiel (Chat: Schwatzen), in dem die Referenzübergabe wegen der gerade beschriebenen Seiteneffekte benötigt wird. Unser Chat-Programm soll so realisiert werden, dass ein Server als eine Art Verstärker für die Redebeiträge der einzelnen Chat-Teilnehmer arbeitet (s. Bild 6.9). Das bedeutet: Ein Teilnehmer teilt durch einen RMI-Aufruf dem Server seinen Redebeitrag mit (in Bild 6.9 einfach dargestellt durch den Pfeil mit der Beschriftung „hallo“ von Client 3 zum Server). Der Server muss dann diesen Redebeitrag an alle Teilnehmer verteilen. Da dies ebenfalls über RMI geschehen soll, braucht der Server Referenzen auf je ein Objekt jedes Clients. Diese Referenz wird dem Chat-Server bei der Anmeldung eines neuen Clients ebenfalls über einen RMI-Methodenaufruf übergeben. Der Server muss also eine Liste aller angemeldeten Clients führen. Diese Liste enthält Referenzen (in Form von Stubs) auf die bei der Anmeldung übergebenen Client-Objekte. Damit kann der Server auf alle Client-Objekte einen entsprechenden Methodenaufruf absetzen. Dieser wird bei den Clients so realisiert, dass Text auf dem Bildschirm ausgegeben wird. An dieser Stelle sieht man nun deutlich, dass der Methodenaufruf des Servers bei den Clients einen Seiteneffekt besitzt, der sich natürlich bei den Clients auswirken muss und nicht beim Server.



**Bild 6.9** Prinzip der RMI-Chat-Anwendung

Wir entwickeln nun die Anwendung wieder auf bewährte Art und Weise.

**1. Schnittstellen definieren:** In dieser Anwendung benötigen wir gemäß den soeben gegebenen Erläuterungen zwei RMI-Schnittstellen:

- eine Schnittstelle des Servers namens ChatServer, die Clients zum An- und Abmelden (Methoden addClient und removeClient) sowie zum Übermitteln eines Redebeitrags an den Server (Methode sendMessage) benutzen,
- und eine Schnittstelle des Clients namens ChatClient, die der Server zum Übermitteln eines Redebeitrags an den Client (Methode print) benutzt.

Die Schnittstelle ChatClient enthält daneben eine weitere Methode namens getName. Damit kann der Server den Spitznamen eines Clients erfragen. Diese Methode wird in der Anwendung intensiv genutzt. Man könnte den Spitznamen eines Clients dem Server beim Anmelden als zusätzlichen Parameter einfach mitteilen. Bei der hier gezeigten Realisierung wird aber kein besonderer Wert auf Effizienz gelegt. Stattdessen steht eine intensive Nutzung von RMI zu Demonstrationszwecken im Vordergrund.

Wir zeigen zunächst in Listing 6.20 die Schnittstelle ChatClient:

#### **Listing 6.20**

```
public interface ChatClient extends java.rmi.Remote
{
    public String getName() throws java.rmi.RemoteException;
    public void print(String msg) throws java.rmi.RemoteException;
}
```

In Listing 6.21 folgt die Schnittstelle ChatServer:

#### **Listing 6.21**

```
import java.rmi.*;

public interface ChatServer extends Remote
{
    public boolean addClient(ChatClient objRef)
        throws RemoteException;
    public void removeClient(ChatClient objRef)
        throws RemoteException;
    public void sendMessage(String name, String msg)
        throws RemoteException;
}
```

Die Methode addClient gibt einen boolean-Wert zurück. Falls die Anmeldung erfolgreich war, wird true zurückgegeben, sonst false. Eine Anmeldung ist immer dann erfolgreich, falls niemand anders unter diesem Namen angemeldet ist (d.h. der Name des Clients muss eindeutig sein).

Der entscheidende und wichtige Punkt bei der Definition der beiden Schnittstellen ist der Typ des Parameters der Methoden addClient und removeClient. Als Typ ist hier ChatClient angegeben. Dies ist eine RMI-Schnittstelle. Beim Aufruf der Methode addClient durch den Client wird hier ein Objekt angegeben, dessen Klasse diese Schnittstelle implementiert. Beim Aufruf der „tatsächlichen“ Methode auf dem Chat-Server wird aber als Parameter eine Referenz auf ein entsprechendes Stub-Objekt angegeben (vgl. Bild 6.8 mit den Referenzen r

und r'). Deshalb können wir hier als Typ des Parameters nicht die Implementierungsklasse angeben. Die Angabe der Schnittstelle aber, die sowohl von der von uns selbst geschriebenen Klasse als auch von der automatisch generierten Stub-Klasse implementiert wird, passt für beide Fälle, sowohl für den Aufruf auf Client-Seite als auch auf Server-Seite. Die Vorgehensweise für Referenzübergabe lautet also: in der Schnittstelle muss als Typ eine RMI-Schnittstelle angegeben werden.

**2. Schnittstellen implementieren:** Wir haben im ersten Schritt zwei RMI-Schnittstellen definiert. Folglich müssen wir auch zwei Schnittstellen implementieren. Betrachten wir zuerst die Implementierung der Server-Schnittstelle ChatServer in Listing 6.22. Die Liste der Referenzen auf die Client-Objekte, genauer auf die dazugehörigen Stubs (s. Bild 6.9), wird durch ein Objekt der Klasse ArrayList des Packages java.util realisiert. In einer ArrayList können Referenzen auf Objekte jeder beliebigen Klasse mit der Methode add gespeichert und mit der Methode remove wieder gelöscht werden. Seit Java 5 kann über das Sprachkonzept Generics angegeben werden, dass in die ArrayList nur Objekte eines bestimmten Typs, in diesem Fall des Typs ChatClient, eingefügt werden dürfen. Mit Hilfe eines so genannten Iterators, den man sich durch die Methode iterator von einem ArrayList-Objekt beschaffen kann, kann man alle abgespeicherten Elemente der ArrayList durchlaufen. Auch der Iterator ist durch den Typ der in der ArrayList enthaltenen Elemente parametrisiert. Die Methode hasNext des Iterators gibt an, ob es weitere Elemente gibt, oder ob man bereits alle gesehen hat. Das jeweils nächste Element des ArrayList-Objekts kann man sich durch die Methode next vom Iterator beschaffen.

### Listing 6.22

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.*;

public class ChatServerImpl extends UnicastRemoteObject
    implements ChatServer
{
    private ArrayList<ChatClient> allClients;

    public ChatServerImpl() throws RemoteException
    {
        allClients = new ArrayList<ChatClient>();
    }

    public synchronized boolean addClient(ChatClient objRef)
        throws RemoteException
    {
        String name = objRef.getName();
        for(Iterator<ChatClient> iter = allClients.iterator();
            iter.hasNext();)
        {
            ChatClient cc = iter.next();
            try
            {
                if(cc.getName().equals(name))
                {
                    return false;
                }
            }
        }
    }
}
```

```

        catch(RemoteException exc)
        {
            iter.remove();
        }
    }
    allClients.add(objRef);
    return true;
}

public synchronized void removeClient(ChatClient objRef)
    throws RemoteException
{
    allClients.remove(objRef);
}

public synchronized void sendMessage(String name, String msg)
    throws RemoteException
{
    for(Iterator<ChatClient> iter = allClients.iterator();
        iter.hasNext());
    {
        ChatClient cc = iter.next();
        try
        {
            cc.print(name + ": " + msg);
        }
        catch(RemoteException exc)
        {
            iter.remove();
        }
    }
}
}

```

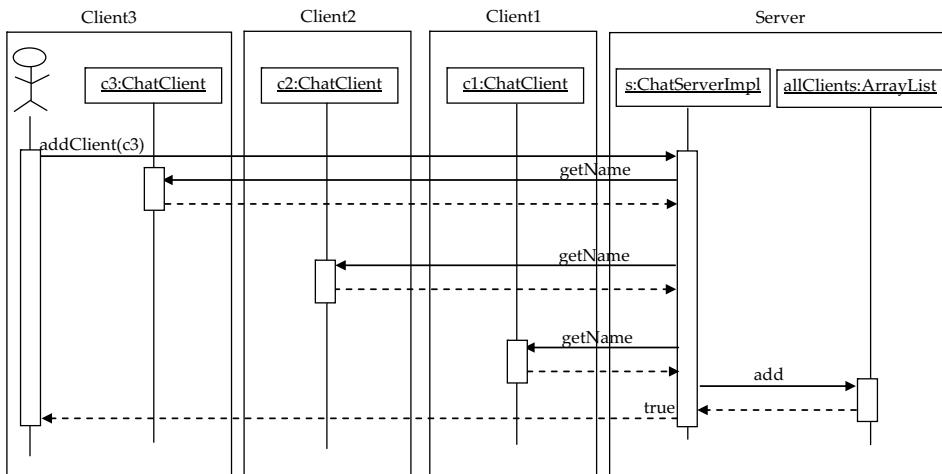
Auch wenn diese Klasse nicht viele Zeilen Code enthält, so hat sie es dennoch in sich; es gibt eine Reihe von Details zu besprechen. Zunächst fällt auf, dass alle drei Methoden der Klasse ChatServerImpl synchronized sind. Dies ist aus mehreren Gründen nötig. Zum einen sind die Methoden der Klasse ArrayList nicht synchronized, so dass es beim parallelen Einfügen und Löschen in die ArrayList zu Problemen kommen könnte. Ein gegenseitiger Ausschluss für die ArrayList-Methoden wäre aber nicht ausreichend, denn auch während des Durchlaufens der Liste sollen keine Änderungen an der Liste vorgenommen werden. Darüber hinaus könnte ohne synchronized die Eindeutigkeit für die Namen der angemeldeten Benutzer nicht mehr garantiert werden. Damit dies klarer wird, betrachten wir den Fall, dass sich zwei Clients unter demselben Namen anmelden wollen. Wenn die Methode addClient nicht synchronized wäre, dann könnten beide gleichzeitig prüfen, ob es diesen Namen schon gibt. Da der jeweils andere Client seinen Namen noch nicht eingetragen hätte, würden beide zu dem Schluss kommen, dass eine Anmeldung möglich ist. Daraufhin würden sich beide anmelden. Damit ist eine Situation entstanden, die eigentlich vermieden werden sollte. Da addClient aber als synchronized gekennzeichnet ist, ist der eben beschriebene Ablauf nicht möglich.

Ein weiterer bemerkenswerter Punkt in Listing 6.22 ist die Tatsache, dass alle drei Methoden der Klasse ChatServerImpl intensiv Gebrauch vom Callback-Mechanismus machen, ohne dass dies besonders auffällt. In der Methode addClient wird zunächst die Methode

getName auf das sich gerade anmeldende Objekt angewendet. Bitte beachten Sie, dass diese Methode den Client „zurückruft“ und sich dabei den Namen des sich anmeldenden Clients beschafft. Anschließend wird mit einer for-Schleife geprüft, ob es diesen Namen bereits gibt. Dazu erfolgt ein Rückruf bei allen Clients, die sich zuvor bereits angemeldet haben. Es sei an dieser Stelle daran erinnert, dass unsere Chat-Anwendung primär nicht auf hohe Effizienz zielt, sondern das Prinzip der Referenzübergabe bei RMI intensiv demonstrieren soll.

Wird der Name bei den bereits angemeldeten Clients gefunden, ist die Methode zu Ende; ihr Rückgabewert ist false. Damit wird angezeigt, dass der Client nicht angemeldet werden konnte. Falls es den Namen bislang noch nicht gibt, wird der Client in die ArrayList all-Clients aufgenommen. Die Methode gibt in diesem Fall true zurück, um anzudeuten, dass die Anmeldung erfolgreich durchgeführt wurde.

Das erfolgreiche Anmelden eines Clients ist in Bild 6.10 in Form eines UML-Sequenzdiagramms abgebildet. In horizontaler Richtung sind die beteiligten Objekte aufgetragen, in vertikaler Richtung (von oben nach unten) die Zeit. Es ist dargestellt, welche Methode welche Methode aufruft und auf welches Objekt eine Methode jeweils angewendet wird. Die Zeit, in der eine Methode aktiv ist, wird durch einen weißen Balken angezeigt. Der Methodenaufruf wird durch einen durchgezogenen Pfeil, die Methodenrückkehr durch einen gestrichelten Pfeil dargestellt. Im Diagramm wird der Umgang mit dem Iterator nicht gezeigt. Bei den Client-Objekten wird nicht die Klasse, sondern die implementierte Schnittstelle ChatClient angegeben. Ferner ist nicht näher spezifiziert, aus welcher Methode heraus der Aufruf addClient erfolgt und auf welches Objekt diese Methode angewendet wurde. Statt eines Objekts wird eine menschliche Figur skizziert. Damit wird angedeutet, dass der Aufruf addClient von einer Benutzeraktion herrührt.

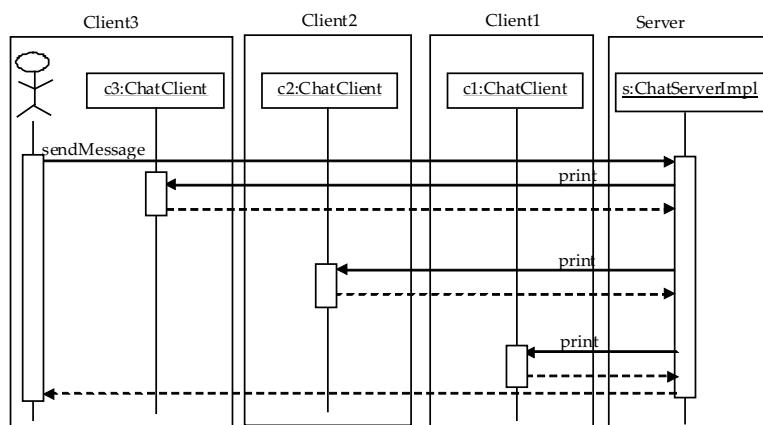


**Bild 6.10** UML-Sequenzdiagramm für das Anmelden eines Clients

Beim Abmelden eines Clients wird der Methode `removeClient` auf Client-Seite dasselbe Objekt übergeben wie beim Aufruf von `addClient`. Auf der Server-Seite wird der Aufruf aber mit einem Stub-Objekt durchgeführt, wobei für jeden Aufruf ein neues Stub-Objekt zur Verfügung gestellt wird. Das heißt, für denselben Client wird `addClient` ein anderer Parameter übergeben als `removeClient`. Die `ArrayList`-Methode `remove` entfernt aus der Liste dasjenige

Objekt, das mit dem als Parameter übergebenen Objekt gleich, nicht aber notwendig identisch ist (das heißt: die Methode equals liefert true zurück, der Vergleich mit == muss nicht, wird aber in der Regel false sein). Zwei Stub-Objekte sind genau dann gleich, wenn sie eine Referenz auf dasselbe Objekt in der Ferne repräsentieren. Aus diesem Grund funktioniert nun also das Entfernen eines Stubs aus der ArrayList mit der Methode remove; der Client gibt bei seinem Aufruf von removeClient zwar genau dasselbe Objekt an wie bei seinem Aufruf von addClient, aber auf dem Server werden removeClient und addClient mit unterschiedlichen Stub-Objekten aufgerufen, die aber auf dasselbe Objekt auf dem Client zeigen und für die damit equals gilt. Somit kann also durch Angabe eines anderen Objekts in removeClient das in addClient der Liste hinzugefügte Element wieder entfernt werden.

Mit Hilfe der Methode sendMessage schließlich kann ein Redebeitrag an alle angemeldeten Clients verteilt werden. Ein beispielhafter Ablauf für das Senden eines Redebeitrags ist wiederum als UML-Sequenzdiagramm in Bild 6.11 zu sehen.



**Bild 6.11** UML-Sequenzdiagramm für das Übermitteln eines Redebeitrags

Ein letzter erwähnenswerter Aspekt der Klasse ChatServerImpl, der sowohl in der Methode addClient als auch in der Methode sendMessage vorkommt, ist die Behandlung der Ausnahme RemoteException. Wenn diese Ausnahme ausgelöst wird, kann man davon ausgehen, dass der betreffende Client nicht mehr verfügbar ist, z.B. weil er abgebrochen wurde, ohne sich vorher „ordentlich“ abzumelden. Als Reaktion wird der Client dann folgerichtig aus der Liste entfernt. Es ist wichtig, dass zum Entfernen nicht die Methode remove der Klasse ArrayList verwendet wird, sondern dass dies über den Iterator erfolgt, denn sonst würde beim weiteren Iterieren eine ConcurrentModificationException geworfen werden.

Wenden wir uns nun der Implementierung der ChatClient-Schnittstelle zu. Für den Client werden zwei Realisierungen angegeben: eine einfache, kommandozeilenorientierte Version und eine etwas komfortablere Version mit grafischer Benutzeroberfläche. Für das Verständnis der Chat-Anwendung bezüglich RMI genügt es, wenn Sie die einfache Version verstehen. Zu der einfachen Version gehört die Klasse ChatClientImplSimple (s. Listing 6.23), welche die ChatClient-Schnittstelle implementiert. Diese Implementierung der Client-Schnittstelle ist sehr einfach. Dem Konstruktor wird der Name des Chat-Teilnehmers als Parameter übergeben. Dieser Parameter wird in das Attribut übernommen. Der Name kann

später durch die Methode getName wieder abgefragt werden. Davon macht unter anderem auch der Server in der Methode addClient Gebrauch (s. Bild 6.10). Die Methode print wird durch eine simple Ausgabe auf den Bildschirm mit System.out.println realisiert.

### Listing 6.23

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

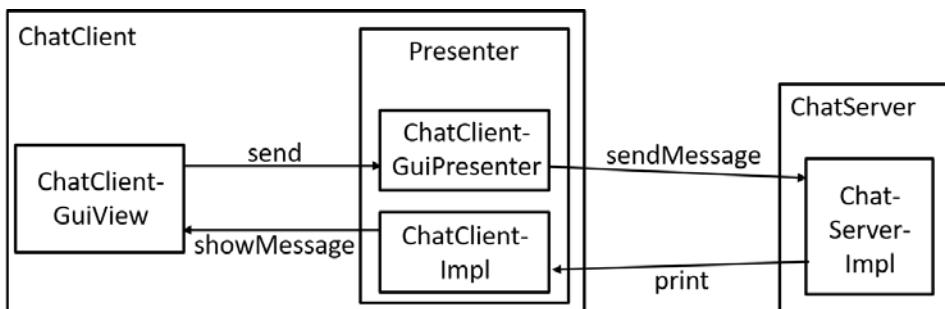
public class ChatClientImplSimple extends UnicastRemoteObject
    implements ChatClient
{
    private String name;

    public ChatClientImplSimple(String n) throws RemoteException
    {
        name = n;
    }

    public String getName() throws RemoteException
    {
        return name;
    }

    public void print(String msg) throws RemoteException
    {
        System.out.println(msg);
    }
}
```

Die Client-Version mit grafischer Benutzeroberfläche wird wieder nach dem MVP-Prinzip gestaltet (s. Bild 6.12). Die Presenter-Komponente besteht in diesem Fall aus dem „klassischen“ Teil, der für die Steuerung des Ablaufs zuständig ist, der über die Benutzeroberfläche angestoßen wird (s. später). Der andere Teil des Presenters steuert den Ablauf, der von Seiten des Modells, das sich auf dem Server befindet, über RMI-Callback-Aufrufe angestoßen wird. Dieser zweite Teil des Presenters besteht aus einer Implementierung der ChatClient-Schnittstelle.



**Bild 6.12** Fenster der Chat-Client-Version mit grafischer Benutzeroberfläche

Die Implementierung der ChatClient-Schnittstelle ist in Listing 6.24 zu sehen. Die Methode print gibt den auszugebenden String an eine dafür vorgesehene Methode der View weiter.

Da die RMI-Callback-Aufrufe von Threads durchgeführt werden, die automatisch durch den RMI-Mechanismus erzeugt werden, müssen wir dazu wieder Platform.runLater einsetzen.

#### **Listing 6.24**

```
import java.rmi.RemoteException;
import java.rmi.server.UnicastRemoteObject;
import javafx.application.Platform;

public class ChatClientImpl extends UnicastRemoteObject
                                implements ChatClient
{
    private String name;
    private ChatClientGuiView view;

    public ChatClientImpl(String name, ChatClientGuiView view)
        throws RemoteException
    {
        this.name = name;
        this.view = view;
    }

    public String getName() throws RemoteException
    {
        return name;
    }

    public void print(String msg) throws RemoteException
    {
        Platform.runLater(()->view.showMessage(msg));
    }
}
```

**3. Server programmieren:** Der Server hat dieselbe Form wie in allen Beispielen zuvor (s. Listing 6.25):

#### **Listing 6.25**

```
import java.rmi.*;

public class ChatServerMain
{
    public static void main(String[] args)
    {
        try
        {
            ChatServerImpl server = new ChatServerImpl();
            Naming.rebind("ChatServer", server);
        }
        catch(Exception e)
        {
            System.out.println(e);
            e.printStackTrace();
        }
    }
}
```

Denkbar wäre, im Server mehrere ChatServerImpl-Objekte zu erzeugen und diese unter unterschiedlichen Namen bei der RMI-Registry anzumelden, wie dies beim Sleep-Beispiel in Abschnitt 6.3 gezeigt wurde. Damit könnte man leicht unterschiedliche Chat-Gruppen für verschiedene Themenbereiche realisieren (häufig auch als Chat-Räume oder Chat-Gruppen bezeichnet). Die Clients gehen im Folgenden bereits von dieser Möglichkeit aus, denn beim Starten des Programms muss u.a. der Name des RMI-Objektes angegeben werden, das man benutzen möchte. Dieser Name fungiert als Name der Chat-Gruppe.

**4. Clients programmieren:** In diesem Fall haben wir zwei Versionen zu programmieren, eine ohne und eine mit grafischer Benutzeroberfläche. Beide Client-Versionen müssen mit drei Kommandozeilenargumenten gestartet werden: mit einem selbst gewählten Namen, unter dem die eigenen Redebbeiträge angezeigt werden sollen, mit dem Namen des Rechners, auf dem der Server gestartet wurde, und mit dem Namen, unter dem das RMI-Objekt registriert wurde.

In der kommandozeilenorientierten Client-Version (s. Listing 6.26) wird zunächst – wie in allen anderen Beispielen auch – über Naming.lookup eine Referenz auf ein Server-Objekt beschafft. Danach wird ein eigenes Chat-Client-Objekt mit dem eigenen Namen als Argument erzeugt. Dieses Objekt wird als Parameter der Methode addClient übergeben, mit der sich der Client auf dem Server anmeldet. Falls die Anmeldung erfolgreich war, wird die lokale Methode sendInputToServer aufgerufen. Diese liest so lange Zeilen von der Tastatur und sendet sie mit sendMessage an den Server, bis „Ende“ oder „ende“ eingegeben wird. Daraufhin ist die Methode beendet. Zurück im Hauptprogramm wird der Client beim Server abgemeldet und der Client beendet.

### Listing 6.26

```
import java.rmi.*;
import java.io.*;

public class ChatClientMainSimple
{
    public static void main(String[] args)
    {
        if(args.length != 3)
        {
            System.out.println("Argumente: Spitzname Server Chat-Gruppe");
            return;
        }

        try
        {
            ChatServer server = (ChatServer) Naming.lookup("rmi://" + 
                args[1] + "/" + 
                args[2]);
            System.out.println("Server kontaktiert.");

            ChatClientImplSimple client =
                new ChatClientImplSimple(args[0]);
            if(server.addClient(client))
            {
                System.out.println("Ende durch Eingabe von 'ende'");
                sendInputToServer(server, args[0]);
                server.removeClient(client);
            }
        }
    }
}
```

```

        }
    else
    {
        System.out.println("Name auf Server schon bekannt.");
    }
}
catch(Exception e)
{
    System.out.println(e);
}
System.exit(0);
}

private static void sendInputToServer(ChatServer server, String name)
{
    try
    {
        BufferedReader input = new BufferedReader(
            new InputStreamReader(System.in));
        String line;
        while((line = input.readLine()) != null)
        {
            if(line.equals("ende") || line.equals("Ende"))
            {
                break;
            }
            server.sendMessage(name, line);
        }
    }
    catch(Exception e)
    {
        System.out.println(e);
    }
}
}
}

```

System.in ist ein InputStream. Darauf wird ein InputStreamReader gesetzt, mit dem die von der Tastatur kommenden Bytes in Zeichen gewandelt werden. Darauf wird ein BufferedReader gesetzt. Damit kann mit readLine jeweils eine komplette Zeile gelesen werden. Es sei noch erwähnt, dass am Ende der Main-Methode zum expliziten Beenden des Prozesses System.exit aufgerufen wird. Ohne diesen Aufruf wäre nur die Main-Methode zu Ende, nicht aber der Prozess, da durch das Erzeugen des RMI-Objekts des Typs ChatClientImplSimple zusätzliche Threads zuvor gestartet worden sind.

Wenden wir uns nun der Client-Version mit grafischer Benutzeroberfläche zu. In Bild 6.12 wurde die Struktur schon gezeigt und eine Klasse wurde auch schon implementiert. Es fehlt noch die View-Klasse, die keine Besonderheiten enthält: Zum einen wird die Oberfläche aufgebaut. Diese besitzt lediglich ein einzeiliges Eingabefeld (TextField), um die eigenen Nachrichten einzugeben, und einen mehrzeiligen, nicht vom Benutzer veränderbaren Ausgabebereich (TextArea), um den Chat-Verlauf darzustellen (s. Bild 6.14). Zum anderen gibt es eine Methode, um eine neue Nachricht in den Ausgabebereich einzufügen. Vom Presenter müssen wir noch den fehlenden Teil implementieren, der auf die Eingabe einer neuen Nachricht reagiert. Die Reaktion besteht aus einem Aufruf der RMI-Methode sendMessage, die wieder in einen Thread ausgelagert wird. Der Presenter hat üblicherweise sowohl eine

Referenz auf die View als auch auf das Modell. Hier ist es so, dass der Presenter zweigeteilt ist: Der eine Presenter-Teil kennt nur das Modell, der andere nur die View. Darüber hinaus kennen beide den eigenen Spitznamen der Benutzerin. In der Hauptklasse ChatClientGui schließlich werden unter Nutzung der Kommandozeilenargument der Kontakt zum Server hergestellt, der Client angemeldet, die beteiligten Bausteine erzeugt und miteinander „verdrahtet“ sowie wie üblich das Fenster auf dem Bildschirm angezeigt. Das Client-Programm mit grafischer Benutzeroberfläche finden Sie in Listing 6.27. Bitte beachten Sie, dass in der Main-Methode nach launch noch ein Aufruf von System.exit erfolgt, denn ohne diesen Aufruf würde der Prozess nach dem Schließen des Fensters und der Rückkehr aus der Methode launch und main wegen des RMI-Objekts des Typs ChatClientImpl weiterhin existieren. Auch beim kommandozeilenorientierten Client (s. Listing 6.26) wurde aus diesem Grund schon System.exit aufgerufen.

**Listing 6.27**

```
import java.rmi.Naming;
import java.rmi.RemoteException;
import java.util.List;
import javafx.application.*;
import javafx.geometry.Insets;
import javafx.stage.Stage;
import javafx.scene.*;
import javafx.scene.control.*;
import javafx.scene.layout.*;

class ChatClientGuiView
{
    private ChatClientGuiPresenter presenter;
    private BorderPane root;
    private TextField input;
    private TextArea output;

    public ChatClientGuiView(ChatClientGuiPresenter presenter)
    {
        this.presenter = presenter;
        initView();
    }

    private void initView()
    {
        root = new BorderPane();
        Insets ins = new Insets(10);
        root.setPadding(ins);

        input = new TextField();
        input.setOnAction(e -> handleInput());
        root.setTop(input);
        BorderPane.setMargin(input, new Insets(8));
        output = new TextArea();
        output.setEditable(false);
        root.setCenter(output);
        BorderPane.setMargin(output, new Insets(8));
    }

    private void handleInput()
```

```
{  
    String message = input.getText();  
    input.setText("");  
    presenter.send(message);  
}  
  
public Pane getUI()  
{  
    return root;  
}  
  
public void showMessage(String message)  
{  
    output.appendText(message + "\n");  
}  
}  
  
class ChatClientGuiPresenter  
{  
    private ChatServer model;  
    private String name;  
  
    public void setModelAndName(ChatServer model, String name)  
{  
        this.model = model;  
        this.name = name;  
    }  
  
    public void send(String message)  
{  
        new Thread(()->sendInThread(message)).start();  
    }  
  
    private void sendInThread(String message)  
{  
        try  
        {  
            model.sendMessage(name, message);  
        }  
        catch(RemoteException e)  
        {  
            System.out.println("Es gibt Probleme mit RMI!");  
        }  
    }  
}  
  
public class ChatClientGui extends Application  
{  
    public void start(Stage primaryStage)  
{  
        List<String> args = getParameters().getUnnamed();  
        String name = args.get(0);  
        ChatClientGuiPresenter p = new ChatClientGuiPresenter();  
        ChatClientGuiView v = new ChatClientGuiView(p);  
        try  
        {  
            ChatServer m = (ChatServer) Naming.lookup("rmi://" +  
                args.get(1) + "/" +
```

```
                                args.get(2));
        ChatClient callback = new ChatClientImpl(name, v);
        if(!m.addClient(callback))
        {
            System.out.println("Anmeldung beim Server gescheitert!");
            System.exit(0);
        }
        p.setModelAndName(m, name);
    }
    catch(Exception e)
    {
        System.out.println("Verbindung zu Server nicht möglich!");
        System.exit(0);
    }

    Scene scene = new Scene(v.getUI());
    primaryStage.setScene(scene);
    primaryStage.setTitle(name);
    primaryStage.show();
}

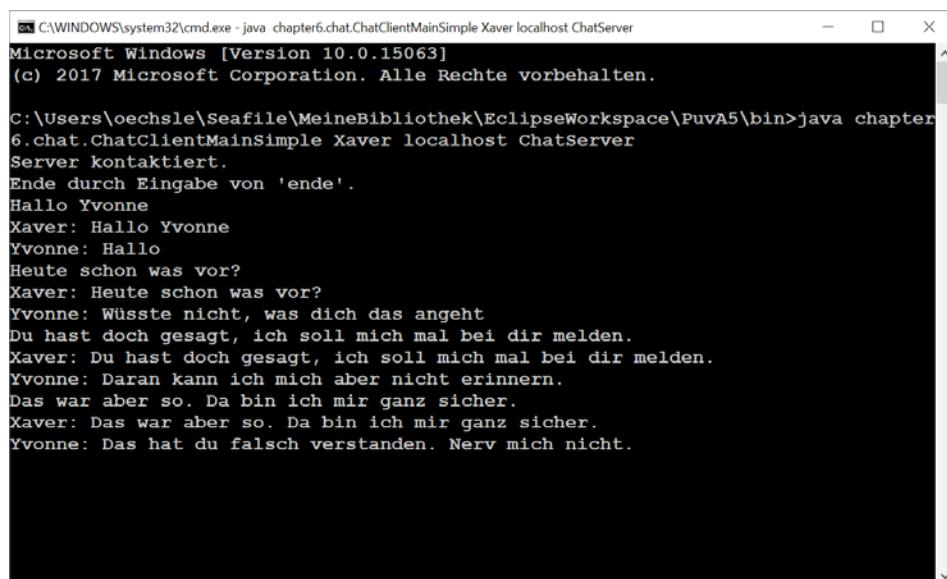
public static void main(String[] args)
{
    if(args.length != 3)
    {
        System.out.println("Argumente: Spitzname Server Chat-Gruppe");
        Platform.exit();
        return;
    }
    launch(args);
    System.exit(0);
}
```

Im Gegensatz zum kommandozeilenbasierten Client gibt es in diesem Client-Programm kein Abmelden vom Server. Dies könnte man noch hinzufügen, indem man an der Stage mit setOnCloseRequest einen Listener anmeldet, der auf das Schließen des Fensters reagiert. Darin könnte man den Client abmelden. Dies müsste man im optimalen Fall aber so programmieren, dass man den RMI-Aufruf zwar wieder in einen Thread auslagert, diesem Thread dann aber noch etwas Zeit gibt seinen RMI-Aufruf durchzuführen und nicht direkt nach dem Starten des Threads den Prozess beendet. Aber auch wenn wir das Programm um das Abmelden beim Server ergänzen würden, könnte es immer noch vorkommen, dass das Programm ohne Abmeldung beendet wird, da man den Prozess auch in anderer Weise (zum Beispiel über Betriebssystemmechanismen oder über seine Entwicklungsumgebung) zwangsweise beenden kann. Diese fehlende Abmeldung ist aber unkritisch, denn beim nächsten Aufruf der Methode addClient oder sendMessage auf dem Server durch einen anderen Client wird der RMI-Callback zum nicht mehr vorhandenen Client scheitern und eine Entfernung dieses Clients aus der Liste der angemeldeten Teilnehmer zur Folge haben. Auf diese Maßnahme auf Server-Seite kann also auch dann nicht verzichtet werden, wenn der Client sich beim Schließen des Fensters abmeldet. Deshalb haben wir uns der Einfachheit halber im vorigen Client-Programm das Abmelden erspart.

**5. Anwendung übersetzen und ausführen:** Es sei noch einmal darauf hingewiesen, dass die RMI-Registry nur auf dem Server-Rechner gestartet werden muss, nicht aber auf den

Client-Rechnern, auch wenn sich nun auf den Clients Objekte befinden, deren Methoden vom Server mit RMI aufgerufen werden. Mit Hilfe der RMI-Registry erhält ein Client eine Referenz auf das Server-Objekt. Dagegen erhält der Server die Referenz auf die Client-Objekte durch die Methode addClient, nicht aber mit Naming.lookup über eine RMI-Registry. Die Client-Objekte werden gar nicht mit Naming.rebind bei einer RMI-Registry angemeldet. Daher ist eine RMI-Registry auf einem Client nicht nötig (das Starten einer RMI-Registry würde aber auch nicht schaden).

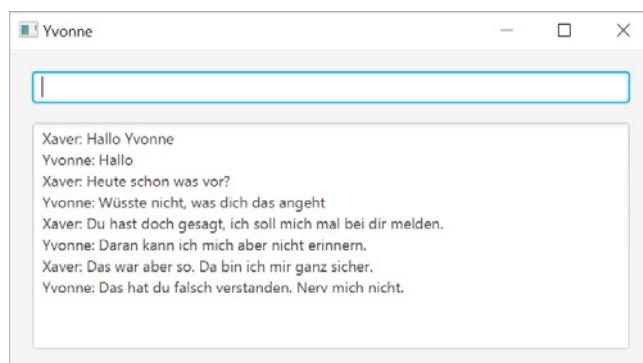
Beachten Sie bitte, dass die Chat-Teilnehmer, die miteinander sprechen („chatten“), nicht gezwungen sind, dieselben Client-Versionen zu verwenden. In Bild 6.13 und Bild 6.14 wird dies an einem Beispiel demonstriert. Die Teilnehmer Xaver und Yvonne führen ein Gespräch miteinander. Xaver verwendet dabei die kommandozeilenorientierte Version (Bild 6.13), während Yvonne die komfortablere Version mit grafischer Benutzeroberfläche (Bild 6.14) bevorzugt.



```
C:\WINDOWS\system32\cmd.exe - java chapter6.chat.ChatClientMainSimple Xaver localhost ChatServer
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\oechsle\Seafie\MeineBibliothek\EclipseWorkspace\PuvA5\bin>java chapter6.chat.ChatClientMainSimple Xaver localhost ChatServer
Server kontaktiert.
Ende durch Eingabe von 'ende'.
Hallo Yvonne
Xaver: Hallo Yvonne
Yvonne: Hallo
Heute schon was vor?
Xaver: Heute schon was vor?
Yvonne: Wüsste nicht, was dich das angeht
Du hast doch gesagt, ich soll mich mal bei dir melden.
Xaver: Du hast doch gesagt, ich soll mich mal bei dir melden.
Yvonne: Daran kann ich mich aber nicht erinnern.
Das war aber so. Da bin ich mir ganz sicher.
Xaver: Das war aber so. Da bin ich mir ganz sicher.
Yvonne: Das hat du falsch verstanden. Nerv mich nicht.
```

**Bild 6.13** Beispielhafte Nutzung des kommandozeilenorientierten Chat-Clients



**Bild 6.14** Beispielhafte Nutzung des Chat-Clients mit grafischer Benutzeroberfläche

Bitte beachten Sie, dass die Referenzübergabe unter Umständen eine Synchronisation nötig macht. Wenn nämlich der Client eine Referenz auf eines seiner Objekte dem Server über gibt und der Server diese Referenz speichert, so dass er auch nach dem RMI-Aufruf noch Zugriff auf das Objekt des Clients hat, dann ist es möglich, dass der Client und der Server gleichzeitig auf das Objekt des Clients zugreifen. Wenn es dabei auch schreibende Zugriffe gibt, dann müssen gemäß unserer Regel aus Kapitel 2 alle lesenden und schreibenden Zugriffe synchronisiert werden. In unserem Chat-Beispiel war eine solche Synchronisation nicht notwendig, denn zum einen wird das Namensattribut nach dem Setzen im Konstruktur nicht mehr verändert. Zum anderen wird der Zugriff auf die grafische Benutzeroberfläche dadurch synchronisiert, dass ein Auftrag zur Ausgabe an den „JavaFX Application Thread“ delegiert wird.

Das Listenbeispiel des vorigen Abschnitts 6.4 könnte nun auch mit einem „Call by reference“ realisiert werden. Es wäre dann möglich, die Änderungen, die der Server an der Liste vornimmt, nach dem Methodenaufruf auf dem Client zu sehen, auch wenn die RMI-Methode void ist und keinen Wert zurückgibt. Dies ist deshalb so, weil der Server die Änderungen an der Liste durch einen RMI-Rückruf direkt auf dem Client durchführt.

Ein Objekt, das sowohl ein RMI-Objekt als auch serialisierbar ist (d. h. die Serializable- und eine RMI-Schnittstelle implementiert und aus UnicastRemoteObject abgeleitet ist), wird bei einem RMI-Methodenaufruf als Referenz übergeben. Das heißt, dass die Referenzübergabe Vorrang hat vor der Wertübergabe. Genauere Angaben hierzu folgen in Abschnitt 6.7.

## ■ 6.6 Transformation lokaler in verteilte Anwendungen

Mit RMI können nun Anwendungen aus vorhergehenden Kapiteln (parallele Anwendungen und Anwendungen mit grafischer Benutzeroberfläche) in einfacher und systematischer Weise auf mehrere Rechner verteilt werden.

### 6.6.1 Rechnergrenzen überschreitende Synchronisation mit RMI

Die Beispiele aus den Kapiteln 2 und 3 lassen sich folgendermaßen in verteilte Anwendungen umwandeln:

- Die passiven Klassen (s. hierzu Tabelle 2.4 in Abschnitt 2.12) wie z. B. die Bank oder das Parkhaus bekommen eine RMI-Schnittstelle und werden durch Ableitung aus Unicast-RemoteObject zu RMI-Klassen. Es werden dann Objekte dieser Klassen von einem RMI-Server bereitgestellt und unter einem bestimmten Namen bei der RMI-Registry registriert.
- Die aktiven Klassen, die z. B. die Bankangestellten oder die Autos repräsentiert haben, werden zu RMI-Clients. Sie beschaffen sich über Naming.lookup Zugriff auf die RMI-Objekte und können diese dann gemeinsam nutzen.

Als Beispiel betrachten wir zunächst den einfachen Semaphor aus Abschnitt 3.1.1 (Listing 3.1). Um ihn aus der Ferne benutzen zu können, brauchen wir zunächst eine RMI-Schnittstelle (s. Listing 6.28):

#### **Listing 6.28**

```
import java.rmi.*;

public interface RMISemaphore extends Remote
{
    public void p() throws RemoteException;
    public void v() throws RemoteException;
}
```

Zur Fernnutzung von Semaphoren muss die Semaphor-Klasse nun diese Schnittstelle implementieren und UnicastRemoteObject als Basisklasse haben (s. Listing 6.299):

#### **Listing 6.29**

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

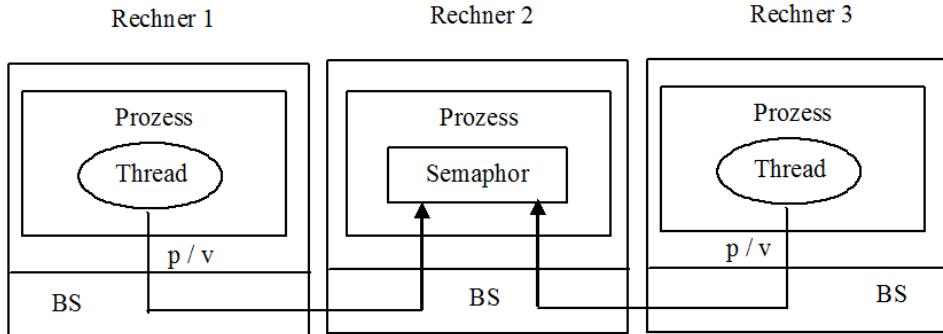
public class RMISemaphoreImpl extends UnicastRemoteObject
    implements RMISemaphore
{
    private int value;

    public RMISemaphoreImpl(int init) throws RemoteException
    {
        if(init < 0)
        {
            throw new IllegalArgumentException("Parameter < 0");
        }
        value = init;
    }

    public synchronized void p() throws RemoteException
    {
        while(value == 0)
        {
            try
            {
                wait();
            }
            catch(InterruptedException e)
            {
            }
        }
        value--;
    }

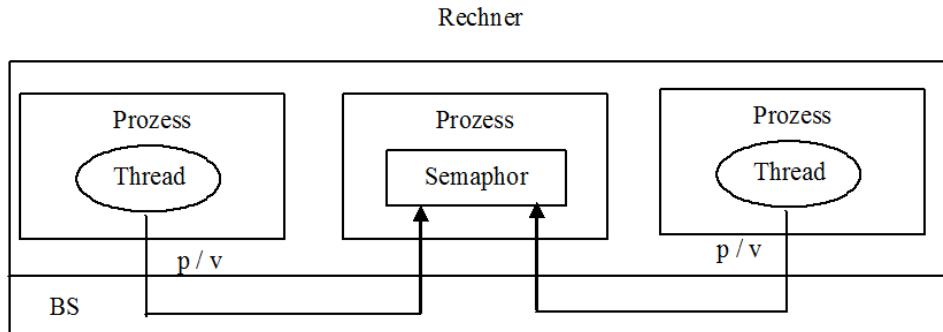
    public synchronized void v() throws RemoteException
    {
        value++;
        notify();
    }
}
```

Damit können die Beispiele für die Nutzung eines Semaphors für den gegenseitigen Ausschluss (Listing 3.2 in Abschnitt 3.1.2) und zur Herstellung von vorgegebenen Ausführungsreihenfolgen (Listing 3.3 in Abschnitt 3.1.3) auch in einer verteilten Umgebung ablaufen. Bisher besaßen die Beispiele ja den Nachteil, dass sie zur Synchronisation nur für Threads desselben Prozesses verwendet werden konnten (s. Bild 3.1 links). Durch RMI ist dies schlagartig anders; jetzt können sich auch Threads in Prozessen, die auf unterschiedlichen Rechnern laufen, synchronisieren, wie Bild 6.15 zeigt.



**Bild 6.15** Rechner- und prozessübergreifende Synchronisation durch RMI

Als Spezialfall ist es damit auch möglich, dass sich Threads in unterschiedlichen Prozessen auf einem einzigen Rechner über unseren selbst programmierten Semaphore synchronisieren (s. Bild 6.16). Ein Unterschied zur Synchronisation über einen Betriebssystem-Semaphore bleibt aber dennoch bestehen: Unser RMI-Semaphore befindet sich in einem Prozess im Benutzeraddressraum und nicht im Betriebssystemkern (vgl. Bild 6.16 mit Bild 3.1 rechts). Als Beispiel können die Threads T1 bis T5 aus Listing 3.3 in Abschnitt 3.1.3 in unterschiedlichen Prozessen und sogar auf unterschiedlichen Rechnern laufen und sich so mit Hilfe eines RMI-Semaphors synchronisieren.



**Bild 6.16** Prozessübergreifende Synchronisation durch RMI auf einem Rechner

Beachten Sie, dass in diesem verteilten Fall nicht der Client-Thread, der die Methode p aufruft, durch wait blockiert wird, falls value == 0 gilt, sondern der Thread, der stellvertretend für den Client auf dem Semaphore-Server läuft. Da dieser Stellvertreter-Thread dadurch aber zunächst nicht aus dem Methodenaufruf zurückkehrt, bleibt auch der Client-Thread hängen.

gen. Durch Aufruf der Methode v durch einen anderen Stellvertreter-Thread kann der blockierte Stellvertreter-Thread weiterlaufen, seinen Methodenaufruf beenden und eine Antwort an den Client zurücksenden, wodurch dann auch der Client weiterlaufen kann.

Ein Aufruf der RMI-Methode p kann also beliebig lange dauern. Dies wird dann zum Problem, falls RMI eine Frist kennt, nach der ein noch nicht beendeter Methodenaufruf durch Auslösen einer Ausnahme abgebrochen wird. In der Tat gibt es bei RMI eine solche Frist. Diese kann durch die Umgebungsvariable sun.rmi.transport.tcp.responseTimeout festgelegt werden. Die Angaben erfolgen in Millisekunden. Die Umgebungsvariable kann beim Starten eines Java-Prozesses in der Kommandozeile z.B. so gesetzt werden (beim Starten eines Programms aus Entwicklungsumgebungen wie Eclipse und NetBeans heraus können entsprechende Angaben gemacht werden):

```
java -Dsun.rmi.transport.tcp.responseTime=5000 ...
```

In diesem Beispiel wird die Frist auf 5 Sekunden eingestellt. Ein Wert von 0 bedeutet „unendlich“. Dies ist die Voreinstellung, so dass also im Normalfall die Umgebungsvariable nicht explizit gesetzt werden muss, damit ein RMI-Methodenaufruf beliebig lange dauern kann. Sollte die Einstellung aus irgendeinem Grund jedoch anders sein, so müsste ein Client, der die Methode p aufruft, so programmiert werden, dass er auf die Ausnahme einer Fristüberschreitung durch erneutes Aufrufen der Methode p reagiert.

## 6.6.2 Asynchrone Kommunikation mit RMI

Wie für Semaphore kann auch für Message Queues (Abschnitt 3.2) und Pipes (Abschnitt 3.3) die Benutzung aus der Ferne ermöglicht werden (die Überlegungen bezüglich Timeout, wie sie für Semaphore angestellt wurden, gelten hier in gleicher Weise). Anwendungen auf unterschiedlichen Rechnern und in unterschiedlichen Prozessen können dadurch asynchron miteinander kommunizieren. Damit ist gemeint, dass der Sender und der Empfänger einer Nachricht nicht zur gleichen Zeit verfügbar sein müssen. Das heißt: Es ist möglich, dass erst nach der Beendigung einer Anwendung, welche eine Nachricht gesendet hat, eine weitere Anwendung gestartet wird, welche die zuvor versendete Nachricht empfängt.

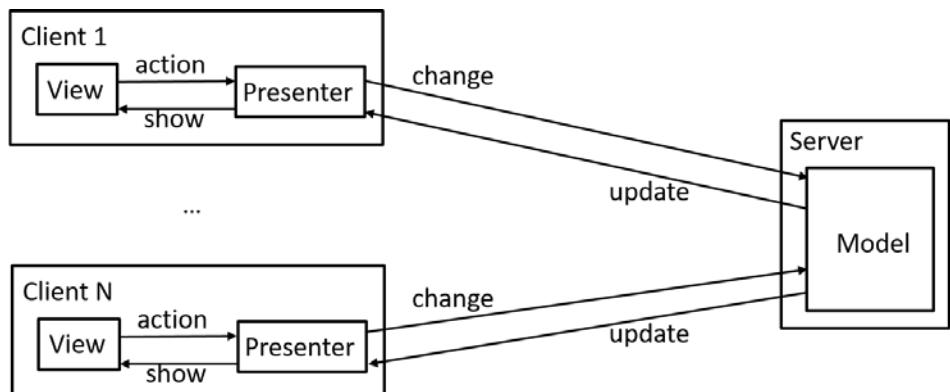
Es existieren eine ganze Reihe von frei verfügbaren und kommerziellen Implementierungen verteilter Message Queues. Diese Art von Software wird üblicherweise *Message-Oriented Middleware (MOM)* genannt. Im Gegensatz zu einer eigenen Implementierung, die leicht aus der Implementierung der Message Queue von Listing 3.7 in Abschnitt 3.2.2 gewonnen werden kann, haben MOMEmpfänger einige zusätzliche Funktionen. Dazu gehört vor allem die Unterstützung des Transaktionskonzepts und daraus folgend der Persistenz (d.h. der zuverlässigen längerfristigen Speicherung). Das bedeutet, dass Nachrichten, die an eine Message Queue gesendet und noch nicht abgeholt worden sind, noch vorhanden sein müssen, wenn der Rechner, auf dem sich die Message Queue befindet, abstürzt und danach wieder hochfährt.

### 6.6.3 Verteilte MVP-Anwendungen mit RMI

Eine lokale Anwendung mit grafischer Benutzeroberfläche, die streng gemäß des MVP-Architekturmusters strukturiert wurde, kann ebenfalls leicht in eine verteilte Anwendung überführt werden. Dies haben wir am Zähler- und am Chat-Beispiel bereits gesehen. Im Allgemeinen geht man dazu wie folgt vor:

- Für die Modellkomponente der lokalen Anwendung muss eine RMI-Schnittstelle definiert werden. Das Modell muss dann als RMI-Klasse realisiert, ein oder mehrere Objekte dieser Klasse auf einem RMI-Server zur Nutzung bereitgestellt und bei einer RMI-Registry angemeldet werden. Im Kern sollte aber an der Modellkomponente keine Code-Änderung nötig sein.
- Die View-Komponente sollte man im Idealfall unverändert in die verteilte Variante übernehmen können.
- Die Presenter-Komponente, deren Methoden bei Benutzeraktionen aufgerufen werden, hat eine Referenz auf die Modellkomponente, die dem Presenter als Parameter übergeben wird. Im lokalen Fall ist dies eine „normale“ Referenz auf das Modellobjekt. Im verteilten Fall ist dies eine Referenz auf die in der Ferne liegende RMI-Modellkomponente, die vom Client zu Beginn durch Naming.lookup beschafft und dann dem Presenter übergeben wird (also eine Referenz auf ein Stub-Objekt). Wie schon erwähnt wurde, sollten wegen der Gefahr des „Einfrierens“ der Oberfläche die RMI-Aufrufe nicht im „JavaFX Application Thread“, sondern in einem separaten Thread ausgeführt werden. In manchen Anwendungen kann es wie im Chat-Beispiel zusätzlich wünschenswert sein, dass Callback-Aufrufe vom Server an die Presenter-Komponente des Clients erfolgen, um den Client über Änderungen am Server zu informieren. Dazu muss sich der Client am Server mit einem RMI-Objekt als Parameter registrieren. Methodenaufrufe, die vom Server getriggert werden, dürfen nicht direkt Methoden der View aufrufen, da darin in der Regel auf die Benutzeroberfläche zugegriffen wird. Die Aufrufe an die View müssen über Platform.runLater an den „JavaFX Application Thread“ delegiert werden.

Das in den Bildern 4.11 und 4.12 dargestellte Zusammenspiel zwischen den M-, V- und P-Komponenten ist in Bild 6.17 auf den verteilten Fall übertragen worden.



**Bild 6.17** Prinzip einer verteilten MVP-Anwendung

Wir haben mit dem Zähler-Beispiel aus Unterabschnitt 6.2.2 und dem Chat-Beispiel aus Abschnitt 6.5 schon zwei Beispiele für verteilte MVP-Anwendungen kennengelernt. Beim Zähler-Beispiel hat allerdings noch der „Rückkanal“ (vom Server zu den Clients) gefehlt. Die Anwendung wurde nämlich so programmiert, dass sich die Anzeige nur aktualisiert, wenn man selbst eine Aktion zur Änderung des Zählers anstößt. Da man aber in einer Client-Server-Anwendung in der Regel immer von mehreren Clients ausgehen muss, könnte sich der Zähler auch durch eine Aktion eines anderen Clients verändern. In diesem Fall könnten sich die Clients beim Server ähnlich wie im Chat-Beispiel registrieren, um immer dann benachrichtigt zu werden, wenn sich der Zählerwert verändert hat.

## ■ 6.7 Dynamisches Umschalten zwischen Wert- und Referenzübergabe – Migration von Objekten

### 6.7.1 Das Exportieren und „Unexportieren“ von Objekten

Bisher sah es so aus, als würde zum Zeitpunkt der Programmierung festgelegt, ob ein Objekt durch seinen Wert oder seine Referenz als Parameter bzw. als Rückgabewert übergeben wird. Wertübergabe wurde durch Implementierung der Schnittstelle Serializable, Referenzübergabe durch Implementierung der Schnittstelle Remote und Ableitung aus der Klasse UnicastRemoteObject festgelegt. Eine genauere Betrachtung zeigt jedoch, dass die Ableitung einer Klasse aus UnicastRemoteObject nicht zwingend erforderlich für die Referenzübergabe ist. Entscheidend ist vielmehr das so genannte *Exportieren eines Objekts*, welches mit einer der folgenden Static-Methoden der Klasse UnicastRemoteObject durchgeführt werden kann.

```
public class UnicastRemoteObject extends RemoteServer
{
    public static RemoteStub exportObject(Remote obj)
        throws RemoteException {...}
    public static Remote exportObject(Remote obj, int port)
        throws RemoteException {...}
    public static Remote exportObject(Remote obj, int port,
                                    RMIClientSocketFactory csf,
                                    RMIServerSocketFactory ssf)
        throws RemoteException {...}
}
```

Die erste der Methoden namens *exportObject* mit nur einem Remote-Objekt als Parameter werden wir hier nicht näher betrachten, da diese Methode eine mit dem RMI-Compiler rmic erzeugte Stub-Klasse benötigt, was wir hier nicht verwenden wollen. Zu der dritten Variante kommen wir später (Abschnitt 6.10). Wir konzentrieren uns also zunächst auf die zweite Variante.

Durch das Exportieren wird ein Objekt, welches die Remote-Schnittstelle implementieren muss, für die Benutzung „von außen“ (d.h. von außerhalb des Prozesses) zugänglich. Es wird ein ServerSocket mit der angegebenen Portnummer geöffnet, an dem Verbindungen für die Benutzung des Objekts angenommen werden. Dazu wird ein Thread erzeugt, der an diesem ServerSocket auf eintreffende Verbindungsaufbauwünsche wartet. Die Portnummer des Servers, die bei zwei der drei Exportieren-Methoden explizit angegeben wird, wurde in Bild 6.5 mit x bezeichnet. Ferner wird beim Exportieren eine Kennung für das Objekt vergeben und das Objekt wird in eine interne Tabelle des Server-Prozesses eingetragen. Die Tabelle (eine Hash-Tabelle) ordnet der Kennung das Objekt (genauer: die lokale Referenz auf das Objekt) zu. Bitte beachten Sie, dass dieser Tabelleneintrag nichts mit dem Eintrag in der RMI-Registry zu tun hat; die Tabelle befindet sich im Server-Prozess (sie ist in Bild 6.5 nicht dargestellt). Der Tabelleneintrag, von dem hier die Rede ist, ist unbedingt notwendig, damit das Objekt ein RMI-Objekt ist und von außerhalb benutzt werden kann. Der Eintrag in der RMI-Registry ist dagegen nicht unbedingt notwendig, wie wir beim Chat-Beispiel für die Client-Objekte gesehen haben. Die Objektkennung ist notwendig, da ein Server selbstverständlich mehrere RMI-Objekte vorhalten kann (wie z.B. beim Sleep-Beispiel in Abschnitt 6.3).

Die Konstruktoren der Klasse UnicastRemoteObject führen nun alle einen solchen Export durch. Daraus folgt, dass alle Objekte von Klassen, die aus UnicastRemoteObject abgeleitet werden, exportiert sind. Da die Methoden für das Exportieren alle eine Ausnahme des Typs RemoteException werfen können und diese in den Konstruktoren der Klasse UnicastRemoteObject nicht abgefangen werden, können auch diese Konstruktoren alle eine RemoteException werfen. Dadurch wird verständlich, dass in Klassen, die aus UnicastRemoteObject abgeleitet werden, die Konstruktoren auch mit „throws RemoteException“ gekennzeichnet werden müssen, denn in den Konstruktoren wird implizit oder explizit ein Konstruktor der Basisklasse aufgerufen.

Aus den bisherigen Ausführungen folgt, dass ein RMI-Objekt also nicht von einer Klasse stammen muss, die aus UnicastRemoteObject abgeleitet ist, sondern dass das Exportieren später nachgeholt werden kann. Wenn das Objekt nun auch die Serializable-Schnittstelle implementiert, dann sieht man, dass ein Objekt zunächst als Wert und später, nach dem Exportieren, als Referenz übergeben werden kann. Die Art der Übergabe kann sich also dynamisch (d.h. zur Laufzeit ändern).

Mit der Methode *unexportObject* kann das Exportieren eines Objekts sogar wieder rückgängig gemacht werden:

```
public class UnicastRemoteObject extends RemoteServer
{
    public static boolean unexportObject(Remote obj, boolean force)
        throws NoSuchObjectException {...}
}
```

Der boolesche Parameter force gibt an, ob das Rückgängigmachen des Exportierens in jedem Fall erfolgen soll, auch wenn ein RMI-Aufruf gerade durchgeführt wird oder weitere anstehen. Dann ist als Wert des Parameters true anzugeben. Wird false angegeben, dann wird das „Unexportieren“ nur durchgeführt, falls kein RMI-Aufruf läuft und keine weiteren zur Ausführung anstehen. In diesem Fall kann man am Rückgabewert erkennen, ob die Aktion erfolgreich durchgeführt wurde oder nicht. Falls das Objekt momentan gar nicht exportiert ist, wird eine Ausnahme geworfen.

Durch das Exportieren und „Unexportieren“ kann also beliebig oft zwischen Wert- und Referenzübergabe hin- und hergeschaltet werden. Eine genauere Betrachtung zeigt weiter, dass auch bei der Referenzübergabe eine Wertübergabe stattfindet. Und zwar wird eine Referenzübergabe durch die Wertübergabe eines Stub-Objekts realisiert. Wie schon am Ende des Abschnitts 6.5 erwähnt wurde, hat die Referenzübergabe Vorrang gegenüber der Wertübergabe. Wir können diesen Sachverhalt damit nun präziser formulieren:



Für die Übergabe von Parametern bzw. Rückgabewerten beim Aufruf von RMI-Methoden gilt:

1. Wenn ein Objekt eine RMI-Schnittstelle implementiert und exportiert wurde, wird es als Referenz übergeben. Das heißt, es wird ein Stub-Objekt für das angegebene Objekt durch Serialisierung als Wert übergeben.
2. Wenn die unter 1 genannten Bedingungen nicht zutreffen, das Objekt aber die Schnittstelle Serializable implementiert, dann wird das Objekt selbst durch Serialisierung als Wert übergeben.
3. Wenn weder die Bedingungen aus 1 noch die aus 2 zutreffen, so kann das Objekt nicht übergeben werden. Es wird eine Ausnahme geworfen.

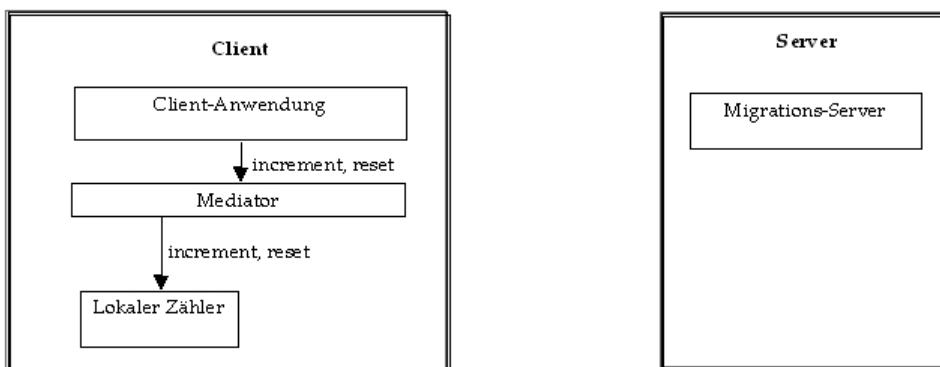
Mit diesem Wissen kann nun die in Bild 6.5 dargestellte Situation und speziell die Rolle der RMI-Registry eine Stufe genauer erklärt werden. Die Methoden der RMI-Registry (u.a. rebind und lookup) sind selbst auch RMI-Methoden. Wenn also beim Aufruf von rebind ein exportiertes Objekt als Parameter übergeben wird, wird gemäß der Regel eine serialisierte Kopie des Stubs für dieses Objekt an die RMI-Registry gesendet. Wir erinnern uns, dass es bei der Serialisierung um die Daten eines Objekts (nicht um den Programmcode der Klasse) geht. Das Stub-Objekt besitzt natürlich die Daten, um von jedem Rechner aus die Methoden des exportierten Objekts, für das es steht, in Anspruch nehmen zu können. Dazu gehört im Wesentlichen der Rechner (d.h. die IP-Adresse), auf dem sich der Server befindet, die Portnummer, an dem der Server lauscht, sowie die Objektkennung. Damit kann der Stub eine Verbindung zum Server aufbauen und dem Server durch die Objektkennung mitteilen, mit welchem Objekt er arbeiten möchte. Diese Angaben, die in Bild 6.5 in der rechten Spalte der Tabelle der RMI-Registry stehen, werden also durch ein serialisiertes Stub-Objekt repräsentiert. Durch den Aufruf von Naming.lookup wird dieser Stub nun auf den Client kopiert, denn der Stub ist selbst ein nicht exportiertes Objekt und wird deshalb als Kopie übergeben. Wenn der Client die entsprechenden Methoden aufruft, dann hat der Stub die nötigen Angaben, um den Methodenaufruf auf dem Server anzustoßen. Genau wie die RMI-Registry kann ein Client einen erhaltenen Stub in einem RMI-Methodenaufruf an andere Clients weitergeben, so dass auch diese Clients dadurch Zugriff auf das RMI-Objekt bekommen, auf die der Stub zeigt.

Anhand zweier Beispiele soll das Verständnis für die Wert- und Referenzübergabe nun weiter vertieft werden.

## 6.7.2 Migration von Objekten

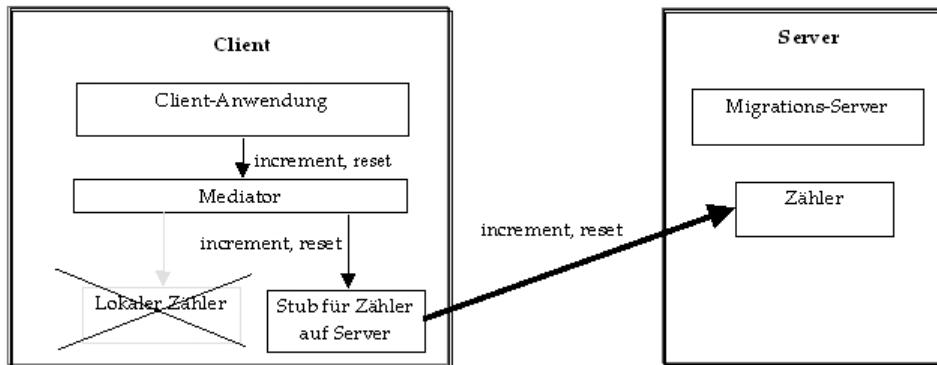
Im ersten Beispiel soll der dynamische Wechsel zwischen Wert- und Referenzübergabe demonstriert werden. Dabei wird ein Objekt zunächst lokal auf dem Client benutzt. Danach wird es auf den Server kopiert und auf dem Client gelöscht. Obwohl auf dem Server nur eine Kopie des ursprünglichen Objekts existiert und nicht das Original-Objekt selber, kann man dennoch so tun, als ob das Original-Objekt auf den Server gewandert (*migriert*) wäre. Der Zustand des Objekts bleibt bei der *Migration* durch die Serialisierung erhalten. Das Objekt kann anschließend vom Client aus weiter über RMI benutzt werden, ohne dass die Client-Anwendung etwas von der Migration erfährt. Zu einem späteren Zeitpunkt kann das Objekt wieder heimgeholt werden. Auch dieser Schritt ist für die Anwendung auf dem Client transparent (d.h. unsichtbar).

Als Beispielklasse verwenden wir wieder den Zähler aus Abschnitt 6.2 mit den Methoden increment und reset. Damit die Client-Anwendung den Wechsel nicht mitbekommt, wird ein Vermittler-Objekt zwischen die Client-Anwendung und den Zähler gesetzt. In Bild 6.18 ist die Ausgangslage dargestellt. Die Client-Anwendung hat eine Referenz auf das Vermittler-Objekt des Typs Mediator. Darin befindet sich eine Referenz auf ein lokales Zähler-Objekt. Alle Aufrufe von increment und reset werden vom Mediator an das lokale Zählerobjekt delegiert.



**Bild 6.18** Ausgangssituation des Migrationsbeispiels

Wenn die Client-Anwendung dem Mediator-Objekt befiehlt, das Objekt auf den Server zu verlagern, wird eine entsprechende RMI-Methode des Migrations-Servers ausgeführt. Das lokale Zählerobjekt wird dabei als Parameter übergeben. Da das Zählerobjekt eine Serializable-Schnittstelle besitzt, aber nicht exportiert ist, wird es auf den Server kopiert. Auf dem Server wird die Kopie exportiert. Wenn nun der empfangene Parameter zurückgegeben wird, dann wird eine Referenz in Form eines serialisierten Stub-Objekts an den Client zurückgegeben. Der Mediator ersetzt seine Referenz auf die lokale Zählerkopie durch eine Referenz auf das empfangene Stub-Objekt (s. Bild 6.19). Alle Aufrufe der Client-Anwendung werden nun vom Mediator wie zuvor delegiert. Da der Mediator aber jetzt eine Referenz auf das Stub-Objekt hat, werden die Aufrufe auf dem Server ausgeführt.



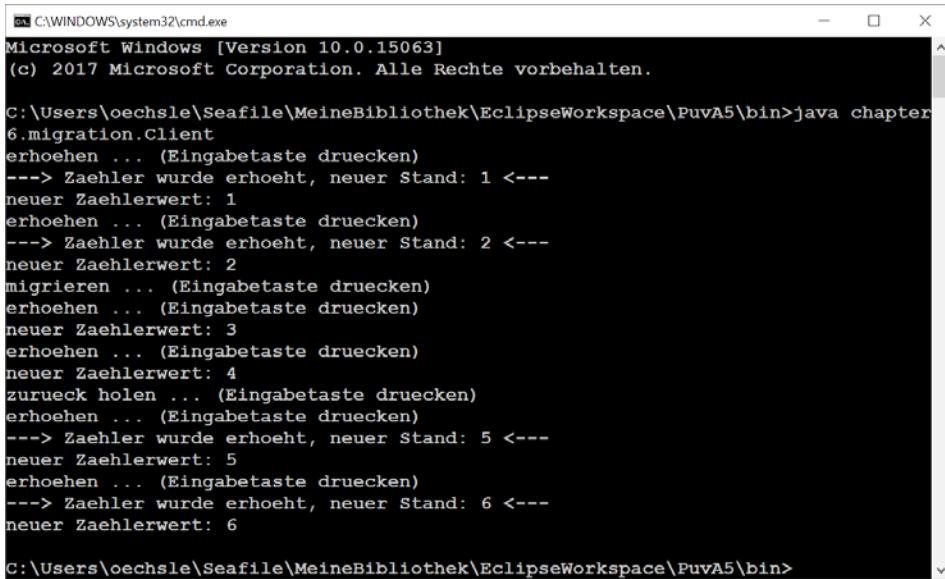
**Bild 6.19** Situation nach der Migration des Objekts vom Client auf den Server

Die Zählerklasse besitzt neben den Methoden increment und reset eine weitere Methode zum Zurückholen des migrierten Objekts. Darin wird this einfach „unexportiert“ und zurückgegeben. Dadurch, dass das Objekt am Ende der Methode nicht mehr exportiert ist, wird das Objekt als Wert zum Client zurückgeliefert. Der Mediator ersetzt die Referenz auf das Stub-Objekt durch eine Referenz auf die erhaltene Kopie; wir haben dadurch wieder die Ausgangssituation aus Bild 6.18 erreicht.

Die hier dargestellte Realisierung verzichtet aus Platzgründen auf eine grafische Benutzeroberfläche für den Client. Es ist aber ohne Weiteres möglich, die Oberfläche aus Bild 6.4 durch zwei Buttons zu erweitern, welche das Verlagern des Zähler-Objekts vom Client zum Server und umgekehrt anstoßen. Sowohl Client und Server sind also kommandozeilenorientiert. Um zu demonstrieren, wo sich das Zählerobjekt momentan befindet, werden in den Methoden increment und reset des Zählerobjekts Ausgaben gemacht. Damit diese Ausgaben von den restlichen Ausgaben leicht unterschieden werden können, sind sie mit „ $\rightarrow$ “ und „ $\leftarrow$ “ geklammert.

Bevor wir das Programm betrachten, sehen wir uns einen Probelauf des Programms an. In Bild 6.20 und Bild 6.21 sind die Ausgaben des Clients bzw. des Servers zu sehen.

Der Ablauf des Clients ist fest programmiert. Vor jedem Schritt muss der Benutzer lediglich die Eingabetaste drücken, damit es weitergeht. Zunächst wird der Zähler zwei Mal erhöht. Man sieht in Bild 6.20 anhand der Ausgabe der Increment-Methode, dass dies lokal auf dem Client ausgeführt wird (achten Sie auf die Ausgaben der Art „ $\rightarrow \dots \leftarrow$ “). Danach wird das Objekt auf den Server migriert. Nun wird der Zähler wieder zwei Mal erhöht. Man erkennt einerseits, dass diese Erhöhungen jetzt auf dem Server durchgeführt werden (s. Bild 6.21). Andererseits sieht man auch, dass das Objekt, das sich auf dem Server befindet, den Zustand vom Client (Wert 2) übernommen hat. Anschließend wird das Objekt wieder zurückgeholt. Die zwei abschließenden Increment-Aufrufe finden wieder lokal statt. Der aktuelle Wert des Servers (4) wurde auf dem Client übernommen. Die Client-Anwendung bekommt bei den 6 Aufrufen von increment die Werte 1 bis 6 zurück, unabhängig davon, wo sich der Zähler befindet.

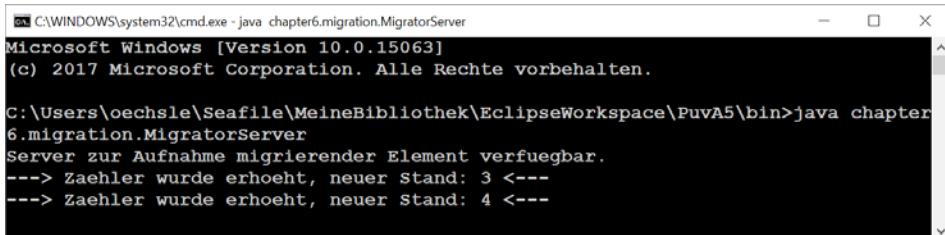


```
C:\WINDOWS\system32\cmd.exe
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\oechsle\Seafie\MeineBibliothek\EclipseWorkspace\PuvA5\bin>java chapter6.migration.Client
erhoehen ... (Eingabetaste druecken)
---> Zaehler wurde erhoeht, neuer Stand: 1 <---
neuer Zaehlerwert: 1
erhoehen ... (Eingabetaste druecken)
---> Zaehler wurde erhoeht, neuer Stand: 2 <---
neuer Zaehlerwert: 2
migrieren ... (Eingabetaste druecken)
erhoehen ... (Eingabetaste druecken)
neuer Zaehlerwert: 3
erhoehen ... (Eingabetaste druecken)
neuer Zaehlerwert: 4
zurueck holen ... (Eingabetaste druecken)
erhoehen ... (Eingabetaste druecken)
---> Zaehler wurde erhoeht, neuer Stand: 5 <---
neuer Zaehlerwert: 5
erhoehen ... (Eingabetaste druecken)
---> Zaehler wurde erhoeht, neuer Stand: 6 <---
neuer Zaehlerwert: 6

C:\Users\oechsle\Seafie\MeineBibliothek\EclipseWorkspace\PuvA5\bin>
```

**Bild 6.20** Ausgaben des Clients für das Migrationsbeispiel



```
C:\WINDOWS\system32\cmd.exe - java chapter6.migration.MigratorServer
Microsoft Windows [Version 10.0.15063]
(c) 2017 Microsoft Corporation. Alle Rechte vorbehalten.

C:\Users\oechsle\Seafie\MeineBibliothek\EclipseWorkspace\PuvA5\bin>java chapter6.migration.MigratorServer
Server zur Aufnahme migrrierender Element verfuegbar.
---> Zaehler wurde erhoeht, neuer Stand: 3 <---
---> Zaehler wurde erhoeht, neuer Stand: 4 <---
```

**Bild 6.21** Ausgaben des Servers für das Migrationsbeispiel

Wir entwickeln nun diese RMI-Anwendung auf bekannte Art und Weise:

**1. Schnittstellen definieren:** Diese Anwendung benötigt zwei RMI-Schnittstellen:

- eine Schnittstelle für den Zähler, die neben increment und reset auch eine Methode zum Zurückholen des Zählerobjekts vom Server auf den Client enthält (Listing 6.30),
- und eine Schnittstelle des Migrations-Servers, um das Zählerobjekt vom Client auf den Server zu senden und den Stub für die Kopie auf dem Server zurückzubekommen (Listing 6.31).

#### Listing 6.30

```
import java.rmi.*;

public interface Counter extends Remote
{
    public int reset() throws RemoteException;
    public int increment() throws RemoteException;
    public Counter comeBack() throws RemoteException;
}
```

**Listing 6.31**

```
import java.rmi.*;  
  
public interface Migrator extends Remote  
{  
    public Counter migrate(Counter counter) throws RemoteException;  
}
```

**2. Schnittstellen implementieren:** Die Implementierung des Zählers (Listing 6.32) erzeugt die zuvor gesehenen Ausgaben. Beim Zurückholen des Objekts wird das Exportieren rückgängig gemacht (fett gedruckt). Wichtig ist, dass durch den zweiten Parameter true das „Unexportieren“ erzwungen wird, da momentan ein RMI-Aufruf im Gange ist. Da nur der Aufrufer eine Referenz auf das exportierte Objekt erhält und somit nicht für andere Clients zur Verfügung steht, wurde darauf verzichtet, die Methoden des Zählers synchronized zu machen. Achten Sie insbesondere darauf, dass wir dieses Mal zwar die Remote-Schnittstelle implementieren, aber eben nicht von UnicastRemoteObject ableiten.

**Listing 6.32**

```
import java.io.Serializable;  
import java.rmi.RemoteException;  
import java.rmi.server.UnicastRemoteObject;  
  
public class CounterImpl implements Counter, Serializable  
{  
    private int counter;  
  
    public int reset() throws RemoteException  
    {  
        counter = 0;  
        System.out.println("--> Zaehler wurde zurueckgesetzt <--");  
        return counter;  
    }  
  
    public int increment() throws RemoteException  
    {  
        counter++;  
        System.out.println("--> Zaehler wurde erhoeht, "  
                           + "neuer Stand: " + counter  
                           + " <--");  
        return counter;  
    }  
  
    public Counter comeBack() throws RemoteException  
    {  
        UnicastRemoteObject.unexportObject(this, true);  
        return this;  
    }  
}
```

Bei der Implementierung der Migrationsschnittstelle (Listing 6.33) wird die Kopie des Zählerobjekts, die man als Parameter erhält, exportiert (fett gedruckt). Die Angabe der Portnummer 0 bedeutet, dass keine spezifische Portnummer vorgeschrieben wird, sondern dass

sich das System eine Portnummer aussuchen darf. Diese Klasse ist wie üblich aus UnicastRemoteObject abgeleitet.

#### Listing 6.33

```
import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class MigratorImpl extends UnicastRemoteObject
    implements Migrator
{
    public MigratorImpl() throws RemoteException
    {
    }

    public Counter migrate(Counter counter) throws RemoteException
    {
        try
        {
            UnicastRemoteObject.exportObject(counter, 0);
        }
        catch(RemoteException e)
        {
            System.out.println("Ausnahme: " + e);
        }
        return counter;
    }
}
```

**3. Server programmieren:** Der Server startet auch gleich die RMI-Registry (s. Listing 6.34).

#### Listing 6.34

```
import java.rmi.*;
import java.rmi.registry.*;

public class MigratorServer
{
    public static void main(String args[])
    {
        try
        {
            LocateRegistry.createRegistry(1099);
            Naming.rebind("Migrator", new MigratorImpl());
            System.out.println("Migrations-Server hochgefahren.");
        }
        catch(Exception e)
        {
            System.out.println("Ausnahme: " + e.getMessage());
        }
    }
}
```

**4. Client programmieren:** Der Client besteht neben der Anwendung (Main-Methode) aus dem zuvor beschriebenen Mediator (Listing 6.35). Dieser besitzt als Attribut eine Referenz des Typs Counter. Diese Referenz zeigt entweder auf ein lokales Objekt des Typs CounterImpl oder auf einen Stub. Alle Methodenaufrufe für increment und reset werden

ohne Fallunterscheidung an dieses Objekt weitergegeben. Beim Verschieben des Objekts auf den Server besorgt man sich zuerst eine Referenz auf den Migrations-Server und ruft die Methode migrate mit dem lokalen Zählerobjekt als Parameter auf. Der Rückgabewert wird dem Counter-Referenz-Attribut zugewiesen. Beim Zurückholen des Objekts wird die Counter-Referenz wieder überschrieben.

### **Listing 6.35**

```
import java.net.MalformedURLException;
import java.rmi.*;

public class Mediator
{
    private Counter counter;

    public Mediator(Counter counter)
    {
        this.counter = counter;
    }

    public int increment() throws RemoteException
    {
        return counter.increment();
    }

    public int reset() throws RemoteException
    {
        return counter.reset();
    }

    public void migrate(String host) throws RemoteException
    {
        try
        {
            Migrator migrator = (Migrator) Naming.lookup("rmi://" +
                host +
                "/Migrator");
            counter = migrator.migrate(counter);
        }
        catch(MalformedURLException | NotBoundException e)
        {
        }
    }

    public void comeBack() throws RemoteException
    {
        counter = counter.comeBack();
    }
}
```

Die Anwendung in der Main-Methode der Klasse Client (Listing 6.36) wartet auf eine Eingabebestätigung der Benutzerin vor jedem Schritt. Der Server, auf den das Objekt migriert wird, ist fest in das Programm „eingebrannt“ (hier „localhost“). Der Server könnte stattdessen zur Laufzeit vom Anwender erfragt werden. Denkbar wäre auch, dass das Zählerobjekt der Reihe nach auf unterschiedliche Server migriert und wieder zurückgeholt wird.

**Listing 6.36**

```
import java.io.*;  
  
public class Client  
{  
    public static void main(String args[])  
    {  
        BufferedReader sysIn = new BufferedReader(  
            new InputStreamReader(  
                System.in));  
        try  
        {  
            Counter counter = new CounterImpl();  
            Mediator mediator = new Mediator(counter);  
            int value;  
  
            System.out.print("erhoehen ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            value = mediator.increment();  
            System.out.println("neuer Zahlerwert: " + value);  
  
            System.out.print("erhoehen ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            value = mediator.increment();  
            System.out.println("neuer Zahlerwert: " + value);  
  
            System.out.print("migrieren ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            mediator.migrate("localhost");  
  
            System.out.print("erhoehen ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            value = mediator.increment();  
            System.out.println("neuer Zahlerwert: " + value);  
  
            System.out.print("erhoehen ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            value = mediator.increment();  
            System.out.println("neuer Zahlerwert: " + value);  
  
            System.out.print("zurueck holen ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            mediator.comeBack();  
  
            System.out.print("erhoehen ... "  
                + "(Eingabetaste druecken)");  
            sysIn.readLine();  
            value = mediator.increment();  
            System.out.println("neuer Zahlerwert: " + value);  
  
            System.out.print("erhoehen ... "  
                + "(Eingabetaste druecken)");  
        }  
    }  
}
```

```

        + "(Eingabetaste druecken)");
    sysIn.readLine();
    value = mediator.increment();
    System.out.println("neuer Zaehlerwert: " + value);
}
catch(Exception e)
{
    System.out.println("Ausnahme: " + e);
}
}
}
}
```

- 5. Anwendung übersetzen und ausführen:** Was passiert, wenn die Anwendung ausgeführt wird, wurde bereits erläutert (s. Bild 6.20 und Bild 6.21).

### 6.7.3 Eintrag eines Nicht-Stub-Objekts in die RMI-Registry

Am Ende des Abschnitts 6.7.1 wurde erläutert, dass Naming.rebind auch ein RMI-Aufruf ist, bei dem für ein exportiertes Objekt ein serialisiertes Stub-Objekt an die RMI-Registry übertragen und in die Tabelle der RMI-Registry eingetragen wird. Daraus folgt, dass die Kopie des Objekts selbst und nicht sein Stub in die RMI-Registry geschrieben wird, wenn man ein nicht exportiertes Objekt Naming.rebind übergibt. In Listing 6.37 wird dies ausprobiert. Es wird ein Objekt des Typs CounterImpl mit Naming.rebind bei der RMI-Registry angemeldet. Da die Klasse CounterImpl nicht aus UnicastRemoteObject abgeleitet ist, ist das CounterImpl-Objekt folglich nicht exportiert.

**Listing 6.37**

```

import java.rmi.*;

public class CounterServer
{
    public static void main(String args[])
    {
        try
        {
            Counter c = new CounterImpl();
            Naming.rebind("Counter", c);
        }
        catch(Exception e)
        {
            System.out.println("Ausnahme: " + e.getMessage());
        }
    }
}
```

Bei der Ausführung des Servers fällt auf, dass der Server im Gegensatz zu allen zuvor diskutierten Servern dieses Mal zu Ende läuft. Da das angemeldete Objekt nicht exportiert wurde, wurde kein Thread gestartet. Das Programm ist deshalb zu Ende, nachdem die Main-Methode geendet hat. Wenn man sich nun in einem Client-Programm mit Naming.lookup eine Referenz auf das angemeldete Objekt beschafft und darauf die Methoden increment und reset anwendet, erkennt man anhand der Ausgabe, dass diese Methoden auf dem Client

ausgeführt werden, denn die Kopie des Objekts aus der RMI-Registry wurde durch Naming.lookup auf den Client kopiert (es handelt sich ja um ein serialisierbares, nicht exportiertes Objekt). Es ist daher sinnvoll, dass der Server zu Ende gelaufen ist, denn dieser hat nach dem Eintragen des Objekts bei der RMI-Registry keine weiteren Aufgaben; die Methoden des Zählerobjekts werden in diesem Fall vom Client selbst ausgeführt.

## ■ 6.8 Laden von Klassen über das Netz

Bei der Diskussion des vorigen Abschnitts ging es um die Übertragung von serialisierten Objekten (Stubs und Nicht-Stubs) über das Netz. Dabei werden Daten (im Wesentlichen die aktuellen Werte der Attribute) übertragen. Im Zusammenhang mit RMI kann aber auch Programmcode (Inhalte von Class-Dateien) über das Netz übertragen werden. Ein Szenario, in dem dies z.B. für den Server nötig ist, ist das Folgende: Angenommen, es gibt eine RMI-Schnittstelle mit einer Methode m, die einen Parameter des Typs X besitzt. X sei eine Klasse. Wenn nun ein Client die Klasse Y aus X ableitet und dabei Methoden überschreibt, ein Objekt der Klasse Y erzeugt und dieses als Parameter beim Aufruf der RMI-Methode m übergibt, dann muss der Server Zugriff auf den Programmcode der Klasse Y haben, um die „richtigen“ Methoden auf den Parameter anwenden zu können. RMI sieht aus diesem Grund eine Möglichkeit vor, den nötigen Code zur Laufzeit von einem Web-Server zu laden und auszuführen. Da dies natürlich hochgradig gefährlich ist, funktioniert dies nur dann, wenn bestimmte Sicherheitseinstellungen vorgenommen werden.

Im Folgenden wird erläutert, was zu tun ist, damit der Server Programmcode des Clients nachladen kann:

- Der Client muss die benötigten Class-Dateien auf einem Web-Server zum Herunterladen bereitstellen. Die entsprechende Basis-URL muss beim Client in der Umgebungsvariable java.rmi.server.codebase gesetzt werden. Eine Umgebungsvariable kann durch die Static-Methode setProperty der Klasse System im Programmcode mit

```
System.setProperty("java.rmi.server.codebase", "...");
```

oder durch ein spezielles Kommandozeilenargument des Java-Kommandos beim Starten des Clients gesetzt werden:

```
java -Djava.rmi.server.codebase=... <Java-Klasse> <Programmarg.>
```

- Beim Server muss ein SecurityManager gesetzt werden, damit das Laden einer Klasse über das Netz möglich ist (der Standard-Security-Manager erlaubt das Laden von Klassen nur aus dem lokalen Dateisystem). Auch das Setzen des Security-Managers kann im Programmcode mit

```
System.setSecurityManager(new RMISecurityManager());
```

oder durch ein Kommandozeilenargument erfolgen:

```
java -Djava.security.manager <Java-Klasse> <Programmargumente>
```

Ein Security-Manager benötigt eine Policy-Datei, in der die benötigten Rechte entsprechend gesetzt sind. Da auf Einzelheiten des Java-Sicherheitskonzepts nicht eingegangen werden kann, wird hier eine Policy-Datei angegeben, die alle Rechte gewährt. Diese Einstellung ist natürlich äußerst gefährlich und sollte nur zum Testen in einem abgeschlossenen Netz verwendet werden:

```
grant
{
    permission java.security.AllPermission;
};
```

Die zu benutzende Policy-Datei muss spezifiziert werden. Sie kann u.a. als Kommandozeilenargument beim Starten des Servers angegeben werden:

```
java -Djava.security.policy=<Dateiname> <Java-Klasse> <Prog.arg.>
```

Die Rollen von Client und Server sind vertauscht, wenn das einführende Szenario von einem Parameter des Typs X auf den Rückgabetyp übertragen wird. Oder, wenn das Beispiel so belassen wird, kehren sich die Rollen von Client und Server auch dann um, wenn es sich bei dem RMI-Aufruf um einen Callback vom Server auf den Client handelt. In einer konkreten Anwendung kann es sein, dass sowohl der Client als auch der Server Klassen über das Netz nachlädt.

## ■ 6.9 Realisierung von Stubs und Skeletons

Bis Java-Version 1.2 wurden durch den RMI-Compiler (Kommando rmic) für jede Anwendung spezifische Stub- und Skeleton-Klassen erzeugt. Eine Skeleton-Klasse alter Art kann man sich vorstellen wie den parallelen TCP-Server für das Zähler-Beispiel (s. Listing 5.13 und 5.14 aus Abschnitt 5.6.2): Wenn der String „increment“ empfangen wird, dann wird die Methode increment aufgerufen, wenn aber der String „reset“ empfangen wird, dann muss die Methode reset aufgerufen werden. Den Stub kann man sich für das Zählerbeispiel als eine Klasse mit den Methoden increment und reset vorstellen. In der Methode increment wird über eine zuvor aufgebaute TCP-Verbindung der String „increment“ verschickt, dann wird auf der TCP-Verbindung auf eine Antwort gewartet, die Antwort wird in eine Zahl gewandelt und zurückgegeben. Die Methode reset ist ähnlich, nur wird stattdessen der String „reset“ über die TCP-Verbindung an den Server geschickt.

Wenn sich nun die RMI-Schnittstelle ändert, muss der Code sowohl des Skeletons als auch des Stubs geändert werden: In der Skeleton-Klasse muss auf andere Kommandos reagiert werden, und es müssen andere Methodenaufrufe im Programmcode stehen. Die Stub-Klasse muss andere Methoden haben, und es müssen andere Daten verschickt und als Rückgabewert empfangen werden.

Seit Java-Version 1.2 ist die Erzeugung von Skeleton-Klassen und seit Java-Version 5 die Erzeugung auch von Stub-Klassen nicht mehr nötig. Warum dies so ist, wird im Folgenden grob erläutert.

## 6.9.1 Realisierung von Skeletons

Seit Java-Version 1.2 gibt es eine generische Skeleton-Klasse, die für alle RMI-Anwendungen verwendet werden kann. Dabei wird die so genannte *Reflection-Programmierschnittstelle* der Java-Klassenbibliothek benutzt. Damit kann man sich z. B. von jedem beliebigen Objekt das dazugehörige Class-Objekt geben lassen. Von diesem Class-Objekt wiederum kann man sich durch Angabe des Namens der Methode als String und der Typen der Parameter ein Method-Objekt beschaffen. Mit diesem Method-Objekt kann man dann die entsprechende Methode auf das entsprechende Objekt anwenden. Das folgende Programmfragment fasst diesen Ablauf auf dem Server zusammen:

```
//Objektkennung wird vom Client gesendet, daraus kann der Server
//über eine Hash-Tabelle eine Referenz auf das richtige RMI-Objekt
//beschaffen:
Object obj = ...;

//Methodename wird vom Client gesendet:
String methodName = ...;

//Parameterliste wird ebenfalls vom Client gesendet:
Object[] actualParameterList = ...;

//aus Parameterliste wird Parametertypenliste gebildet:
Class[] formalParameterList = new Class[actualParameterList.length];
for(int i = 0; i < formalParameterList.length; i++)
{
    formalParameterList[i] = actualParameterList[i].getClass();
}

//Methode wird herausgesucht:
Class c = obj.getClass();
Method m = c.getMethod(methodName, formalParameterList);

//Methode m wird auf obj mit actualParameterList angewendet:
Object result = m.invoke(obj, actualParameterList);
```

Bitte beachten Sie, dass dies eine stark vereinfachte Darstellung ist. Wenn der Typ eines Parameters der aufzurufenden Methode z. B. X ist und als konkreter Parameter ein Objekt einer Klasse Y, die aus X abgeleitet ist, angegeben wird, dann sollte dieser Aufruf eigentlich problemlos funktionieren. In obigem Programmcode würde aber getMethod die passende Methode nicht finden. Deshalb wird das Finden der richtigen Methode in der Realität anders realisiert.

## 6.9.2 Realisierung von Stubs

Seit Java-Version 5 werden die Stub-Klassen dynamisch mit Hilfe der so genannten *Dynamic-Proxy-Schnittstelle* erzeugt. Damit kann unter Angabe einer Liste von Schnittstellen (beschrieben durch ein Class-Feld) dynamisch zur Laufzeit ein Class-Objekt erzeugt werden. Die von diesem Class-Objekt beschriebene Klasse, welche die angegebenen Schnittstellen implementiert, existiert nur in der Java Virtual Machine im Hauptspeicher; es existiert

keine entsprechende Class-Datei auf der Festplatte wie bei anderen „normalen“ Klassen. Hat man ein Class-Objekt, kann man sich unter Angabe einer Parameterliste (angegeben durch ein Class-Feld) einen Konstruktor geben lassen (dies ist sehr ähnlich zu `getMethod` oben). Der Konstruktor wird durch ein `Constructor`-Objekt repräsentiert (entspricht dem `Method`-Objekt oben). Durch Anwendung der Methode `newInstance` kann mit Angabe einer Parameterliste ein Objekt der dynamisch generierten Klasse erzeugt werden. Bitte machen Sie sich klar, dass wir zwar eine Klasse erzeugt haben, die alle Methoden aller angegebenen Schnittstellen besitzt. Wir können aber keinen Programmcode für diese Methoden angeben. Die Methoden funktionieren nun so, dass bei jedem Aufruf einer dieser Methoden eine Methode des im Konstruktor angegebenen `InvocationHandler`-Objekts aufgerufen wird. Dieses Objekt implementiert die Schnittstelle `InvocationHandler`:

```
public interface InvocationHandler
{
    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable;
}
```

Für die dynamische Stub-Implementierung existiert eine Implementierung der `InvocationHandler`-Schnittstelle, bei der im Wesentlichen die Methode `invoke` so realisiert ist, dass der aus dem Methodenparameter gelesene Methodename zusammen mit der Argumentliste `args` an den RMI-Server gesendet wird. Das könnte dann z.B. so aussehen:

```
public class RemoteObjectInvocationHandler extends RemoteObject
    implements InvocationHandler
{
    //Attribute, die nötig sind zur erfolgreichen Kommunikation
    //mit fernem RMI-Objekt: Rechner, Port, Objektkennung
    ...
    //Attribut für Verbindung zum RMI-Server:
    ...

    //Konstruktor oder Setter-Methoden zum Setzen der Attribute:
    ...

    public Object invoke(Object proxy, Method method, Object[] args)
        throws Throwable
    {
        //falls noch keine Verbindung zum RMI-Server existiert,
        //Verbindung aufbauen:
        ...

        //Methodename und Argumente mit Objektkennung an RMI-Server
        //über aufgebaute Verbindung senden:
        ...

        //auf Antwort warten:
        ...

        //falls Ausnahme signalisiert wird, Ausnahme werfen:
        if(...)
        {
            throw new ...;
        }
    }
}
```

```

        //sonst erhaltenen Rückgabewert zurückgeben:
        return ...;
    }
}

```

Zum Verständnis des folgenden Programmfragments muss noch erklärt werden, dass eine RMI-Klasse wie in unseren bisherigen Beispielen nicht nur eine einzige RMI-Schnittstelle, sondern mehrere RMI-Schnittstellen implementieren kann. Der Stub muss folglich auch alle diese Schnittstellen implementieren. Je nachdem, aus welcher Schnittstelle die Methode stammt, die ein Client auf den Stub anwenden will, muss jeweils auf die richtige RMI-Schnittstelle gecastet werden. Damit sollte der prinzipielle Ablauf für die dynamische Erzeugung der Stub-Klasse und des Stub-Objekts problemlos verständlich sein:

```

//Liste der zu implementierenden RMI-Schnittstellen aus RMI-Objekt
//bestimmen, für das ein Stub-Objekt generiert werden soll;
//dazu können mit der Reflection-Programmierschnittstelle alle
//Schnittstellen, die die Klasse des Objekts implementiert, gelesen
//werden; die RMI-Schnittstellen sind dann diejenigen,
//die entweder direkt oder indirekt aus Remote abgeleitet sind:
Class[] interfaces = ...;

//entsprechendes Class-Objekt dynamisch generieren,
//welches die RMI-Schnittstellen implementiert:
Class stubClass = Proxy.getProxyClass(..., interfaces);

//Constructor mit InvocationHandler-Parameter geben lassen:
Class[] formalParameterList = new Class[1];
formalParameterList[0] = InvocationHandler.class;
Constructor constructor =
    stubClass.getConstructor(formalParameterList);

//RemoteObjectInvocation-Objekt erzeugen
//(Klasse in Package java.rmi.server):
RemoteObjectInvocationHandler handler =
    new RemoteObjectInvocationHandler(...);

//Rechner, Port und Objektkennung für RMI-Objekt der Ferne
// in handler setzen:
...

//Stub-Objekt erzeugen, bei dem alle Methodenaufrufe
//an das Objekt handler delegiert werden:
Object[] actualParameterList = new Object[1];
actualParameterList[0] = handler;
Object stubObject = constructor.newInstance(actualParameterList);

```

## ■ 6.10 Verschiedenes

Auch wenn das Thema RMI verhältnismäßig ausführlich besprochen wurde, so konnten dennoch nicht alle Themen vertiefend behandelt werden. Hier sind einige weitere Aspekte, die nur angerissen werden können:

- **Sicherheit durch Verschlüsselung:** Wenn mit RMI sicherheitskritische Daten übermittelt werden, dann ist es verhältnismäßig einfach, diese Daten zu verschlüsseln. Wie in Abschnitt 6.7.1 schon beschrieben wurde, besitzt die Klasse `UnicastRemoteObject` auch eine Export-Methode mit Parametern des Typs `RMIClientSocketFactory` und `RMIServerSocketFactory`. Wenn man das Exportieren durch Ableitung aus `UnicastRemoteObject` realisieren möchte, gibt es auch die Möglichkeit, einen Konstruktor der Basisklasse `UnicastRemoteObject` mit einer Portnummer und den beiden Factory-Argumenten zu nutzen. Die Factory-Objekte werden benutzt, um Socket-Objekte und ServerSocket-Objekte für die RMI-Kommunikation zu erzeugen (Methoden `createSocket` bzw. `createServerSocket`). Standardmäßig wird die Klasse `RMISocketFactory` dafür benutzt, wenn die Factory-Klasse nicht explizit angegeben wird. Es ist aber auch möglich, eigene Implementierungen der Schnittstellen `RMIClientSocketFactory` und `RMIServerSocketFactory` zu nutzen. Für sicherheitskritische Anwendungen befinden sich in der Java-Klassenbibliothek bereits `Socket`- und `ServerSocket`-Factory-Klassen namens `SslRMIClientSocketFactory` und `SslRMIServerSocketFactory`, welche eine mit *SSL (Secure Socket Layer)* verschlüsselte Datenübertragung gewährleisten.
- **RMI-Aktivierung:** Wenn ein RMI-Server relativ selten benutzt wird, dann kann es ressourcenschonend sein, wenn dieser Server erst bei Bedarf gestartet und nach längerer Nichtnutzung wieder beendet wird. Ferner wäre es praktisch, wenn ein RMI-Server nach einem Absturz automatisch neu gestartet werden würde. Diese Wünsche werden durch das so genannte Aktivierungskonzept von RMI erfüllt. Mit der Java-Laufzeitumgebung erhält man einen Aktivierungsserver (ausführbares Programm `rmid`). Die selbst programmierten Server melden nun lediglich am Aktivierungsserver, den man zuvor gestartet haben muss, aktivierbare RMI-Klassen an. Durch die Anmeldung erhalten sie spezielle Objekte, die sie in die RMI-Registry mit `Naming.bind` oder `Naming.rebind` eintragen können. Die RMI-Server erzeugen typischerweise selbst keine RMI-Objekte. Sie laufen folglich wie der RMI-Server aus Abschnitt 6.7.3 zu Ende. Wenn ein Client das in der RMI-Registry eingetragene Objekt mit `Naming.lookup` abruft und eine entsprechende Methode aufruft, dann wendet sich dieser spezielle Stub an den Aktivierungsserver, der daraufhin automatisch einen entsprechenden RMI-Server mit RMI-Objekten startet. Der Aktivierungsserver `rmid` verwendet übrigens eine Log-Datei, um alle angemeldeten Aktivierungen zu speichern. Beim Neustart des Aktivierungsservers werden alle zuvor existierenden Aktivierungen aus der Datei gelesen, so dass diese wieder gültig sind. Ein nochmaliges Aufrufen der Programme zur Einrichtung der Aktivierungen ist nicht nötig. Will man diesen Mechanismus außer Kraft setzen, so muss die Log-Datei vor dem Neustart des Aktivierungsservers gelöscht werden.
- **Verteilte Abfallsammlung (Distributed Garbage Collection):** Innerhalb einer Java Virtual Machine werden nicht mehr referenzierte Objekte durch Abfallsammlungs-Threads eingesammelt und gelöscht. Mit RMI hat man nun rechnerübergreifende Referenzen, die bei der Abfallsammlung berücksichtigt werden müssen. Ein einfacher Referenzzähler

würde das Problem nicht lösen, denn ein Stub, der für ein Objekt ausgegeben wird, kann von den Stub-Nutzern beliebig oft kopiert und weiter verteilt werden (die RMI-Registry führt bei jedem Aufruf der Methode lookup ein solches Kopieren des Stubs durch, aber auch jeder andere, der einen Stub besitzt, kann diesen beliebig oft kopieren). Deshalb wird die Abfallsammlung für RMI-Objekte so realisiert, dass jeder existierende Stub periodisch ein Signal an den Prozess sendet, auf dem sich das RMI-Objekt befindet. Der RMI-Server weiß dann, welche Referenzen es neben den lokal vorhandenen in anderen Prozessen auf ein Objekt gibt. Wenn dieses Signal eine gewisse Zeit lang ausbleibt, wird die dazugehörige Referenz als nicht mehr existent angenommen. Falls es weder lokale noch ferne Referenzen auf ein Objekt gibt, kann es gelöscht werden. Bitte beachten Sie, dass ein RMI-Objekt damit nicht gelöscht wird, solange es in der RMI-Registry eingetragen ist, denn in der RMI-Registry befindet sich damit ein Stub, der periodisch die Existenz seiner Referenz dem RMI-Objekt mitteilt.

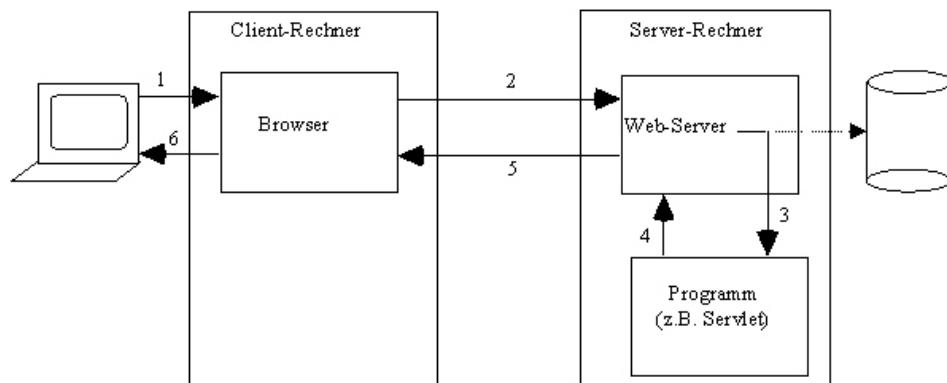
## ■ 6.11 Zusammenfassung

Nachdem wir das Prinzip von RMI und ein Kochrezept für die Entwicklung von RMI-Anwendungen kennen gelernt und an einem ersten Beispiel demonstriert haben, wurden drei Aspekte von RMI anhand je eines Beispiels besprochen: implizite Parallelität bei RMI-Aufrufen, Wertübergabe und Referenzübergabe von Parametern bzw. Rückgabewerten von Methoden. Mit grundlegenden Kenntnissen über RMI ausgestattet, wurde dann erläutert, wie zuvor besprochene lokale Anwendungen systematisch in verteilte Anwendungen überführt werden können. Eine herausragende Rolle spielt dabei das MVC-Entwurfsmuster. Anschließend wurde das Thema RMI vertieft, indem die Realisierung der Wert- und Referenzübergabe genauer unter die Lupe genommen wurde. Dabei wurde deutlich, dass es durch das Exportieren und „Unexportieren“ von Objekten möglich ist, dynamisch zwischen Wert- und Referenzübergabe umzuschalten. Demonstriert wurde diese Möglichkeit durch ein Objekt, das von einem Rechner auf einen anderen Rechner verlagert werden kann, dabei aber der Nutzer des Objekts von dieser Migration nichts mitbekommt. Bei dem Thema Wert- und Referenzübergabe geht es um den Transport von Objekten (Stub- und Nicht-Stub-Objekten) mit Hilfe der Serialisierung, d. h. um den Transport von Daten über das Netz. Im weiteren Verlauf des Kapitels wurde dann auch besprochen, dass es Sinn machen kann, Programmcode über das Netz zu laden. Es wurde erläutert, welche Vorbereitungen getroffen werden müssen, damit das dynamische Laden von Klassen über das Netz automatisch vom RMI-Laufzeitsystem durchgeführt wird. Abschließend wurden einige weitere Aspekte von RMI angesprochen, die leider nicht vertiefend behandelt werden konnten.

Ich hoffe, dass ich Sie davon überzeugen konnte, dass mit RMI eine ausgereifte, elegante, mächtige und in sich geschlossene Technik für die Entwicklung eigenständiger Client-Server-Anwendungen in Java bereitsteht. Es gibt jedoch verteilte Anwendungen, bei denen RMI nicht eingesetzt werden kann, weil als Benutzerschnittstelle ein Brower verwendet werden soll, der das HTTP-Protokoll benutzt. Wir wenden uns im Folgenden derartigen webbasierten Anwendungen zu.

# Webbasierte Anwendungen mit Servlets und JSF

Nachdem in Kapitel 5 und 6 eigenständige Client-Server-Anwendungen behandelt wurden, geht es in diesem Kapitel um webbasierte Anwendungen. Damit ist gemeint, dass die Client-Seite durch einen Web-Browser wie z.B. Firefox oder Internet Explorer realisiert wird. Auf Server-Seite wird ein Web-Server wie z.B. Apache Tomcat verwendet, der um selbst entwickelte Zusatzprogramme erweitert werden kann. Bild 7.1 zeigt den Ablauf bei der Nutzung dieser Zusatzprogramme.



**Bild 7.1** Prinzip dynamisch generierter Webseiten

Im ersten Schritt fordert eine Benutzerin eine Webseite an, z.B. indem sie eine entsprechende URL in die Adresszeile des Browser eingibt und die Return-Taste drückt, oder indem sie einen Link in einem bereits geladenen Dokument anklickt. Der Browser baut daraufhin eine TCP-Verbindung zu dem Web-Server auf, dessen Adresse bzw. Rechnername der URL entnommen werden. Über diese Verbindung wird mit Hilfe des HTTP-Protokolls das von der Benutzerin gewünschte Dokument angefordert (Schritt 2 in Bild 7.1). Der Web-Server analysiert die Anfrage. Im Normalfall ist das angeforderte Dokument eine HTML-Seite, die der Web-Server von der Festplatte liest (in Bild 7.1 gestrichelt dargestellt). Durch spezielle Konfigurationseinstellungen auf dem Web-Server kann der Server aber auch veranlasst werden, bei der Anforderung eines Dokuments mit einem bestimmten Namen oder einem bestimmten Namensmuster diese Anforderung an ein Programm (z.B. ein Servlet) weiterzuleiten (Schritt 3 in Bild 7.1). Das Programm produziert nun eine Webseite und leitet diese an den

Web-Server weiter (Schritt 4). Der Web-Server schickt das von dem Programm produzierte Dokument mit Hilfe des HTTP-Protokolls an den Browser zurück (Schritt 5), der es dann auf dem Bildschirm anzeigt (Schritt 6).

In diesem Kapitel beschäftigen wir uns mit der Entwicklung von Servlets. Dazu ist aber ein gutes Verständnis des HTTP-Protokolls nötig.

## ■ 7.1 HTTP und HTML

In Abschnitt 5.1 wurden das Schichtenmodell sowie die beiden Transportprotokolle UDP und TCP besprochen. Im restlichen Teil von Kapitel 5 wurden dann selbst definierte Anwendungsprotokolle verwendet. So wurde z. B. in Abschnitt 5.3 (Bild 5.4) ein Protokoll definiert, bei dem in UDP-Datagrammen die Strings „increment“ oder „reset“ vom Client an den Server geschickt werden, worauf der Server mit einer als String codierten Zahl antwortet. In Abschnitt 5.5 wurde dann dieses Protokoll auf TCP übertragen, wobei wegen der Datenstromorientierung von TCP alle Strings durch ein Newline-Symbol abgeschlossen wurden. In Kapitel 6 wurde auf das von RMI benutzte Protokoll nicht näher eingegangen. Aber natürlich ist auch für RMI ein Protokoll auf der Schicht 5 definiert, mit dem festgelegt wird, welche Daten in welcher Reihenfolge und welcher Codierung bei einem Methodenaufruf (Objektkennung, Methodename, Parameterliste usw.) bzw. bei einer Methodenrückkehr übermittelt werden. In diesem Abschnitt betrachten wir nun das *HTTP-Protokoll*. HTTP steht für *HyperText Transfer Protocol*. Wie die von uns in Kapitel 5 selbst definierten Protokolle ist HTTP auch ein *ASCII-Protokoll* im Gegensatz beispielsweise zum RMI-Protokoll.

### 7.1.1 GET

Betrachten wir zu Beginn ein kleines Beispiel. Angenommen, es soll von einem Browser folgende *URL (Universal Resource Locator)* angefordert werden:

```
http://www.hochschule-trier.de:8080/puva/hallo.html
```

Die URL gibt das Protokoll (HTTP), den Server (www.hochschule-trier.de) und die Portnummer des Servers (hier 8080, ohne Angabe einer Portnummer wird 80 benutzt) sowie den Pfadnamen des Dokuments an (/puva/hallo.html). Der hier noch häufiger vorkommende Name puva steht übrigens für „parallele und verteilte Anwendungen“.

Es muss zuerst eine TCP-Verbindung zum Rechner www.hochschule-trier.de und der Portnummer 8080 aufgebaut werden (dies kann z. B. mit TELNET durch Eingabe des Kommandos

```
telnet www.hochschule-trier.de 8080
```

erreicht werden). Im einfachsten Fall kann das Dokument durch folgende *HTTP-Anfrage* (Request) angefordert werden (z. B. durch Eingabe des folgenden Textes in TELNET):

```
GET /puva/hallo.html HTTP/1.0
```

Achtung: Es ist ganz wichtig, dass nach der *GET-Zeile* eine Leerzeile folgt. Wenn man das Beispiel mit TELNET ausführt, muss nach Eingabe des Textes zwei Mal die Return-Taste gedrückt werden, einmal zum Abschließen der GET-Zeile und einmal zur Eingabe einer Leerzeile (Entsprechendes gilt, falls man die Anforderung des Dokuments mit Hilfe der Socket-Schnittstelle programmiert). Die zusätzliche Leerzeile ist notwendig, da der HTTP-GET-Request aus beliebig vielen Zeilen bestehen kann (siehe unten). Mit der Leerzeile wird das Ende angezeigt.

Die Antwort des Web-Servers sieht so aus:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Accept-Ranges: bytes
ETag: W/"128-1506592237563"
Last-Modified: Thu, 28 Sep 2017 09:50:37 GMT
Content-Type: text/html
Content-Length: 128
Date: Thu, 28 Sep 2017 09:59:50 GMT
Connection: close

<html>
<head>
<title>Hallo Welt</title>
</head>
<body>
<h1>Hallo Welt</h1>
<p>Herzlich willkommen!</p>
</body>
</html>
```

Die *HTTP-Antwort* (Response) besteht aus einem Kopf (Header) und einem Datenteil. Beide sind durch die erste vorkommende Leerzeile voneinander getrennt (weitere im Datenteil vorkommende Leerzeilen haben keine Bedeutung mehr, sondern sind normale Bestandteile der Daten). Die erste Zeile des Kopfs gibt das benutzte Protokoll (HTTP/1.1) und den Status als Zahl (200) und Text (OK) an. Die Statuscodes sind immer dreistellig. Beginnt die erste Ziffer mit 2, ist alles in Ordnung. Danach folgen beliebig viele Zeilen der Form:

Name: Wert
------------

Hier wird z.B. das letzte Änderungsdatum des Dokuments (Last-Modified), der Typ des Datenteils (Content-Type, hier HTML), die Größe des Datenteils (Content-Length, hier 128 Bytes) und das aktuelle Datum mit Uhrzeit (Date) angegeben. Nach der ersten Leerzeile folgt dann das HTML-Dokument. Unmittelbar nach dem Senden der Antwort schließt der Server von sich aus die TCP-Verbindung (wenn man das Beispiel mit TELNET durchführt, sieht man aus diesem Grund die Ausgabe „Verbindung zu Host verloren.“). Dass der Server die Verbindung direkt nach dem Senden der Antwort schließen wird, kündigt er im Kopf durch „Connection: close“ an (siehe oben).

Solche einfache HTML-Seiten wie die soeben abgerufene gibt es heute nur noch als Beispiele in Lehrbüchern. Im realen Einsatz enthalten Webseiten in der Regel viele andere Elemente (z.B. Bilder, auf deren Dateinamen durch IMG-Tags im HTML-Text verwiesen wird). Um die komplette Webseite anzuzeigen, muss der Web-Browser dann für jedes im HTML-Text enthaltene IMG-Tag die entsprechende Bild-Datei durch einen erneuten HTTP-

Request vom Web-Server anfordern. Da dazu bei HTTP/1.0 jedes Mal eine erneute TCP-Verbindung aufgebaut werden muss, und da das Auf- und Abbauen nicht ohne Aufwand möglich ist, wurde in HTTP/1.1 folgende Verbesserung integriert: Wenn der Browser in seiner Anfrage mitteilt, dass er das HTTP/1.1-Protokoll unterstützt, dann hält der Server nach dem Senden der Antwort die Verbindung noch einige Sekunden geöffnet. In dieser Zeit kann der Browser eine erneute Anfrage senden, ohne eine neue Verbindung aufzubauen zu müssen. Wird die Verbindung für einige Sekunden vom Browser nicht mehr benutzt, dann schließt der Server die Verbindung.

Man kann dies mit TELNET versuchen, indem man folgende Zeile an den Web-Server sendet:

```
GET /puva/hallo.html HTTP/1.1
```

(Bitte wieder auf die zusätzliche Leerzeile achten!)

Der Server antwortet daraufhin nicht wie erhofft. Die erste Zeile der Server-Antwort lautet:

```
HTTP/1.1 400 Bad Request  
...
```

Statuscodes, deren erste Ziffer 4 oder 5 ist, weisen immer auf einen Fehler hin. Wie kann dieser Fehler trotz der Nutzung von HTTP/1.1 vermieden werden?

Wie eine HTTP-Antwort kann auch eine HTTP-Anfrage mehrere Zeilen der Form

```
Name: Wert
```

enthalten. Verwendet man HTTP/1.1, so muss die Anfrage mindestens eine weitere Zeile enthalten, und zwar eine so genannte Host-Zeile:

```
GET /puva/hallo.html HTTP/1.1  
Host: www.hochschule-trier.de:8080
```

Diese Zeile dient zur Unterstützung eines Servers, der z.B. den Web-Auftritt mehrerer Firmen beinhaltet. Angenommen, man will die Webseiten für Firmen ABC und XYZ auf einem einzigen Server halten. Die Startseiten für die beiden Firmen sollen unter folgenden URLs erreichbar sein: <http://www.abc.de/index.html> und <http://www.xyz.com/index.html>.

Die Rechnernamen [www.abc.de](http://www.abc.de) und [www.xyz.com](http://www.xyz.com) werden beide auf die IP-Adresse des Web-Servers abgebildet. In der GET-Anforderung wird dann nur index.html angefordert. Der Server hat damit keine Möglichkeit zu unterscheiden, ob er die Startseite der Firma ABC oder der Firma XYZ liefern soll.

Geben wir also als Protokoll HTTP/1.1 und die zusätzliche Host-Zeile an, dann erhalten wir eine Antwort mit dem OK-Status 200 wie zuvor. Allerdings fehlt in der Antwort die Zeile „Connection: close“. Folglich schließt auch der Web-Server die Verbindung nicht sofort nach dem Senden der Antwort. Der Client kann somit auf derselben Verbindung weitere Requests an den Server senden und muss nicht jedes Mal eine neue Verbindung aufbauen. Dies ist wie beschrieben dann nützlich, wenn in einer Web-Seite mehrere Bilder enthalten sind. Nach dem Empfang der HTML-Daten analysiert der Browser diese Seite und fordert über dieselbe Verbindung die noch fehlenden Bilder an. Erst, wenn sich eine gewisse Zeit lang nichts mehr tut, schließt der Server die Verbindung.

Ein „echte“ HTTP-Anfrage z. B. vom Firefox-Browser sieht wie folgt aus (die Zeile User-Agent wurde dabei gekürzt, natürlich ist auch hier die letzte Zeile eine Leerzeile):

```
GET /Servlets/hallo.html HTTP/1.1
Host: www.hochschule-trier.de:8080
User-Agent: Mozilla/5.0 ... Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1
```

Der Browser teilt u. a. seine Sprache (de) und die von ihm akzeptierten Datenformate mit (z. B. text/html). Zusätzlich wird zur Angabe des Protokolls HTTP/1.1 betont, dass der Client wünscht, dass der Server die Verbindung noch eine Zeit lang geöffnet lässt (Connection: Keep-Alive).

## 7.1.2 Formulare in HTML

Bei webbasierten Anwendungen werden aber nicht nur Dokumente angefordert, sondern der Benutzer übermittelt auch Daten an den Web-Server, z. B. um einen Suchbegriff einer Suchmaschine mitzuteilen, um Start, Ziel und Uhrzeit an ein Fahrplanauskunftssystem zu senden, oder um die eigene Anschrift bei der Bestellung von Waren zu übertragen. Die Benutzerin gibt diese Daten in der Regel in ein Formular ein. In Bild 7.2 ist ein Beispiel für ein Formular zu sehen, das bereits ausgefüllt wurde.

The screenshot shows a web browser window with the title 'Beispiel für Formular'. The address bar displays 'localhost:8080/puva/'. The main content area contains a form titled 'Ein Beispiel für ein Formular' with the following fields:

- Teebestellung**
- Geben Sie Ihren Namen an:
- Geben Sie Ihre Kreditkartennummer an:
- Welchen Tee wollen Sie:  Assam  Darjeeling  Ceylon
- Eilbestellung:
- Zum Zurücksetzen aller Eingaben:
- Zum Bestellen:

**Bild 7.2** Beispiel eines Formulars in einer Webseite

Der dazugehörige HTML-Quelltext für das Formular sieht so aus:

```
<form method="get" action="Tee">
<p>Geben Sie Ihren Namen an:
<input type="text" name="Besteller" size="40"/></p>
<p>Geben Sie Ihre Kreditkartennummer an:
<input type="password" name="Kreditkartennummer" size="20"/></p>
<p>Welchen Tee wollen Sie:
<input type="radio" name="Teesorte" value="Assam"/>Assam
<input type="radio" name="Teesorte" value="Darjeeling"/>Darjeeling
<input type="radio" name="Teesorte" value="Ceylon"/>Ceylon</p>
<p>Eilbestellung:
<input type="checkbox" name="Eile"/></p>
<p>Zum Zur&uuml;cksetzen aller Eingaben:
<input type="reset" value="Zur&uuml;cksetzen"/></p>
<p>Zum Bestellen:
<input type="submit" value="Absenden des Formulars"/></p>
</form>
```

Das Formular wird durch `<form>` und `</form>` begrenzt. Das Attribut `method` des Form-Tags hat den Wert GET. Damit wird beim Absenden des Formulars eine GET-Anfrage an den Web-Server übertragen. Eine Alternative dazu wird in 7.1.3 beschrieben. Mit `action` wird die angeforderte URL festgelegt. Diese kann relativ oder absolut (d.h. mit `http://...` beginnend) sein. Im Beispiel steht Tee. Wenn die HTML-Seite, in der sich das Formular befindet, durch `http://www.hochschule-trier.de:8080/puva/Teebestellung.html` abgerufen worden wäre (tatsächlich wurde es von einem Web-Server auf demselben Rechner geladen, wie in der Adresszeile des Browsers in Bild 7.2 zu sehen ist), dann würde beim Absenden des Formulars die Seite `/puva/Tee` vom Server `www.hochschule-trier.de` am Port 8080 angefordert. Die Elemente zur Dateneingabe werden in diesem Beispiel alle durch Input-Tags beschrieben. Durch das Typ-Attribut kann man die Art des Eingabeeelements angeben:

- Mit `type="text"` wird ein Eingabefeld spezifiziert, in das man einen Text eingeben kann.
- Mit `type="password"` erhält man wie mit `type="text"` ein Eingabefeld, allerdings wird der Text auf dem Bildschirm nicht angezeigt (s. Bild 7.2).
- Mit `type="radio"` erhält man einen Radio-Button. Um mehrere Radio-Buttons so zu einer Gruppe zusammenzufassen, dass höchstens einer der Buttons selektiert sein kann, muss allen Radio-Buttons der Gruppe denselbe Name zugewiesen werden. Als `value` kann man angeben, welcher String an den Web-Server gesendet wird, falls dieser Radio-Button selektiert wurde. Das muss nicht identisch sein mit dem, was auf der Web-Seite zu sehen ist. Im Beispiel oben wurde genau der String geschickt, der auch angezeigt wird. Deshalb kommen die Namen der Teesorten immer jeweils doppelt im HTML-Text vor.
- Mit `type="checkbox"` wird eine Check-Box spezifiziert.
- Mit `type="reset"` erhält man einen Button, mit dem man alle Eingabefelder auf den ursprünglichen Zustand zurücksetzen kann.
- Mit `type="submit"` schließlich erhält man einen Button zum Absenden des Formulars. Das heißt, dass damit eine Anfrage an den entsprechenden Web-Server gesendet wird, wobei die eingegebenen Werte mit übertragen werden.

Eine weiteres Eingabeelement in einem Formular ist eine ComboBox. Diese definiert man über das Tag <select>, wobei zwischen öffnendem und schließendem Tag die Auswahlmöglichkeiten stehen (die Bedeutung von value ist dabei wie oben bei den Radio-Buttons):

```
<select name="Tessorte">
    <option value="Assam">Assam</option>
    <option value="Darjeeling">Darjeeling</option>
    <option value="Ceylon">Ceylon</option>
</select>
```

Die meisten Eingabefelder haben ein Namensattribut. Dieses Attribut ist wichtig, um die eingegebenen Werte den entsprechenden Feldern zuordnen zu können. Wird das in Bild 7.2 gezeigte Formular abgeschickt, dann übermittelt der Browser folgende Anfrage an den Web-Server.

```
GET /puva/Tee?Besteller=Rainer+Oechsle&Kreditkartennummer=123456789&
    Teesorte=Darjeeling&Eile=on HTTP/1.1
Host: ...
...
```

(Nach „123456789&“ folgt kein Newline-Symbol, hier wird nur aus Platzgründen eine neue Zeile benutzt und künstlich eingerückt. Die restlichen Zeilen sind wie zuvor.)

Wie man sieht, wird wie üblich mit GET ein Dokument angefordert. An den Dokumentnamen (im Beispiel /puva/Tee) wird ein Fragezeichen und danach eine Folge von Name-Wert-Paaren (Name=Wert) angehängt, wobei die Name-Wert-Paare durch das Symbol & voneinander getrennt sind. Der Name ist immer der Name eines entsprechenden Eingabeelements des Formulars (z.B. Besteller, Teesorte usw.). Der Wert ist bei einem Eingabefeld der eingegebene Text (wobei Leerzeichen durch Pluszeichen ersetzt werden). Wie oben erläutert wurde, ist es bei einem Radio-Button der Wert, der durch das Value-Attribut des selektierten Radio-Buttons vorgegeben wurde. Wenn kein Wert vorgegeben wurde wie bei der CheckBox im Beispiel, so wird als Wert „on“ übermittelt. Ist die Check-Box nicht selektiert, so wird nicht etwa „Eile=off“ übermittelt, wie man erwarten könnte, sondern das Name-Wert-Paar für den Namen „Eile“ fehlt komplett.

### 7.1.3 POST

Als Alternative zu GET kann für method auch *POST* in einem Formular festgelegt werden:

```
<form method="post" action="Tee">
...
</form>
```

Die Anzeige im Browser wird dadurch nicht geändert. Wird das Formular ausgefüllt wie zuvor und abgesendet, so sendet der Browser dieses Mal allerdings folgende Anfrage an den Web-Server:

```
POST /puva/Tee HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 ... Gecko/20100101 Firefox/55.0
```

```

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/puva/Teebestellung.html
Content-Type: application/x-www-form-urlencoded
Content-Length: 81
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1

Besteller=Rainer+Oechsle&Kreditkartennummer=123456789&Teesorte=Darjeeling&
Eile=on

```

Wie man sieht, ist die erste Zeile anders als zuvor. Statt GET steht nun POST. Als Dokument ist nur /puva/Tee angegeben. Wie die HTTP-Antwort hat jetzt auch die HTTP-Anfrage nicht nur einen Kopf-, sondern auch einen Datenteil, wobei beide durch eine Leerzeile voneinander getrennt sind. Im Kopfteil befindet sich nun auch eine Content-Type- und eine Content-Length-Zeile, welche den Typ und die Länge des Inhalts angeben. Der Inhalt besteht aus einer durch & getrennten Folge von Name-Wert-Paaren (wie zuvor in 7.1.2 beschrieben). Diese Folge von Name-Wert-Paaren im Datenteil wird übrigens nicht durch ein Newline-Symbol abgeschlossen (auch nach „Darjeeling“ steht kein Newline-Symbol).

Die Vor- und Nachteile bei der Verwendung von POST und GET sind:

- Ein mit GET abgeschicktes Formular kann in die Favoritenliste des Browsers aufgenommen werden. Weil die Formulardaten in die URL hineincodiert werden, kann die URL mit den Formulardaten vom Browser abgespeichert werden. Wird diese URL wieder angefordert, so werden damit die ursprünglichen Formulardaten erneut zum Server geschickt. Dies ist bei POST nicht möglich.
- Da der Browser die angeforderte URL in seiner Adresszeile anzeigt, sieht man bei GET Daten auf dem Bildschirm, die man eigentlich nicht sehen wollte. In obigem Beispiel wurde z.B. die Kreditkartennummer in ein Passwort-Feld eingegeben, so dass die Eingabe nicht auf dem Bildschirm sichtbar sein sollte. Wird aber das Formular über GET abgesendet, so taucht diese Nummer in der URL auf und wird deshalb auch in der Adresszeile des Browsers angezeigt. Bei POST sieht man diese Daten in der Adresszeile nicht.
- Die mit GET übertragenen Daten sind in ihrer Länge begrenzt. Bei POST ist dies nicht der Fall.

Aus diesen Ausführungen folgt, dass man in der Regel POST verwenden sollte, was im Folgenden auch immer eingehalten werden wird.

#### 7.1.4 Format von HTTP-Anfragen und -Antworten

Die bisherigen Erläuterungen zu HTTP lassen sich so zusammenfassen:

Eine HTTP-Anfrage hat im Allgemeinen folgende Struktur:

```

<Kommando> <Dokument> HTTP/1.1
<Name>: <Wert>
...
<Name>: <Wert>

<evtl. Daten>

```

Als Kommando kommt GET oder POST in Frage. Bei GET ist der Datenteil leer, während es bei POST in der Regel einen Datenteil gibt.

Eine HTTP-Antwort sieht im Allgemeinen so aus:

```
HTTP/1.1 <Statuscode> <Statustext>
<Name>: <Wert>
...
<Name>: <Wert>

<evt1. Daten>
```

Bei Antworten gibt es in der Regel immer einen Datenteil. Bei Fehlern wird dadurch z. B. der Grund des Fehlers in Form einer HTML-Seite genauer beschrieben.

## ■ 7.2 Einführende Servlet-Beispiele

### 7.2.1 Allgemeine Vorgehensweise

Wie schon in der Einleitung beschrieben, dienen Servlets zur Erzeugung dynamischer Web-Seiten. Häufig beeinflussen Formulardaten, die dem Servlet übermittelt werden, die vom Servlet erzeugte Web-Seite. In der Regel nutzt man eine Entwicklungsumgebung (IDE – Integrated Development Environment) wie Eclipse, IntelliJ oder NetBeans zur Entwicklung von Web-Anwendungen. Im Folgenden ist das Vorgehen für Eclipse angegeben. Konkret benötigt man hier die Variante „Eclipse EE“ (Enterprise Edition). Weiterhin muss man einen Web-Server, der Servlets unterstützt, auf seinem Rechner installieren. Wir gehen hier von Tomcat aus. Folgendes Vorgehen ist zu empfehlen:

- 1. Installation von Eclipse EE und Tomcat:** Nach der Installation von Eclipse EE und Tomcat sollte man in Eclipse EE einen Server konfigurieren und so Eclipse EE das Installationsverzeichnis des Tomcat-Servers mitteilen. Es ist dann möglich, den Tomcat-Server aus Eclipse EE heraus zu starten und wieder herunterzufahren,
- 2. Erzeugung einer Web-Anwendung:** Eine Web-Anwendung wird durch ein „Dynamic Web Project“ erzeugt. In diesem speziellen Eclipse-Projekt kann man sowohl seine statischen HTML-Seiten als auch seinen Java-Code verwalten. Damit die Web-Anwendung über den Web-Server verfügbar ist, muss die Web-Anwendung auf dem Web-Server installiert werden (Deployment). Auch dies lässt sich bequem über die Entwicklungsumgebung Eclipse realisieren.
- 3. Servlet-Klasse programmieren und über eine Annotation konfigurieren:** Dabei ist auf Folgendes zu achten:
  - Die Klasse muss aus *HttpServletRequest* abgeleitet werden.
  - Es sollten die Methoden *doGet* oder *doPost* oder beide überschrieben werden. Die beiden Parameter sind durch die Methoden der Basisklasse vorgegeben, ebenso die möglichen Ausnahmen, die mit *throws* angegeben werden müssen.
  - Bei Bedarf können auch noch die Methoden *init* und *destroy* überschrieben werden. Die Methode *init* wird bei der Aktivierung des Servlets aufgerufen (z. B. beim Hochfahren)

des Web-Servers bzw. der Installation der Web-Anwendung oder beim ersten Ansprechen des Servlets, je nach Konfiguration). Die Methode `destroy` wird aufgerufen, bevor das Servlet vernichtet wird (z. B. beim Herunterfahren des Web-Servers bzw. der Deinstallation [Undeployment] der Web-Anwendung).

Bevor es in Java Annotationen gab, mussten die Servlets in einer XML-Datei (`web.xml`) angegeben werden, wobei dabei u. a. auch der Name vereinbart wurde, unter dem das Servlet über eine URL mit dem HTTP-Protokoll angesprochen wurde. Dabei konnten auch weitere Konfigurationseinstellungen vorgenommen werden. In diesem Buch werden wir jedoch (fast) ausschließlich die modernere Kennzeichnung und Konfiguration von Servlets durch die Annotation `@WebServlet` verwenden. Als Parameter von `@WebServlet` muss zumindest der Name angegeben werden, unter dem das Servlet von außen angesprochen werden kann.

Ein Servlet benötigt keine Main-Methode. An keiner Stelle unseres Programmcodes werden Objekte der Servlet-Klassen erzeugt und deren Methoden aufgerufen; dies erfolgt alles automatisch durch die Laufzeitumgebung des Web-Servers.

Eine Web-Anwendung kann beliebig viele Servlets enthalten.

#### 4. Web-Server starten:

Falls der Web-Server noch nicht läuft, muss er gestartet werden.

Möglich ist das Deployment und Undeployment jedoch auch bei laufendem Web-Server. Änderungen innerhalb der Web-Anwendungen, insbesondere auch Änderungen eines bereits geladenen Servlets werden in der Regel automatisch von einem bereits laufenden Web-Server erkannt und führen spätestens nach einigen Sekunden Wartezeit zu einer automatischen Neuinstallation (Redeployment) der Web-Anwendung. Manchmal kann es jedoch nötig sein, den Web-Server manuell neu zu starten. Dies lässt sich schnell und bequem über die Entwicklungsumgebung erledigen.

#### 5. Servlet mit Hilfe eines Web-Browsers oder der Entwicklungsumgebung ansprechen und ausprobieren

Man kann durch Angabe der richtigen URL das Servlet aus einem Browser heraus ansprechen. Wenn man die URL in die Adresszeile des Browsers eingibt, so sendet der Browser ein GET-Kommando ab. Wenn das Servlet aber nur die Methode `doPost` implementiert, erhält man einen Fehlerstatus als Antwort zurück. Um dem Web-Server ein POST-Kommando mit Hilfe des Browsers zu senden, benötigt man ein entsprechendes Formular, das man zuvor laden muss. Das Formular kann sich in einer statischen HTML-Seite befinden oder es kann von einem Servlet erzeugt werden. Die Entwicklungsumgebung bietet wie für Java-Programme an, das Servlet auf ganz bequeme Art auszuführen. Damit stößt man ein Deployment der Anwendung an. Außerdem öffnet sich in Eclipse ein Browser-ähnliches Fenster, in dem die vom Servlet erzeugte Web-Seite angezeigt wird.

Nun wird es Zeit für ein erstes, konkretes Beispiel.

### 7.2.2 Erstes Servlet-Beispiel

Wir gehen davon aus, dass Eclipse EE und Tomcat bereits installiert sind. Bei der Erzeugung eines „Dynamic Web Project“ muss man den Namen des Projekts angeben. Im Normalfall ist dies auch der Name, der in der URL vorkommt. Man kann aber auch hierfür einen anderen Namen angeben. Danach ist man bereit, das erste Servlet zu programmieren und über eine Annotation zu konfigurieren.

Die Servlet-Klasse wird aus HttpServlet abgeleitet. In diesem ersten Beispiel überschreiben wir nur die Methode doGet. Die Methode doGet hat wie die Methode doPost zwei Parameter: einen, mit dem lesend auf die Bestandteile der HTTP-Anfrage zugegriffen werden kann, und einen, mit dem die vom Web-Server gesendete HTTP-Antwort durch das Servlet bestimmt werden kann. In diesem ersten Beispiel kümmern wir uns um die HTTP-Anfrage nicht. Es wird lediglich eine statische HTML-Seite erzeugt, wozu man eigentlich kein Servlet bräuchte (s. Listing 7.1). Durch die Annotation @WebServlet wird die Klasse als Servlet-Klasse gekennzeichnet. Der Parameter der Annotation gibt den Namen an, unter der das Servlet angesprochen werden kann (s. später). Bitte beachten Sie, dass der Name mit einem Slash („/“) beginnen muss.

### **Listing 7.1**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/HelloWelt")
public class HelloWorldServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title> Hallo Welt</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1> Hallo Welt</h1>");
        out.println("Herzlich willkommen!");
        out.println("</body></html>");
    }
}
```

Mit dem Aufruf der Methode *setContentType* wird die Zeile „Content-Type: text/html“ im Kopf der HTTP-Antwort gesetzt. Auf den Datenteil greift man zu, indem man sich von dem Parameter *response* mit der Methode *getWriter* ein *PrintWriter*-Objekt geben lässt (auch das ist eine Klasse aus dem Package *java.io*, s. Bild 5.6). Die Klasse *PrintWriter* hat mehrere überladene Methoden *print* und *println*, mit denen man den Datenteil in Form von ASCII-Zeichen schreiben kann. In diesem für Servlets nicht untypischen Fall wird eine HTML-Seite generiert.

Die Methoden *doGet* und *doPost* können *protected* sein, da es in der Basisklasse eine Methode namens *service* gibt, die zuerst aufgerufen wird. Diese Methode ruft abhängig vom HTTP-Kommando die in der Basisklasse definierten Protected-Methoden *doGet* oder *doPost* auf. Beide sind so realisiert, dass sie eine Fehlermeldung ausgeben, dass dieses Kommando nicht unterstützt wird. Wenn eine oder beide der Methoden in der abgeleiteten Klasse überschrieben werden, dann wird die entsprechende überschriebene Methode aus der Methode *service* heraus aufgerufen. Werden sie dagegen nicht überschrieben, erfolgt die Ausgabe,

dass dieses Servlet GET bzw. POST nicht unterstützt. Da die Methoden doGet und doPost in der Basisklasse protected sind, genügt es, wenn sie auch in der abgeleiteten Klasse protected sind.

In jeder Web-Anwendung gibt es einen Bereich, in dem man die statischen Inhalte wie HTML-, Bild- und CSS-Dateien ablegen kann. In Eclipse EE heißt dieser Bereich WebContent. Darin kann eine beliebige Unterordnerstruktur erzeugt werden. Angenommen, unsere Web-Anwendung würde puva heißen und im WebContent-Bereich gäbe es einen Ordner namens v1, darin einen Ordner v2 und darin eine Datei namens beispiel.html. Wenn der Tomcat-Server auf dem Rechner mit dem Namen tomcat.hochschule-trier.de gestartet wird und dieser Web-Server auf dem Port 8080 erreichbar ist, dann kann die Datei beispiel.html unter folgender URL vom Tomcat-Server abgerufen werden:

```
http://tomcat.hochschule-trier.de:8080/puva/v1/v2/beispiel.html
```

Nach der Angabe des Rechnernamens (oder alternativ der IP-Adresse), des Ports und des Namens der Anwendung (in diesem Fall puva) wird der normale Pfadname angegeben. Eine Sonderstellung nimmt das Verzeichnis WEB-INF in WebContent ein. Alle darin enthaltenen Dateien und Verzeichnisse sind von außen nicht zugänglich. Der Ordner WEB-INF wird von der Entwicklungsumgebung im WebContent-Bereich direkt mit angelegt. Angenommen, wir würden die Datei beispiel.html in WEB-INF ablegen, so würde die URL nach unserer bisherigen Regel wie folgt lauten:

```
http://tomcat.hochschule-trier.de:8080/puva/WEB-INF/beispiel.html
```

Die Datei ist aber auf diese Art nicht erreichbar.

Ersetzt man in der URL die Pfadstruktur des WebContent-Bereichs (im Beispiel /v1/v2/beispiel.html) durch den Namen, der in der @WebServlet-Annotation angegeben wurde (im Beispiel oben /HalloWelt), dann erhält man die URL, unter der das Servlet von außen erreichbar ist:

```
http://tomcat.hochschule-trier.de:8080/puva/HalloWelt
```

Da das Servlet die Methode doGet überschreibt, kann die URL in die Adresszeile des Browsers eingegeben werden. Alternativ kann man auch eine HTML-Datei mit einem entsprechenden Link (<a href=...>) verwenden.

### 7.2.3 Zugriff auf Formulardaten

Im ersten Beispiel erzeugte das Servlet eine Ausgabe, die unabhängig von der HTTP-Anfrage war. Im nun folgenden Beispiel werden die Daten eines Eingabeformulars aus der HTTP-Anfrage gelesen und zu Demonstrationszwecken lediglich in die erzeugte Webseite geschrieben. In Abschnitt 7.1 wurde gezeigt, dass die eingegebenen Formulardaten bei GET und POST auf unterschiedliche Weise übertragen werden. Erfreulicherweise wird dieser Unterschied durch die Zugriffsmethoden verdeckt; sowohl in doGet als auch in doPost können aus dem ersten Argument mit der Methode *getParameter* die Parameterdaten ausgelesen werden. Als Parameter ist ein String für den Namen des Formulareingabefeldes anzugeben. Es

kommt dann der dazugehörige Wert als String zurück oder null, falls der Name bei den aktuellen Formulardaten nicht dabei ist.

Listing 7.2 zeigt ein Servlet, das die Formulardaten für das Tee-Formular aus 7.1.2 ausliest. Beachten Sie, dass die Annotation @WebServlet den Namen „Tee“ definiert, der genau zu dem Wert des Attributs action im Teeformular (s. Abschnitt 7.1.2) passen muss. Aus Abschnitt 7.1.3 wissen Sie, dass POST gegenüber GET zu bevorzugen ist. Um zu zeigen, dass das Auslesen der Formulardaten bei GET genauso funktioniert wie bei POST, reagiert dieses Servlet sowohl auf POST- als auch auf GET-Anfragen. Die Methode doGet ist einfach durch einen Aufruf von doPost realisiert.

### Listing 7.2

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/Tee")
public class TeeServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Servlet-Bearbeitung der Teebestellung");
        out.println("</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Auftragsbest&uuml;tigung</h1>");
        out.println("<p>Hiermit best&uuml;tigen wir Ihren Auftrag:</p>");
        out.println("<ul>");
        out.println("<li>Teesorte: " +
                   request.getParameter("Teesorte") + "</li>");
        out.println("<li>bestellt von: " +
                   request.getParameter("Besteller") + "</li>");
        out.println("<li>Kartennummer (nur zur Demo angezeigt): " +
                   request.getParameter("Kreditkartennummer") + "</li>");
        if(request.getParameter("Eile") != null)
        {
            out.println("<li><b>Eilbestellung</b></li>");
        }
        out.println("</ul>");
        out.println("<p>Vielen Dank f&uuml;r Ihren Auftrag. " +
                   "Bestellen Sie bald wieder bei uns!</p>");
        out.println("</body>");
        out.println("</html>");
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
    {
```

```

        doPost(request, response);
    }
}
```

## 7.2.4 Zugriff auf die Daten der HTTP-Anfrage und -Antwort

*HttpServletRequest* und *HttpServletResponse* sind Schnittstellen, in denen Methoden definiert werden, mit denen man auf die Daten von HTTP-Anfragen und -Antworten zugreifen kann.

Das vorige Beispiel zeigt das Auslesen von Formulardaten für den Fall, dass man die Namen der Formularfelder beim Schreiben des Servlets kennt, was der Normalfall ist; denn wenn man die Namen der Felder nicht kennt, wird das Servlet-Programm in der Regel auch die eingegebenen Daten nicht interpretieren können. Dennoch kann man sich mit der Methode *getParameterNames* eine Enumeration (Auflistung) aller vorhandenen Parameternamen vom Request-Parameter geben lassen. Da nicht auszuschließen ist, dass mehrere Felder denselben Namen tragen, kann es zu einem Parameternamen im Allgemeinen mehrere Werte geben. Um alle Werte zu bekommen, kann man statt der Methode *getParameter* die Methode *getParameterValues* verwenden. Statt eines Strings wird in diesem Fall ein Feld von Strings zurückgeliefert. Folgendes Programmfragment zeigt, wie man alle Werte zu allen Formularfeldnamen lesen und auf einer HTML-Seite ausgeben kann:

```

out.println("<h3>Alle Parameternamen und ihre Werte</h3>");
Enumeration pars = request.getParameterNames();
out.println("<ul>");
while(pars.hasMoreElements())
{
    String par = (String) pars.nextElement();
    String[] values = request.getParameterValues(par);
    out.print("<li>" + par + ":" );
    out.println("<ul>");
    for(int i = 0; i < values.length; i++)
    {
        out.print("<li>" + values[i] + "</li>");
    }
    out.println("</ul>");
}
out.println("</ul>");
```

In ähnlicher Weise kann man die Kopfzeilen mit der parameterlosen Methode *getHeaderNames* lesen, die ebenfalls eine Enumeration zurückliefert. Damit erhält man eine Auflistung aller Namen vor dem Doppelpunkt im Kopfteil der HTTP-Anfrage. Gibt man einen solchen Namen als Parameter bei *getHeader* an, so liefert die Methode den hinter dem Doppelpunkt stehenden String zurück. Die folgenden Codezeilen lesen alle Kopfzeilen (mit Ausnahme der ersten GET- oder POST-Zeile) der HTTP-Anfrage und geben diese in einer Webseite aus:

```

out.println("<h3>Alle Kopfzeilen (bis auf die erste)</h3>");
Enumeration headers = request.getHeaderNames();
out.println("<ul>");
```

```

        while(headers.hasMoreElements())
    {
        String header = (String) headers.nextElement();
        String hvalue = request.getHeader(header);
        out.println("<li>" + header + ": " + hvalue);
    }
out.println("</ul>");
```

Weitere vom Request-Parameter abfragbare Informationen sind z.B. die Art der HTTP-Anfrage (*getMethod*, Rückgabe „GET“ oder „POST“) oder der Name bzw. die IP-Adresse des Rechners, von dem die HTTP-Anfrage kommt (*getRemoteHost*, *getRemoteAddr*).

Mit dem zweiten Parameter *response* kann der Kopfteil der HTTP-Antwort festgelegt werden. Im ersten Beispiel wurde schon die Methode *setContentType* verwendet. Es existiert darüber hinaus die allgemeinere Methode *setHeader*, mit der eine Kopfzeile im Kopfteil erzeugt bzw. ersetzt werden kann. Die Methode hat zwei String-Parameter: der erste Parameter ist der Teil vor dem Doppelpunkt und der zweite Parameter der Teil danach. Ferner kann mit der Methode *setStatus* der Rückgabecode der HTTP-Antwort bestimmt werden (erste Zeile der HTTP-Antwort, s. Abschnitt 7.1). Das Argument dieser Methode ist eine Zahl des Typs int. In *HttpServletResponse* sind eine ganze Reihe von Konstanten für Statuscodes definiert (z.B. SC\_OK für 200 und SC\_NOT\_FOUND für 404).

## ■ 7.3 Parallelität bei Servlets

### 7.3.1 Demonstration der Parallelität von Servlets

Wie in Abschnitt 6.3 für RMI können wir auch bei Servlets die parallelen Abläufe durch länger laufende Methodenaufrufe sichtbar machen. Wie bei RMI verwenden wir auch dieses Mal zu diesem Zweck die Sleep-Methode. Wie lange der auszuführende Thread schlafen soll, könnte eine Konstante sein. Im RMI-Beispiel konnte man die Schlafenszeit jedoch durch einen Parameter einstellen. Wenn dieses Verhalten mit Servlets nachgeahmt werden soll, können wir dies durch ein Formular mit einem Eingabefeld realisieren. Das Eingabefeld hat den Namen „Sekunden“. Ein Servlet, welches dann zur Bearbeitung so viele Sekunden braucht, wie man in das Eingabefeld eingetragen hat, zeigt Listing 7.3. Formulardaten werden immer als Strings übermittelt. Deshalb muss aus der Eingabe zuerst ein entsprechender Zahlenwert berechnet werden. Wie im RMI-Beispiel könnte unser Programm auch dieses Mal zusätzlich noch Ausgaben mit *System.out.println* produzieren. Diese Ausgaben würden im Fenster des Tomcat-Servers erscheinen. Darauf wurde aber verzichtet.

**Listing 7.3**

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/Schlafen")
```

```

public class SleepingServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head><title>Schlafen</title>");
        out.println("</head><body><h1>Schlafen</h1>");
        String secsString = request.getParameter("Sekunden");
        if(secsString != null)
        {
            try
            {
                int secs = Integer.parseInt(secsString);
                Thread.sleep(secs * 1000);
                out.println("<p>Es wurde " + secs
                           + " Sekunden geschlafen.</p>");
            }
            catch(Exception e)
            {
                out.println("<p>Es gab Probleme beim Schlafen.</p>");
            }
        }
        else
        {
            out.println("<h2>GET-Formular");
            out.println("<form method=\"get\">");
            out.println("<input name=\"Sekunden\">");
            out.println("<input type=\"submit\" value=\"Los!\">");
            out.println("</form>");
            out.println("<h2>POST-Formular");
            out.println("<form method=\"post\">");
            out.println("<input name=\"Sekunden\">");
            out.println("<input type=\"submit\" value=\"Los!\">");
            out.println("</form>");
        }
        out.println("</body></html>");
    }
    protected synchronized void doPost(HttpServletRequest request,
                                      HttpServletResponse response)
        throws IOException, ServletException
    {
        doGet(request, response);
    }
}

```

Wenn das Servlet ohne den Parameternamen „Sekunden“ aufgerufen wird (dies ist in der Regel beim ersten Mal der Fall, wenn man nur die URL des Servlets in die Adresszeile eingibt und Return drückt), erzeugt es zwei Formulare, die gleich aussehen. Eines wird mit GET, das andere wird mit POST verschickt. In beiden Fällen wird dasselbe Servlet, das die Formulare erzeugt hat, wieder aktiv. Bitte beachten Sie, dass aus diesem Grund in den vom Servlet erzeugten Formularen die Action-Angabe in den Formular-Tags fehlt. Als Action-URL wird in diesem Fall die aktuelle URL (d.h. die URL, mit der die Formulare geladen wurden) verwendet. Dies ist hier möglich, da dasselbe Servlet, welches die Formulare erzeugt hat, auch auf das Absenden der ausgefüllten Formulare reagiert.

Mit dem einen Formular wird doGet, mit dem anderen doPost aufgerufen. Die beiden Methoden doGet und doPost machen dasselbe. Der Aufruf von doPost ist im Gegensatz zu doGet allerdings synchronisiert. Es zeigt sich beim Ausprobieren, dass zwei GET-Anfragen gleichzeitig bearbeitet werden. Dies beweist, dass wie bei RMI hier implizite Parallelität im Spiel ist und jede HTTP-Anfrage in einem eigenen Thread ausgeführt wird. Beim Ausprobieren stellt man aber auch fest, dass mehrere POST-Anfragen nicht gleichzeitig abgewickelt werden können. Das beweist, dass alle Anfragen mit demselben Servlet-Objekt arbeiten (pro Servlet-Klasse wird also in der Regel nur ein einziges Objekt erzeugt). Zusammenfassend kann gefolgert werden, dass man mit Parallelität umgehen muss, auch wenn man selbst explizit keine Threads erzeugt. Dies wird im nächsten Beispiel berücksichtigt werden.

Ein solches Servlet, das ziemlich lange braucht, um seine Ausgabe zu erzeugen, dient nur zu Demonstrationszwecken. Wie man Servlets, die länger dauernde Aufträge ausführen, gestalten sollte, wird in Abschnitt 7.4.4 erläutert. Im Folgenden befassen wir uns zunächst mit der Parallelität bei Servlets.

### 7.3.2 Paralleler Zugriff auf Daten

Im Folgenden wird das Beispiel zum Hochzählen und Zurücksetzen eines Zählers, das schon mit Sockets und RMI realisiert wurde, auf ein Servlet übertragen. Da – wie gerade gesehen – jede HTTP-Anfrage in einem eigenen Thread ausgeführt wird, muss auf korrekte Synchronisation geachtet werden. Im einfachsten Fall könnte man den Zähler durch ein Attribut der Servlet-Klasse repräsentieren. Alle lesenden und schreibenden Zugriffe auf dieses Attribut müssen dann synchronisiert erfolgen. Dies kann durch einen Synchronized-Block oder durch das Kennzeichnen der Methode doGet bzw. doPost mit synchronized erfolgen. Die folgenden Zeilen skizzieren die erste der beiden Möglichkeiten:

```
@WebServlet("/Zaehler")
public class CounterServlet extends HttpServlet
{
    private int counter;

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        ...
        int newValue;
        synchronized(this)
        {
            if(...) //Kommando increment
            {
                counter++;
            }
            else if(...) //Kommando reset
            {
                counter = 0;
            }
            newValue = counter;
        }
        ... //Erzeugung der Webseite mit Anzeige von newValue
    }
}
```

Alternativ kann der Zugriff auf das Zählerattribut auch durch private Synchronized-Methoden erfolgen:

```
@WebServlet("/Zaehler")
public class CounterServlet extends HttpServlet
{
    private int counter;

    private synchronized int increment()
    {
        counter++;
        return counter;
    }

    private synchronized int reset()
    {
        counter = 0;
        return counter;
    }

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws IOException, ServletException
    {
        ...
        int newValue;
        if(...) //Kommando increment
        {
            newValue = increment();
        }
        else if(...) //Kommando reset
        {
            newValue = reset();
        }
        ... //Erzeugung der Webseite mit Anzeige von newValue
    }
}
```

Schließlich kann das Zählerattribut zusammen mit den Zugriffsmethoden increment und reset in eine eigene Klasse Counter (s. Listing 7.4) ausgelagert werden (die Methode set wird später benötigt). In der Servlet-Klasse wird dann ein Attribut auf ein Objekt dieser Klasse gehalten. Das Objekt kann bei der ersten Benutzung angelegt werden. Dazu muss in den Servlet-Methoden jedes Mal geprüft werden, ob die Referenz auf das Counter-Objekt null ist. Wenn dies der Fall ist, wird ein Objekt erzeugt (Achtung: Synchronisation notwendig). Alternativ kann auch die Methode init überschrieben werden, die für solche Zwecke gedacht ist. Sie wird aufgerufen, bevor der erste Aufruf von doGet oder doPost erfolgt. In Listing 7.5 wird diese Variante umgesetzt.

#### **Listing 7.4**

```
public class Counter
{
    private int counter;

    public synchronized int increment()
    {
```

```
        counter++;
    return counter;
}

public synchronized int reset()
{
    counter = 0;
    return counter;
}

public synchronized void set(int counter)
{
    this.counter = counter;
}
```

### Listing 7.5

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/Zahler")
public class CounterServlet extends HttpServlet
{
    private Counter counter;

    public void init()
    {
        counter = new Counter();
    }

    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
            throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Z&auml;hler</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Z&auml;hler</h1>");

        String message = "";
        String operation1 = request.getParameter("increment");
        if(operation1 != null)
        {
            int value = counter.increment();
            message = "Der Z&auml;hler wurde auf " + value
                      + " erh&ouml;ht.<p>";
        }
        String operation2 = request.getParameter("reset");
        if(operation2 != null)
        {
```

```

        int value = counter.reset();
        message = "Der Zähler wurde auf " + value
                  + " zurückgesetzt.<p>";
    }
    out.println(message);

    out.println("<form method=\"post\">");
    out.println("<input type=\"submit\" name=\"increment\" "
               + "value=\"Erhöhen\">");
    out.println("<input type=\"submit\" name=\"reset\" "
               + "value=\"Zurücksetzen\">");
    out.println("</form>");

    out.println("</body>");
    out.println("</html>");
}
protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
throws IOException, ServletException
{
    doGet(request, response);
}
}

```

Obwohl wir POST für Formulare bevorzugen, unterstützt das Servlet sowohl das POST- als auch das GET-Kommando. Dies liegt zum einen daran, dass wir weder viele noch geheime Formulardaten übertragen. Zum anderen erzeugt das Servlet das Eingabeformular selbst. Um den ersten Aufruf des Servlets in einfacher Weise zu ermöglichen (z.B. durch Eingabe der Servlet-URL in die Adresszeile des Browsers), ist die Unterstützung des GET-Kommandos notwendig.

Das Formular zum Erhöhen und Zurücksetzen des Zählers ist sehr einfach. Es besteht aus lediglich zwei Submit-Buttons. Um zu unterscheiden, welcher der beiden Buttons das Absenden des Formulars ausgelöst hat, wird den Submit-Buttons ebenfalls ein Name gegeben. Der Name des Buttons wird dann zusammen mit seinem Wert als Name-Wert-Paar wie die anderen Name-Wert-Paare an den Web-Server übermittelt. Man könnte den beiden Buttons denselben Namen geben. In diesem Fall müsste die Fallunterscheidung durch den Wert erfolgen. In dem Programm aus Listing 7.5 wurde ein anderer Weg beschritten: Den beiden Buttons werden unterschiedliche Namen zugewiesen. Welcher der beiden Buttons gedrückt wurde, erkennt man dann daran, welcher Name definiert ist (d.h. für welchen Namen die Methode getParameter nicht null zurückgibt). Wie in Listing 7.3 fehlt auch hier in Listing 7.5 die Action-Angabe im Formular-Tag.

Im Normalfall wird ein Servlet-Objekt erzeugt und dessen Methode init aufgerufen, wenn das Servlet zum ersten Mal angesprochen wird. Für diese Anwendung reicht das aus. Man kann in der Annotation @WebServlet (oder alternativ in der Konfigurationsdatei web.xml) aber auch festlegen, dass bereits beim Installieren der Anwendung (also auch z.B. beim Neuladen der Anwendung) das Servlet-Objekt erzeugt und seine Initialisierungsmethode aufgerufen wird. Dies wird im folgenden Beispiel benutzt werden.

### 7.3.3 Anwendungsglobale Daten

In manchen größeren Web-Anwendungen ist es notwendig, von unterschiedlichen Servlets aus auf dieselben Daten zuzugreifen. Wir wollen dies anhand unseres Zählerbeispiels illustrieren. Dieses Beispiel ist zugegeben sehr einfach. Man kann jedoch die prinzipielle Vorgehensweise gut an diesem Beispiel erläutern.

Dazu nehmen wir an, dass wir für das Erhöhen und das Zurücksetzen des Zählers unterschiedliche Formulare benutzen wollen (die Notwendigkeit dafür ist wenig einleuchtend, aber stellen Sie sich bitte umfangreiche Formulare vor, die in unterschiedliche kleinere Formulare aufgeteilt werden sollen). Nun ist es selbstverständlich möglich, auch in unterschiedlichen Formularen dieselbe URL als Action anzugeben und damit dasselbe Servlet für die Auswertung unterschiedlicher Formulare zu benutzen. Wenn wir nun aber annehmen, dass wir aus strukturellen Gründen mit unterschiedlichen Servlets auf das Ausfüllen der beiden Formulare reagieren wollen, d. h. einem IncrementServlet und einem ResetServlet, dann ergibt sich das Problem, aus zwei unterschiedlichen Servlets auf dasselbe Counter-Objekt zugreifen zu müssen. Eine mögliche Lösung hierfür wäre die Benutzung von Static-Attributen. Dies ist zwar möglich. Jedoch sieht die empfohlene Lösung für das geschilderte Problem anders aus: Empfohlen wird die Nutzung des so genannten *ServletContexts*. Ein Objekt dieses Typs wird von der von HttpServlet geerbten Methode `getServletContext` zurückgeliefert. Jede im Web-Server aktuell installierte Anwendung hat ihr eigenes ServletContext-Objekt (ein besserer Name wäre deshalb *ApplicationContext*). Ein ServletContext-Objekt beinhaltet eine Attributliste: Man kann mit der Methode `setAttribute` unter einem angegebenen Namen eine Referenz auf ein Objekt eines beliebigen Typs im ServletContext ablegen. Durch Angabe desselben Strings erhält man mit der Methode `getAttribute` eine Referenz auf das abgelegte Objekt zurück. Mit der Methode `removeAttribute` lässt sich der Name und die dazugehörige Referenz aus dem ServletContext wieder entfernen.

Wenn mehrere Servlets gemeinsame Objekte mit Hilfe des ServletContexts nutzen wollen, müssen zuvor folgende Fragen beantwortet werden:

- Unter welchem Namen werden die Objekte im ServletContext abgelegt?
- Welches der Servlets erzeugt die Objekte und legt sie im ServletContext ab bzw. entfernt sie wieder daraus?

In unserer Beispielanwendung legen wir ein Objekt der Counter-Klasse unter dem Namen „Counter“ ab. Das ResetServlet erzeugt das Objekt in seiner Init-Methode und legt es im ServletContext ab. Da man aber nicht weiß, ob zuerst das ResetServlet oder zuerst das IncrementServlet aktiv wird, soll die Initialisierung nicht erst bei der erstmaligen Aktivierung des ResetServlets, sondern bereits beim Installieren bzw. Neuladen der Anwendung erfolgen. Dies erreicht man, wenn man bei der Annotation @WebServlet den Parameter `loadOnStartup` angibt. Der Wert von `loadOnStartup` ist dann relevant, wenn man mehrere Servlets mit `loadOnStartup` in seiner Anwendung hat. Dann kann über diesen Wert die Reihenfolge der Instanziierung und Initialisierung der Servlets festgelegt werden. Da wir nur ein solches Servlet haben, ist der Wert beliebig.

In Listing 7.6 und Listing 7.7 sind die beiden Servlet-Klassen ResetServlet und IncrementServlet zu finden. Der Umgang mit dem gemeinsam genutzten Objekt der Klasse Counter ist durch Fettdruck hervorgehoben. Achten Sie besonders darauf, dass die Synchronisation in

der schon in Listing 7.4 gezeigten Klasse Counter erfolgt, dass jedoch in den Servlets keine weitere Synchronisation notwendig ist.

#### **Listing 7.6**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@.WebServlet(value="/Zuruecksetzen", loadOnStartup=1)
public class ResetServlet extends HttpServlet
{
    public void init()
    {
        Counter counter = new Counter();
        ServletContext ctx = getServletContext();
        ctx.setAttribute("Counter", counter);
    }

    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response)
            throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>");
        out.println("<title>Z&uuml;hler</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Zur&uuml;cksetzen des Z&uuml;hlers</h1>");

        ServletContext ctx = getServletContext();
        Counter counter = (Counter)ctx.getAttribute("Counter");
        int value = counter.reset();
        out.println("Der Z&uuml;hler wurde auf " + value
                  + " zur&uuml;ckgesetzt.<p>");

        out.println("</body>");
        out.println("</html>");
    }
}
```

#### **Listing 7.7**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/Erhoehen")
public class IncrementServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request,
                         HttpServletResponse response)
            throws IOException, ServletException
    {
```

```

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html><head>");
        out.println("<title>Z&uuml;hler</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Erh&ouml;ung des Z&uuml;hlers</h1>");

        ServletContext ctx = getServletContext();
        Counter counter = (Counter)ctx.getAttribute("Counter");
        int value = counter.increment();
        out.println("Der Z&uuml;hler wurde auf " + value
                  + " erh&ouml;ht.<p>");

        out.println("</body>");
        out.println("</html>");
    }
}

```

Der in diesem Beispiel verwendete ServletContext gehört zu einer Web-Anwendung. Er wird erzeugt, wenn die entsprechende Web-Anwendung gestartet wird, und er wird gelöscht, wenn die entsprechende Web-Anwendung angehalten wird. Sollte man beim Erzeugen oder beim Löschen des ServletContexts noch anwendungsspezifische Aktionen durchführen wollen, so kann man einen entsprechenden Listener programmieren, der die Schnittstelle ServletContextListener mit den Methoden contextInitialized und contextDestroyed implementiert. Die Listener-Klasse muss mit `@WebListener` annotiert (oder alternativ in der Konfigurationsdatei web.xml bekannt gemacht) werden.

Zur Illustration einer möglichen Verwendung eines ServletContextListener kommen wir wieder auf das Beispiel von eben zurück. Man kann die Methode init des ResetServlets sowie den Parameter loadOnStartup in der Annotation @WebServlet entfernen und stattdessen das Counter-Objekt durch einen ServletContextListener erzeugen und in den ServletContext eintragen. Listing 7.8 zeigt die Klasse CounterInitializer. In der Methode contextInitialized befindet sich jetzt die Programmlogik, die zuvor in der Init-Methode des ResetServlets angesiedelt war. Der einzige Unterschied ist der, dass die Referenz auf den ServletContext jetzt über die Methode getServletContext der Klasse ServletContextEvent beschafft wird. Ein Objekt vom Typ ServletContextEvent wird der Methode contextInitialized beim Aufruf übergeben.

### **Listing 7.8**

```

import javax.servlet.*;
import javax.servlet.annotation.*;

@WebListener
public class CounterInitializer implements ServletContextListener
{
    public void contextInitialized(ServletContextEvent event)
    {
        Counter counter = new Counter();
        ServletContext ctx = event.getServletContext();
        ctx.setAttribute("Counter", counter);
    }
}

```

```
public void contextDestroyed(ServletContextEvent event)
{
}
}
```

Es können übrigens beliebig viele ServletContextListener implementiert werden. Beim Erzeugen des ServletContexts werden dann alle ContextInitialized- und beim Löschen alle ContextDestroyed-Methoden aufgerufen.

Weitere Listener können die Schnittstelle ServletContextAttributeListener implementieren. Hier kann man sich durch Aufruf u. a. der Methoden attributeAdded, attributeReplaced und attributeRemoved benachrichtigen lassen, wenn ein Attribut in den Kontext abgelegt, geändert oder daraus gelöscht wird.

## ■ 7.4 Sessions und Cookies

In den webbasierten Zählerbeispielen des vorigen Abschnitts gibt es ein einziges Zählerobjekt, auf das alle Benutzer gemeinsam zugreifen. Es gibt aber Anwendungen, bei denen jeder Benutzer ein eigenes Objekt benötigt. Das Standardbeispiel hierzu ist ein Warenkorb für eine webbasierte Anwendung zum elektronischen Einkaufen. Eine Benutzerin bewegt sich durch die unterschiedlichen Seiten mit den Beschreibungen der Waren. Von Zeit zu Zeit legt sie etwas in den Warenkorb. Da der webbasierte Laden von mehreren Personen gleichzeitig besucht werden kann, ist es natürlich wichtig, dass jede Person ihren eigenen Warenkorb besitzt.

In einem Servlet kann man über den Request-Parameter mit der Methode getRemoteAddr abfragen, von welchem Rechner die Anfrage kommt. Diese Angabe sollte aber in keinem Fall dazu verwendet werden, Benutzer zu unterscheiden. Und zwar aus folgenden Gründen:

- Ein Rechner kann gleichzeitig oder rasch hintereinander von unterschiedlichen Personen benutzt werden. Bei Großrechnern ist die gleichzeitige Nutzung durch mehrere Personen der Normalfall. In einer Hochschule werden die Rechner der PC-Pools von mehreren Personen nacheinander benutzt. Wenn sich eine Studentin in einem Web-Shop Waren in ihren Warenkorb legt und sich dann ohne Abmeldung am Web-Shop am System abmeldet, dann würde der nächste Student, der sich am selben Rechner anmeldet und den Web-Shop betritt, aufgrund der IP-Adresse nicht von der vorigen Benutzerin unterscheiden werden können. Dies könnte unter Umständen dazu führen, dass der Student Waren bestellt, die er nicht haben möchte, oder dass der Student nachsehen kann, welche Waren sich die Studentin in den Warenkorb gelegt hatte.
- Bei WWW werden häufig *Proxies* verwendet, die häufig benutzte Webseiten in einem Cache-Speicher halten. Falls der Proxy die Seite nicht hat, beschafft sich der Proxy die Seite vom Web-Server. In diesem Fall kommen dann die Anfragen für alle Benutzer des Proxy-Rechners immer von diesem Proxy; die Unterscheidung von Benutzern ist nicht möglich.
- Ein ähnlicher Effekt passiert bei der *Adressenabbildung* (*NAT: Network Address Translation*). NAT wird verwendet, wenn ein lokales Netz mit privaten IP-Adressen an das Inter-

net angeschlossen wird, wobei für den Anschluss nur eine einzige öffentliche IP-Adresse zur Verfügung steht, die sich die Rechner des lokalen Netzes teilen. Auch in diesem Fall funktioniert eine Unterscheidung von Benutzern anhand der IP-Adresse nicht. Dieser Fall ist im Prinzip so wie der vorige, nur dass hier die Adressumsetzung auf der Schicht 3 statt wie zuvor auf der Schicht 5 vorgenommen wird.

Die Nutzung der IP-Adresse zur Unterscheidung funktioniert also aus mehreren Gründen nicht. In diesem Abschnitt wird erläutert, wie man das Problem lösen kann. Dazu verwenden wir wieder das einfache Zählerbeispiel. Die Leserinnen und Leser sind sicherlich in der Lage, dieses Beispiel auf kompliziertere Datenstrukturen zu übertragen.

### 7.4.1 Sessions

Mit Hilfe von *Sessions* wird eine Entwicklerin in die Lage versetzt, in einfacher Weise unterschiedliche Benutzer zu unterscheiden und damit z. B. einen Warenkorb oder einen Zähler pro Anwender zu realisieren. Eine Session wird durch die Methode *getSession* des Request-Parameters von doGet bzw. doPost erzeugt:

```
public interface HttpServletRequest extends ServletRequest
{
    public HttpSession getSession(boolean create);
    public HttpSession getSession();
    ...
}
```

Mit getSession wird eine laufende Session des Benutzers zurückgegeben. Falls keine Session existiert (weil der Benutzer z. B. zum ersten Mal dieses Servlet aufruft), wird eine neue Session erzeugt, falls die Methode getSession mit dem Argument true oder ohne Argumente aufgerufen wird. Falls getSession mit dem Parameter false aufgerufen wird, dann wird für den Fall, dass noch keine Session existiert, null zurückgeliefert.

*HttpSession* ist wie HttpServletRequest und HttpServletResponse eine Schnittstelle. Wie zuvor beim ServletContext gibt es Methoden zum Speichern von Objekten beliebigen Typs im Session-Objekt, zum Abrufen und zum Löschen dieser Objekte. Die Objekte werden mit einem frei wählbaren Bezeichner assoziiert:

```
public interface HttpSession
{
    public void setAttribute(String name, Object value);
    public Object getAttribute(String name);
    public void removeAttribute(String name);
    ...
}
```

Damit lässt sich die Aufgabe, für jeden Anwender einen eigenen Zähler zu realisieren, einfach umsetzen. In der Servlet-Methode lässt man sich mit getSession ein Session-Objekt geben. Beim ersten Mal wird die Session neu erzeugt und besitzt folglich noch kein Zähler-Attribut. Wenn dies der Fall ist (getAttribute liefert null zurück), wird ein neues Zähler-objekt erzeugt und unter dem Namen „Counter“ in der Session abgelegt. Bei den folgenden Aufrufen der Servlet-Methode durch denselben Benutzer liefert getSession die zuvor

erzeugte Session zurück, über die man mit getAttribute auf das zuvor dort abgelegte Zählerobjekt zugreifen kann. In Listing 7.9 ist die soeben skizzierte Lösung als Java-Programm formuliert.

### Listing 7.9

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/ZaehlerSitzung")
public class CounterSessionServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Z&auml;hler</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Z&auml;hler</h1>");

        HttpSession session = request.getSession(true);
        Counter counter;
        synchronized(session)
        {
            counter = (Counter)session.getAttribute("Counter");
            if(counter == null)
            {
                counter = new Counter();
                session.setAttribute("Counter", counter);
            }
        }
        String message = "";
        String operation1 = request.getParameter("increment");
        if(operation1 != null)
        {
            int value = counter.increment();
            message = "Der Z&auml;hler wurde auf " + value
                      + " erh&ouml;ht.

>";
        }
        String operation2 = request.getParameter("reset");
        if(operation2 != null)
        {
            int value = counter.reset();
            message = "Der Z&auml;hler wurde auf " + value
                      + " zur&uuml;ckgesetzt.

>";
        }
        out.println(message);

        out.println("<form method=\"post\">");
        out.println("<input type=\"submit\" name=\"increment\" "
                  + "value=\"Erh&ouml;hen\">");
    }
}


```

```

        out.println("<input type=\"submit\" name=\"reset\" "
            + "value=\"Zurücksetzen\">");
        out.println("</form>");

        out.println("</body>");
        out.println("</html>");
    }

    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        doGet(request, response);
    }
}

```

Wenn man davon ausgeht, dass eine Web-Anwendung auf einem Web-Server relativ lange ununterbrochen läuft, dann erkennt man, dass es nicht möglich ist, dass ständig neue Session- und neue Zählerobjekte erzeugt werden können, ohne dass diese wieder gelöscht werden. Die Anwendung würde irgendwann den verfügbaren Speicher aufgebraucht haben und das Erzeugen eines neuen Objekts würde mit einer Ausnahme enden. Dieses Problem besteht für alle Anwendungen mit Sessions, nicht nur für unsere Zähleranwendung.

Zwei Lösungen existieren für dieses Problem: Zum einen können die Benutzer anzeigen, dass sie in Zukunft diese Session nicht mehr benutzen werden. Auf vielen Webseiten gibt es zu diesem Zweck einen Link auf eine Seite zum Abmelden („Logout“). Im entsprechenden Servlet wird die Session explizit gelöscht. Zu diesem Zweck ruft man die Methode *invalidate* auf dem Session-Objekt auf. Beim nächsten Aufruf von *getSession* wird die Session für diesen Benutzer dann nicht mehr gefunden.

Allerdings kann man sich nicht darauf verlassen, dass der Benutzer sich explizit abmeldet. Der Benutzer könnte z. B. auch einfach seinen Browser beenden und nie wieder auf die Web-Anwendung zugreifen. Der Web-Server kann dies nicht erkennen, da HTTP ein zustandsloses Protokoll ist. Das bedeutet, dass ein GET- oder POST-Kommando sowie die dazugehörige Antwort über eine TCP-Verbindung gesendet wird. Die Verbindung wird dann nach einiger Zeit abgebaut. Ob derselbe Benutzer in Zukunft nochmals ein Kommando an denselben Web-Server sendet oder nicht, weiß man nicht. Deshalb muss die Beendigung einer Session durch eine Zeitsteuerung ergänzt werden: Wenn eine Session eine bestimmte Zeit lang nicht mehr benutzt wurde, wird sie automatisch beendet. Durch das Löschen des Session-Objekts verlieren auch alle in der Session gespeicherten Attribute ihre Referenz und werden im Zuge der Java-Abfallsammlung (Garbage Collection) ebenfalls gelöscht. Mit der Methode *setMaxInactiveInterval* kann man für jede Session einstellen, nach welcher Zeit das Session-Objekt automatisch gelöscht wird, falls sie so lange nicht mehr benutzt wurde. Mit weiteren Methoden kann man sich die Zeiten geben lassen, zu denen die Session erzeugt bzw. zuletzt benutzt wurde:

```

public interface HttpSession
{
    public void invalidate();
    public void setMaxInactiveInterval(int interval);
    public int getMaxInactiveInterval();
    public long getCreationTime();
}

```

```
    public long getLastAccessedTime();
    ...
}
```

Wegen des automatischen Löschens von Sessions muss sich also unser Servlet aus Listing 7.9 nicht um das Löschen der Session- und Zählerobjekte kümmern. Da jeder Benutzer sein eigenes Zählerobjekt besitzt, könnte man annehmen, dass die Methoden increment und reset in der Klasse Counter (s. Listing 7.4) nun nicht mehr synchronized sein müssen. Diese Annahme ist jedoch falsch, was jede Leserin und jeder Leser durch ein kleines Experiment ausprobieren kann. Wenn man z. B. den Internet Explorer mehrfach startet und die Anwendung aus Listing 7.9 benutzt, stellt man fest, dass es für jeden Internet Explorer eine eigene Session mit einem eigenen Zähler gibt. Wenn man jedoch in einem Browser mehrere Registerkarten öffnet und in diesen auf die Anwendung zugreift, dann sieht man, dass jetzt auf demselben Zähler gearbeitet wird. Würden die Methoden increment oder reset etwas länger dauern (was man mit Hilfe von Thread.sleep erzwingen kann), dann kann man z. B. in einer Registerkarte den Aufruf von increment anstoßen, auf eine zweite Registerkarte wechseln und ebenfalls den Aufruf von increment anstoßen. Wenn die Methode increment nicht synchronized wäre, dann würde increment zwei Mal gleichzeitig von zwei Threads ausgeführt, und das für dasselbe Objekt. Wie in Kapitel 2 ausführlich diskutiert, sollen solche Abläufe verhindert werden. Auch muss das Anlegen des Counter-Objekts und sein Abspeichern in der Session synchronisiert werden. Auf synchronized kann also auch in diesem Fall nicht verzichtet werden.

Wie für den ServletContext kann man auch für Sessions einen oder mehrere Listener anmelden, deren Methoden beim Erzeugen bzw. Löschen von Sessions aufgerufen werden. Die zu implementierende Schnittstelle heißt in diesem Fall HttpSessionListener und hat die Methoden sessionCreated und sessionDestroyed. Beide Methoden bekommen als Argument ein Objekt des Typs HttpSessionEvent übergeben. Von diesem Event-Objekt kann man mit Hilfe der Methode getSession auf die soeben erzeugte bzw. die demnächst gelöschte Session zugreifen. Ähnlich wie zuvor beim globalen Zähler kann man das Erzeugen und Abspeichern des Counter-Objekts aus dem Programmcode des CounterSessionServlets (Listing 7.9) löschen und in die Methode sessionCreated eines HttpSessionListeners verlagern. Listing 7.10 zeigt die entsprechende Listener-Klasse, die wieder mit @WebListener annotiert ist.

### **Listing 7.10**

```
import javax.servlet.*;
import javax.servlet.annotation.*;

@WebListener
public class CounterSessionInitializer
    implements HttpSessionListener
{
    public void sessionCreated(HttpSessionEvent event)
    {
        Counter counter = new Counter();
        HttpSession session = event.getSession();
        session.setAttribute("Counter", counter);
    }
}
```

```
public void sessionDestroyed(HttpSessionEvent event)
{
}
}
```

Bei dieser Lösung wird also immer dann, wenn eine neue Session erzeugt wird, ein Counter-Objekt generiert und in der Session abgelegt. Diese Lösung ist natürlich dann nicht besonders gut, wenn in der Anwendung auch für andere Zwecke Sessions erzeugt werden, die keinen Zähler benötigen. Dann ist es besser, die Session wie ursprünglich im Servlet zu initialisieren.

Zur Demonstration, dass auch mehrere Listener nebeneinander existieren können, wird in Listing 7.11 noch ein zweiter SessionListener angegeben, der die Anzahl der vorhandenen Sessions zählt und auf die Standardausgabe ausgibt.

#### **Listing 7.11**

```
import javax.servlet.*;
import javax.servlet.annotation.*;

@WebListener
public class SessionListenerExample implements HttpSessionListener
{
    private int numberofSessions;

    public void sessionCreated(HttpSessionEvent event)
    {
        numberofSessions++;
        System.out.println("Anzahl der Sessions: "
                           + numberofSessions);
    }

    public void sessionDestroyed(HttpSessionEvent event)
    {
        numberofSessions--;
        System.out.println("Anzahl der Sessions: "
                           + numberofSessions);
    }
}
```

Wie für den ServletContext gibt es auch für Sessions Listener, die benachrichtigt werden, wenn Attribute zu einer Session hinzugefügt, geändert oder aus einer Session gelöscht werden. Diese Listener müssen die Schnittstelle HttpSessionAttributeListener mit den Methoden attributeAdded, attributeReplaced und attributeRemoved implementieren.

#### **7.4.2 Realisierung von Sessions mit Cookies**

Sessions werden im Regelfall mit Hilfe von *Cookies* realisiert. Um zu verstehen, wie dies funktioniert, kann man einfach den Datenverkehr bei der Benutzung unserer Anwendung aus Listing 7.9 beobachten. Der Wechsel der Datenübertragungsrichtung wird durch gestrichelte Linien angezeigt:

```
GET /puva/ZaehlerSitzung HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 ... Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1

-----
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Set-Cookie: JSESSIONID=881280D5E9CB559993E59FD5C8177E28; Path=/puva/
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 256
Date: Thu, 28 Sep 2017 14:51:36 GMT

<html>
<head>
<title>Z&auml;hler</title>
</head>
<body>
<h1>Z&auml;hler</h1>

<form method="post">
<input type="submit" name="increment" value="Erh&ouml;hen">
<input type="submit" name="reset" value="Zur&ampuumlcksetzen">
</form>
</body>
</html>

-----
POST /puva/ZaehlerSitzung HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 ... Gecko/20100101 Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/puva/ZaehlerSitzung
Content-Type: application/x-www-form-urlencoded
Content-Length: 19
Cookie: JSESSIONID=881280D5E9CB559993E59FD5C8177E28
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1

increment=Erh%F6hen

-----
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html;charset=ISO-8859-1
Content-Length: 299
Date: Thu, 28 Sep 2017 14:51:51 GMT

<html>
<head>
<title>Z&auml;hler</title>
```

```
</head>
<body>
<h1>Zähler</h1>
Der Zähler wurde auf 1 erhöht.<p>
<form method="post">
<input type="submit" name="increment" value="Erhöhen">
<input type="submit" name="reset" value="Zurücksetzen">
</form>
</body>
</html>
```

Zu Beginn wird das Servlet mit GET angesprochen. Der Benutzer hat z. B. die Servlet-URL in die Adresszeile des Browsers eingegeben und die Return-Taste gedrückt. In der Antwort sehen wir eine von unserem Servlet produzierte Webseite, die ein Formular enthält. Das Entscheidende dabei ist nun aber, dass sich im Kopfteil der HTTP-Antwort eine Set-Cookie-Zeile (oben fett gedruckt) befindet. Ein Cookie ist ein Name-Wert-Paar. Der Name lautet in diesem Fall JSESSIONID, der Wert ist eine längliche Kennung, welche die Session auf dem Server eindeutig identifiziert und die nicht ohne Weiteres zufällig erraten werden kann. Diese Kennung lässt sich vom Session-Objekt übrigens mit der Methode `getId` erfragen. In der Regel benötigt ein Servlet diese Kennung aber nicht.

Sofern bei dem benutzten Browser Cookies nicht deaktiviert sind, schickt der Browser bei jeder neuen Anfrage an denselben Web-Server mit demselben URL-Pfad, der in Set-Cookie enthalten war (/puva), das erhaltene Name-Wert-Paar in einer Cookie-Zeile im Kopfteil der HTTP-Anfrage an den Web-Server (oben fett gedruckt). Damit kann man sich nun leicht vorstellen, wie der Session-Mechanismus implementiert ist. Auf dem Web-Server gibt es eine Session-Tabelle, in der Session-Kennungen mit Session-Objekten assoziiert sind. Beim Aufruf der Methode `getSession` wird geprüft, ob der Kopfteil der HTTP-Anfrage eine Cookie-Zeile enthält, die mit JSESSIONID beginnt. Falls dies der Fall ist, wird der nach dem Gleichheitszeichen folgende Wert in der Session-Tabelle gesucht. Ist er darin enthalten, wird das entsprechende Session-Objekt zurückgeliefert. Andernfalls wird eine neue Session-Kennung und ein neues Session-Objekt erzeugt und in der Session-Tabelle abgelegt. Ferner wird in den Kopfteil der HTTP-Antwort eine entsprechende Set-Cookie-Zeile eingefügt.

### 7.4.3 Direkter Zugriff auf Cookies

Wir haben gesehen, dass Sessions durch Cookies realisiert werden. Die Servlet-Klassenbibliothek bietet auch einen direkten Zugriff auf Cookies an. Es existiert eine Klasse `Cookie`, deren wichtigste Attribute, nämlich der Name und der Wert des Cookies, im Konstruktor gesetzt werden. Daneben haben Cookies weitere Attribute wie z. B. einen Kommentar, einen Pfadnamen, eine maximale Lebenszeit und eine Versionsnummer, die alle mit entsprechenden Setter-Methoden gesetzt und mit entsprechenden Getter-Methoden gelesen werden können:

```
public class Cookie implements Cloneable
{
    public Cookie(String name, String value) {...}
    public String getName() {...}
    public String getValue() {...}
```

```

public void setValue(String newValue) {...}
public String getComment() {...}
public void setComment(String purpose) {...}
public String getPath() {...}
public void setPath(String uri) {...}
public int getMaxAge() {...}
public void setMaxAge(int expiry) {...}
public int getVersion() {...}
public void setVersion(int v) {...}
...
}

```

Cookies werden auf dem Server erzeugt und können mit der Methode *addCookie* des Response-Parameters von doGet und doPost in den Kopfteil einer HTTP-Antwort eingefügt und damit zum Client transportiert werden:

```

public interface HttpServletResponse extends ServletResponse
{
    public void addCookie(Cookie cookie);
    ...
}

```

Wie der Name der Methode addCookie vermuten lässt, können mehrere Cookies in den Kopfteil eingefügt werden. Entsprechend kann der Browser auch mehrere Cookies an den Web-Server in einer HTTP-Anfrage senden. Mit der Methode *getCookies* des Request-Parameters von doGet und doPost lassen sich alle Cookies auslesen:

```

public interface HttpServletRequest extends ServletRequest
{
    public Cookie[] getCookies();
    ...
}

```

Durch die Session-Unterstützung ist in der Regel der direkte Zugriff auf Cookies nicht notwendig. Die für die Realisierung von Sessions benutzten Cookies werden aber vom Browser nicht persistent gespeichert (z.B. auf der Festplatte), sondern sind vergessen, sobald der Browser geschlossen wird. Wenn Cookies an einen Browser gesendet werden sollen, die dort längerfristig gespeichert werden, dann muss man direkt mit Cookies arbeiten. Dabei muss mit der Methode *setMaxAge* der Klasse Cookie die Lebenszeit des Cookies entsprechend eingestellt werden. Das Speichern der Cookies durch den Browser wird allerdings nur dann durchgeführt, wenn die Konfigurationseinstellungen des Browsers dies erlauben.

#### 7.4.4 Servlets mit länger dauernden Aufträgen

Wie schon erwähnt wurde, dient das SleepingServlet in Listing 7.3 (Abschnitt 7.3.1) lediglich der Demonstration von impliziter Parallelität bei Servlets. Für grafische Benutzeroberflächen wurde in Abschnitt 4.5 gezeigt, welche negativen Effekte eine länger dauernde Ereignisbehandlung hat. Dieses Beispiel mündete in der Aufstellung einer Regel, die besagt, dass die Ereignisbehandlung in möglichst kurzer Zeit abgeschlossen sein sollte. Diese Regel gilt auch für Servlets. Ein länger laufendes Servlet ist aus ergonomischen Gesichtspunkten

schlecht; die Benutzerin wird unsicher, wenn der Server mit seiner Antwort so lang braucht, und fragt sich, ob denn noch alles in Ordnung ist. Aus diesem Grund sollte man in solchen Fällen ähnlich vorgehen wie in Kapitel 4 bei der Besprechung solcher Aufgaben im Zusammenhang mit grafischen Benutzeroberflächen: Eine Aufgabe, die länger dauern kann, sollte von einem Servlet in einen separaten Thread ausgelagert werden. Anders als bei grafischen Benutzeroberflächen wird aber dieser Thread nicht veranlassen, dass sich die Benutzeroberfläche ändert. Deshalb wird hier das zuvor verschmähte aktive Warten (Polling) benutzt werden: Das Servlet startet einen Thread und produziert sofort eine Seite, mit der die Benutzerin immer wieder nachsehen kann, ob der gestartete Auftrag schon zu Ende gelaufen ist und falls ja, welches Ergebnis er produziert hat (mit WebSockets sind auch andere Lösungen möglich, s. Abschnitt 7.10). Das wiederholte Nachsehen kann automatisiert werden: Im Kopfteil einer HTTP-Antwort kann mit einer Refresh-Zeile der Browser angewiesen werden, nach einer dort angegebenen Zeit automatisch die aktuelle Seite oder eine ebenfalls in der Refresh-Zeile angegebene Seite (neu) zu laden. Es ist in einem solchen Szenario möglich, dass mehrere Aufträge gleichzeitig laufen. Um sicherzustellen, dass der Status des eigenen Auftrags abgefragt wird, sollte man auch für diesen Fall Sessions oder Cookies einsetzen.

In Listing 7.12, Listing 7.13 und Listing 7.14 wird das Sleep-Beispiel aus Listing 7.3 in der soeben skizzierten Weise realisiert. Das Servlet aus Listing 7.12 dient zum Abrufen des Formulars, mit dem die Laufzeit des Auftrags angegeben werden kann, der mit Sleep simuliert wird. Die Reaktion auf das Absenden des ausgefüllten Formulars erfolgt dann auch in diesem Servlet. Es wird ein Thread des Typs SleepingThread (s. Listing 7.13) gestartet, eine Session erzeugt und in dieser Session wird eine Referenz auf das Thread-Objekt abgelegt. Ferner wird eine HTML-Seite produziert, die besagt, dass die Bearbeitung des Auftrags begonnen hat. Diese Seite wird nur 2 Sekunden angezeigt. Danach lädt der Browser automatisch eine weitere Seite, die durch das Servlet aus Listing 7.14 produziert wird und die den aktuellen Stand der Bearbeitung anzeigt; dieses zweite Servlet ist unter dem Namen „SchlafenAbfragen“ erreichbar.

### **Listing 7.12**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/SchlafenSitzung")
public class SleepingSessionServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                         HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Schlafen mit Abfragen</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Schlafen mit Abfragen</h1>");
```

```
String secsString = request.getParameter("Sekunden");
if(secsString != null)
{
    try
    {
        int secs = Integer.parseInt(secsString);
        if(secs < 0)
        {
            throw new NumberFormatException();
        }
        SleepingThread t = new SleepingThread(secs);
        t.start();
        HttpSession session = request.getSession(true);
        session.setMaxInactiveInterval(10);
        session.setAttribute("SleepingThread", t);
        response.setHeader("Refresh",
                            "2; URL=SchlafenAbfragen");
        out.println("<b>Der Auftrag wurde gestartet!</b>");
        out.println("<>");  
out.println("Sie werden automatisch "
+ "weitergeleitet ...");
        out.println("</body>");
        out.println("</html>");
        return;
    }
    catch(NumberFormatException e)
    {
        out.println("Es muss eine nicht negative Zahl "
+ "eingegeben werden.");
    }
}

out.println("<h2>GET-Formular");
out.println("<form method=\\"get\\">");
out.println("<input name=\\"Sekunden\\">");
out.println("<input type=\\"submit\\" value=\\"Los!\\\">");
out.println("</form>");

out.println("<h2>POST-Formular");
out.println("<form method=\\"post\\">");
out.println("<input name=\\"Sekunden\\">");
out.println("<input type=\\"submit\\" value=\\"Los!\\\">");
out.println("</form>");

out.println("</body>");
out.println("</html>");

}

protected void doPost(HttpServletRequest request,
                      HttpServletResponse response)
throws IOException, ServletException
{
    doGet(request, response);
}
```

**Listing 7.13**

```
class SleepingThread extends Thread
{
    private int secs;
    private String message;

    public SleepingThread(int secs)
    {
        this.secs = secs;
    }

    public void run()
    {
        try
        {
            Thread.sleep(secs * 1000);
            message = "Es wurde " + secs
                      + " Sekunden geschlafen.";
        }
        catch(InterruptedException e)
        {
            message = "Es gab Probleme beim Schlafen.";
        }
    }

    public String getMessage()
    {
        return message;
    }
}
```

Das Servlet aus Listing 7.14 dient zum Abfragen, wie weit die Bearbeitung des Auftrags fortgeschritten ist. Die Abfrage erfolgt in diesem Fall durch die Methode `isAlive` der Thread-Klasse. Falls der Auftrag noch läuft, wird eine entsprechende Meldung ausgegeben und der Browser wird aufgefordert, nach 5 Sekunden dasselbe Servlet nochmals zu aktivieren (in diesem Fall fehlt in der Refresh-Zeile die Angabe der URL). Falls der Auftrag beendet ist, wird die Methode `join` aufgerufen. Dies erscheint auf den ersten Blick unnötig, da man ja schon weiß, dass der Thread zu Ende gelaufen ist. Weil der Zugriff auf das vom Thread errechnete Ergebnis (hier simuliert durch das Attribut `message`) in nicht synchronisierter Weise erfolgt, garantiert der Aufruf von `join`, dass es keinen negativen Effekt gibt, der im Zusammenhang mit `volatile` (s. Abschnitt 2.3.5) erläutert wurde. Im Abschnitt 2.3.6 wird explizit darauf eingegangen, dass man mit `join` diesen Effekt verhindern kann. In dem Servlet wird im Falle der Beendigung des Auftrags ein Link auf das Servlet aus Listing 7.12 erzeugt, wobei hierbei angenommen ist, dass dieses Servlet unter dem Namen „Schlafensitzung“ abrufbar ist. Außerdem wird in diesem Fall die Session durch Aufruf der Methode `invalidate` gelöscht.

**Listing 7.14**

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;
```

```
@WebServlet("/SchlafenAbfragen")
public class SleepingPollingSessionServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Schlafen mit Abfragen</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Schlafen mit Abfragen</h1>");

        HttpSession session = request.getSession(false);
        if(session != null)
        {
            SleepingThread t;
            t = (SleepingThread)session.getAttribute(
                "SleepingThread");
            if(t != null)
            {
                if(t.isAlive())
                {
                    out.println("Der Auftrag ist noch nicht "
                               + "beendet!");
                    response.setHeader("Refresh", "5");
                }
                else
                {
                    out.println("<b>Der Auftrag ist beendet!</b>");
                    out.println("<p>");
                    try
                    {
                        t.join();
                    }
                    catch(InterruptedException e)
                    {
                    }
                    out.print("Das Ergebnis lautet: "
                             + t.getMessage());
                    out.println("<p>");
                    out.println("<a " +
                               + "href=\"SchlafenSitzung\">" +
                               + "Zur&uuml;ck!</a>");
                    session.invalidate();
                }
            }
            else
            {
                out.println("Fehler: Der Auftrag ist unbekannt!");
                out.println("<p>");
                out.println("<a " +
                           + "href=\"SchlafenSitzung\">" +
                           + "Zur&uuml;ck!</a>");
            }
        }
    }
}
```

```
        }
    else
    {
        out.println("Fehler: Es gibt keine Sitzung!");
        out.println("<p>");
        out.println("<a href=\"SchlafenSitzung\">" +
                   + "Zur&uuml;ck!</a>");
    }

    out.println("</body>");
    out.println("</html>");
}
}
```

Die Problematik des längeren Wartens steckt auch in so genannten *Wiki-Anwendungen*. Das sind Anwendungen, bei denen mehreren Personen lesender und ändernder Zugriff auf Texte gewährt wird. Man kann einen Text lesen und über einen speziellen Link ein Eingabeformular bekommen, in dem in einem Editorfenster der bisher erstellte Text steht. Dieser kann nun von einem Benutzer beliebig verändert werden. Durch das Absenden des Formulars wird der Text auf den Web-Server übertragen. Wenn auch das Servlet, welches darauf reagiert, den Text in synchronisierter Weise schreibt, so wird dadurch nicht das Problem umgangen, dass mehrere Benutzer den gleichen Ausgangstext abrufen und gleichzeitig Änderungen an dem Text vornehmen können. Wenn dann die beiden Benutzer ihre Änderungen an den Server abschicken, dann gewinnt derjenige, der dies als Letzter macht; die Änderungen des anderen Benutzers werden einfach überschrieben und sind somit verloren. Abhilfe kann dadurch geschaffen werden, dass man sich merkt, dass ein Benutzer den Text zum Editieren abgerufen hat und ihn als gesperrt auf dem Server kennzeichnet. Wenn nun eine Benutzerin den Text ebenfalls editieren möchte, dann wird sie eine negative Nachricht erhalten. Sie muss dann immer wieder versuchen, den Text abzurufen; ein Warten auf die Freigabe des Textes sollte aus den zu Beginn dieses Abschnitts genannten Gründen eben nicht implementiert werden. Wegen der Zustandslosigkeit des HTTP-Protokolls kann es aber passieren, dass derjenige Benutzer, der den Text zum Editieren abgerufen und damit gesperrt hat, seine Änderungen nie an den Web-Server zurücksendet. Dies würde bewirken, dass der Text immer gesperrt bleibt. Aus diesem Grund muss das Sperren befristet werden. Nun kann es aber passieren, dass ein Benutzer, dessen Sperrfrist abgelaufen ist, den Text doch noch an den Web-Server sendet, obwohl er von einer anderen Benutzerin zum Editieren abgerufen wurde und entsprechend nun für sie gesperrt ist. Um dies zu erkennen, können wieder Sessions oder Cookies eingesetzt werden: Bei jedem Sperren wird eine neue Session bzw. ein neues Cookie erzeugt und an den Browser des Anwenders gesendet. Gleichzeitig wird auf dem Server der Text als gesperrt markiert. Dabei werden auch die Session bzw. das Cookie und die aktuelle Sperrzeit gespeichert. Änderungen am Text werden nur akzeptiert, wenn die HTTP-Anfrage die richtige Session-Kennung bzw. das richtige Cookie vorweist. In diesem Fall wird die Sperre dann freigegeben. Wird ein gesperrter Text ein weiteres Mal zum Editieren angefordert, wird aufgrund der verstrichenen Sperrzeit entschieden, ob eine negative Meldung produziert oder die alte Sperre aufgehoben und durch eine neue ersetzt wird. Die Nutzung von Sperren macht die Nutzung von synchronized dennoch nicht überflüssig.

## ■ 7.5 Asynchrone Servlets

In Abschnitt 7.3 wurde gezeigt, dass in der Regel jede HTTP-Anfrage in einem eigenen Thread ausgeführt wird. Wenn sehr viele Anfragen pro Zeiteinheit beim Server ankommen, die alle verhältnismäßig lange dauern, führt dies zu einer großen Zahl von Threads. Diese hohe Zahl von Threads ist dann unnötig, wenn diese Threads nicht wirklich gleichzeitig ihren Auftrag bearbeiten können oder wenn sie den Auftrag durch weitere Server bearbeiten lassen und somit die meiste Zeit gar nicht aktiv sind, sondern nur auf die Antwort der weiteren Server warten. Wenn in solchen Fällen die große Anzahl von Threads problematisch wird, kann man zu einer asynchronen Auftragsbearbeitung übergehen. Im Folgenden gehen wir von einem Szenario aus, bei dem die Bearbeitung einer HTTP-Anfrage das Versenden eines Unterauftrags an einen zusätzlichen Server (zum Beispiel einen Datenbank-Server) bewirkt.

Dabei ist das Entscheidende, dass der Thread, welcher eine HTTP-Anfrage bearbeitet, nach dem Versenden des Unterauftrags nicht auf dessen Ergebnis wartet, um danach die Antwort (in der Regel in Form einer HTML-Seite) zu generieren und an den ursprünglichen Anfrager zurückzusenden, sondern nach dem Versenden des Unterauftrags terminiert oder frei ist, neue HTTP-Anfragen entgegenzunehmen. Ein oder mehrere, aber tendenziell eher wenige Threads warten auf die Ergebnisse der Unteraufträge, erzeugen die dazugehörigen Antwortseiten und schließen damit die zuvor unterbrochene Bearbeitung der HTTP-Anfragen ab. Die Anzahl der gleichzeitig aktiven Threads wird dadurch deutlich reduziert, weil es nicht pro abgesendem Unterauftrag jeweils einen Thread gibt, der auf die Antwort zu diesem Unterauftrag wartet.

Zur Umsetzung dieser Idee stellt der Request-Parameter der Methoden doGet bzw. doPost die parameterlose Methode `startAsync` bereit. Diese Methode liefert ein Objekt des Typs `AsyncContext` zurück, das man speichern kann und von dem man sich später z. B. das Response-Objekt beschaffen kann, das den PrintWriter zum Schreiben der Antwort zur Verfügung stellt.

In Listing 7.15 wird dieses Prinzip anhand eines Beispiels illustriert. Dabei wird die Bearbeitung des Unterauftrags über eine `DelayQueue` simuliert. DelayQueue ist eine Klasse der Concurrent-Bibliothek (s. Abschnitt 3.7.5), in die man Elemente einstellen kann, die erst nach einer bestimmten Verzögerungszeit der Queue entnommen werden können. Im Beispiel entspricht das Einstellen eines Elements in die DelayQueue dem Versenden des Unterauftrags an den weiteren Server. Wenn die angegebene Verzögerungszeit vergangen ist, kann der Auftrag aus der Queue entnommen werden; dies entspricht dem Eintreffen des Unterauftragergebnisses vom Zusatz-Server. Zur Verarbeitung dieser Ergebnisse wird – neben der DelayQueue selbst – ein einziger Thread vom Typ `SingleServletWorker` in der Initialisierungsmethode des Servlets erzeugt. Elemente, die in eine DelayQueue eingestellt werden, müssen die Schnittstelle `Delayed` implementieren, die als einzige Methode die Methode `getDelay` besitzt. Diese Methode soll die noch verbleibende Verzögerungszeit zurückgeben. Da `Delayed` aus `Comparable` abgeleitet ist, muss neben `getDelay` auch die Methode `compareTo` implementiert werden, die zwei `Delayed`-Elemente miteinander vergleicht. In Listing 7.15 sind die Elemente, die `Delayed` implementieren und in die Delay-Queue eingereiht werden, vom Typ `Job`. Die Klasse `Job` stellt den Unterauftrag dar, in dem

u.a. der AsyncContext gespeichert wird, um die Bearbeitung der HTTP-Anfrage später wieder aufgreifen und abschließen zu können.

**Listing 7.15**

```
import java.io.*;
import java.util.concurrent.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.servlet.annotation.*;

@WebServlet(value="/SchlafenAsync", asyncSupported=true,
           loadOnStartup=1)
public class SleepingServletAsync extends HttpServlet
{
    private DelayQueue<Job> queue;
    private SingleServletWorker worker;

    public void init()
    {
        queue = new DelayQueue<>();
        worker = new SingleServletWorker(queue);
        worker.start();
    }

    public void destroy()
    {
        worker.interrupt();
    }

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Asynchrones Schlafen</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Asynchrones Schlafen</h1>");
        out.println("<h3>Das Servlet wird ausgeführt von " +
                   "Thread \"" + Thread.currentThread().getName() + 
                   "\".</h3>");

        String secsString = request.getParameter("Sekunden");
        if(secsString != null)
        {
            try
            {
                long msecs = Integer.parseInt(secsString) * 1000L;
                if(!request.isAsyncSupported())
                {
                    request.setAttribute(
                        "org.apache.catalina.Async_SUPPORTED",
                        true);
                }
            }
        }
    }
}
```

```
        AsyncContext asyncContext = request.startAsync();
        asyncContext.setTimeout(msecs + 1000L);
        Job job = new Job(msecs, asyncContext);
        queue.put(job);
    }
    catch(NumberFormatException e)
    {
        out.println("<p>Keine/falsche Sekundenangabe!</p>");
        out.println("</body>");
        out.println("</html>");
    }
}
else
{
    out.println("<h2>GET-Formular</h2>");
    out.println("<p><form method=\"get\">");
    out.println("<input name=\"Sekunden\">");
    out.println("<input type=\"submit\" value=\"Los!\">");
    out.println("</form></p>");
}
}
}

class Job implements Delayed
{
    private static long jobCounter;

    private long startTime;
    private long delay;
    private AsyncContext asyncContext;
    private long jobNumber;

    public Job(long delay, AsyncContext asyncContext)
    {
        this.startTime = System.currentTimeMillis();
        this.delay = delay;
        this.asyncContext = asyncContext;
        this.jobNumber = getNextJobNumber();
    }

    private static synchronized long getNextJobNumber()
    {
        return ++jobCounter;
    }

    public long getDelay()
    {
        return delay;
    }

    public AsyncContext getAsyncContext()
    {
        return asyncContext;
    }

    public long getJobNumber()
    {
        return jobNumber;
    }
}
```



```

        "\" erzeugt.</p>");
    out.println("<p>Damit ist Job Nr. " +
                job.getJobNumber() + " abgearbeitet.</p>");
    out.println("</body>");
    out.println("</html>");
    asyncContext.complete();
}
System.out.println("Worker-Thread hat Schleife verlassen.");
}
}

```

Bitte beachten Sie, dass in der @WebServlet-Annotation angegeben werden muss, dass von der Möglichkeit der asynchronen Auftragsbearbeitung Gebrauch gemacht wird.

## ■ 7.6 Filter

In manchen Anwendungen, die aus mehreren Servlets bestehen, müssen für jede Auftragsbearbeitung immer dieselben Aufgaben (wie z.B. Zugriffskontrolle oder Protokollierung) durchgeführt werden. Für derartige Aufgaben bieten sich Servlet-Filter an. Ein Filter ist ein Programmstück, das vor dem Aufruf eines Servlets ausgeführt wird. Dabei kann man mehrere Filter definieren, die in einer Art Kette der Reihe nach aufgerufen werden. Ein Filter kann allerdings auch die Ausführung der weiteren Filter und damit auch des eigentlichen Servlets unterbinden. Insofern ist der Begriff Filter angebracht. Die Vorstellung einer reinen Filterfunktion führt allerdings in die falsche Richtung. So könnte ein Filter beispielsweise auch den Request- oder den Response-Parameter verändern.

Wie ein Servlet muss ein Filter konfiguriert werden, wobei wir auch dafür wieder Annotations verwenden. Wie für ein Servlet muss dabei für einen Filter in der Annotation `@WebFilter` angegeben werden, auf welche URL-Muster er reagiert. Ein Filter muss die Schnittstelle Filter mit den Methoden init, doFilter und destroy implementieren. Aufgrund der Namen sollte die Bedeutung dieser Methoden klar sein. Neben einem Request- und Response-Parameter (der Typ dieser Parameter ist etwas allgemeiner, steht also in der Vererbungshierarchie oberhalb von `HttpServletRequest` und `HttpServletResponse`) ist der dritte Parameter der Methode doFilter vom Typ `FilterChain`. Dieser Parameter kann dazu benutzt werden, die Auftragsbearbeitung an den nächsten Filter bzw. an das eigentlich auszuführende Servlet, falls alle Filter schon an der Reihe waren, weiterzureichen.

In Listing 7.16 sehen Sie einen ganz einfachen Filter, der lediglich eine Zeile auf die Standardausgabe schreibt. Außer dem Vorhandensein dieser Klasse in der Web-Anwendung muss nichts Weiteres unternommen werden, damit der Filter aktiv wird. Dieser Filter wird für alle Servlets Ihrer Anwendung aufgerufen, da sein URL-Muster der Wildcard \* ist.

### **Listing 7.16**

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;
import javax.annotation.*;

```

```
@WebFilter(filterName = "ReportFilter", urlPatterns = {"/*"})
public class ReportFilter implements Filter
{
    public void init(FilterConfig filterConfig)
        throws ServletException
    {
    }

    public void destroy()
    {
    }

    public void doFilter(ServletRequest request,
                        ServletResponse response,
                        FilterChain chain)
        throws IOException, ServletException
    {
        HttpServletRequest req = (HttpServletRequest) request;
        System.out.println(req.getRemoteHost() + ": Zugriff auf " +
                           req.getRequestURL() + ".");
        chain.doFilter(request, response);
    }
}
```

Zur Übung können Sie einen zweiten Filter mit einer ähnlichen, aber anderen Meldung anlegen. Sie werden dann beide Zeilen beim Aufruf aller Ihrer Servlets bei dieser Anwendung sehen. Sie können außerdem auch die Implementierung so ändern, dass der Filter seinem Namen alle Ehre macht, indem Sie zum Beispiel nur noch jeden zweiten Aufruf von doFilter an die FilterChain weiterreichen.

## ■ 7.7 Übertragung von Dateien mit Servlets

Dateiübertragungen in einem Client-Server-Umfeld werden in der Regel mit den Begriffen Hochladen (Upload) und Herunterladen (Download) von Dateien bezeichnet. Hier geht man von der Vorstellung aus, dass der Client tiefer und der Server höher positioniert ist. Dementsprechend meint man mit Hochladen das Übertragen einer Datei vom Client zum Server und mit Herunterladen das Übertragen einer Datei vom Server zum Client. In diesem Abschnitt betrachten wir beide Möglichkeiten im Servlet-Umfeld.

### 7.7.1 Herunterladen von Dateien

Wir beginnen unsere Betrachtung mit dem Herunterladen von Dateien. Dabei handelt es sich um nichts, was wesentlich anders wäre als das, was in den bisherigen Beispielen vorkam. Bisher wurde nämlich immer eine HTML-Seite vom Server zum Client übertragen, wobei die Tatsache, ob die Inhalte der HTML-Seite vom Servlet aus einer Datei gelesen oder

auf andere Weise erzeugt werden, für die jetzige Betrachtung irrelevant ist. Das heißt also: Es wurde bereits besprochen, wie HTML-„Dateien“ mittels Servlets heruntergeladen werden können. Wenn also nun andere Dateitypen vom Server zum Client übertragen werden sollen, muss das Servlet lediglich einen anderen Content-Type setzen und einen dazu passenden Inhalt erzeugen. Je nach Dateityp kann der Browser den Inhalt dann direkt in seinem Fenster anzeigen. Oder alternativ öffnet der Browser einen Dialog, mit dem eine Benutzerin auswählen kann, ob sie die Datei abspeichern oder mit einem Programm außerhalb des Browsers öffnen möchte.

In Listing 7.17 ist der Programmcode eines Servlets gezeigt, das eine „normale“ Textdatei erzeugt (Content-Type: text/plain). Das Servlet wird aktiviert, wenn ein Formular (s. Bild 7.3) mit einem Eingabefeld namens lines ausgefüllt wurde. Darin kann der Benutzer eine (positive) Zahl eingeben, mit der er bestimmt, wie viele Zeilen die erzeigte Textdatei enthalten soll.

### Listing 7.17

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;

@WebServlet("/Herunterladen")
public class DownloadServlet extends HttpServlet
{
    protected void doGet(HttpServletRequest request,
                          HttpServletResponse response)
        throws IOException, ServletException
    {
        String linesString = request.getParameter("lines");
        int numberofLines = 0;
        try
        {
            numberofLines = Integer.parseInt(linesString);
            if(numberofLines <= 0)
            {
                throw new NumberFormatException();
            }
        }
        catch(NumberFormatException e)
        {
            response.setStatus(HttpServletResponse.SC_BAD_REQUEST);
            response.setContentType("text/html");
            PrintWriter out = response.getWriter();
            out.println("<html>");
            out.println("<head>");
            out.println("<title>Fehler</title>");
            out.println("</head>");
            out.println("<body>");
            out.println("<h3>Der eingegebene Text '" + linesString +
                       "' ist keine positive ganze Zahl.</h3>");
            out.println("</body>");
            out.println("</html>");
            return;
        }
    }
}
```

```

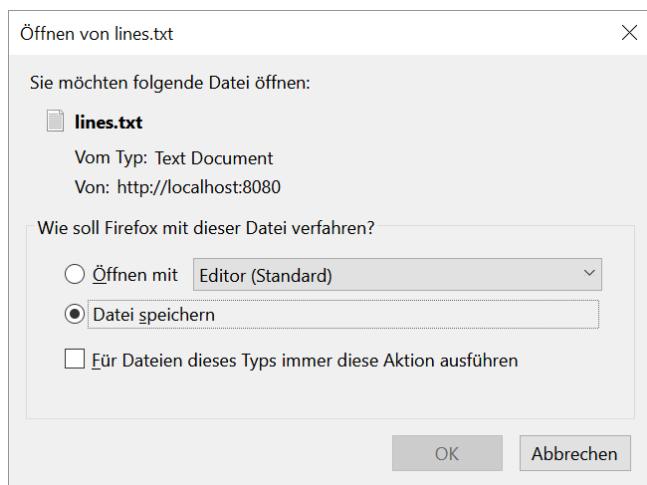
        response.setContentType("text/plain");
        response.setHeader("Content-Disposition",
                           "attachment; filename=\"lines.txt\"");
        PrintWriter out = response.getWriter();
        for(int i = 1; i <= numberOfLines; i++)
        {
            out.println("Zeile " + i);
        }
    }
}

```



**Bild 7.3** Formular zur Eingabe der Zeilenzahl der zu erzeugenden Datei

Nach einer korrekten Eingabe (wie dies im Beispiel aus Bild 7.3 der Fall ist) öffnet der Browser beispielsweise den in Bild 7.4 gezeigten Dialog, mit dem man auswählen kann, ob die Datei gespeichert oder mit einem Programm wie einem einfachen Texteditor geöffnet werden soll. Falls keine korrekte Eingabe erkannt wird, produziert das Servlet eine Fehlerseite.



**Bild 7.4**

Firefox-Dialog zum Auswählen, ob Datei geöffnet oder gespeichert werden soll

Für andere Dateitypen wie etwa PDF, PNG, JPG oder Excel ist das Vorgehen prinzipiell gleich. In diesem Fall ist es in der Regel jedoch etwas aufwändiger, den dazu passenden Inhalt zu erzeugen. Jedoch gibt es hierzu für viele Formate entsprechende Klassenbibliotheken, die die Entwicklerinnen dabei unterstützen.

## 7.7.2 Hochladen von Dateien

Um Dateien hochladen zu können, benötigt man in einem Formular ein spezielles Eingabefeld, mit dem man eine Datei auf dem lokalen Dateisystem des Clients angeben kann:

```
<input name="..." type="file"/>
```

Das folgende Formular ist ein Eingabeformular, in dem man seinen Vornamen und seinen Nachnamen eingeben sowie eine Datei, die den Lebenslauf enthalten soll, hochladen kann:

```
<form method="post" enctype="multipart/form-data"
      action="Hochladen">
<p>Vorname: <input name="vorname" type="text"/></p>
<p>Nachname: <input name="nachname" type="text"/></p>
<p>Lebenslauf: <input name="lebenslauf" type="file"/></p>
<p><input type="submit" value="Jetzt hochladen!" /></p>
</form>
```

Entscheidend ist neben dem Eingabefeld des Typs „file“ auch, dass im Form-Tag als Codierungsart (enctype) „multipart/form-data“ angegeben wird. Bild 7.5 zeigt, wie das Formular in einem Browser dargestellt wird. Durch das Drücken auf den Button „Durchsuchen . . .“ erhält man einen Dialog wie allgemein üblich beim Öffnen von Dateien. Mit Hilfe dieses Dialogs kann man sich durch das lokale Dateisystem hangeln und eine Datei auswählen. Der Button „Durchsuchen . . .“ muss nicht gesondert angegeben werden, sondern er kommt mit dem Eingabefeld des Typs „file“ automatisch mit.

Hochladen einer Datei

Vorname: Rainer

Nachname: Oechsle

Lebenslauf: Durchsuchen... lines.txt

Jetzt hochladen!

**Bild 7.5** Formular mit der Möglichkeit, eine Datei zum Hochladen auszuwählen

Das Servlet, welches auf das Absenden des Formulars reagiert, ist das UploadServlet in Listing 7.18. Es verarbeitet die eingehenden Daten nicht, sondern zeigt lediglich zur Demonstration an, welche Daten angekommen sind.

**Listing 7.18**

```

import java.io.*;
import javax.servlet.*;
import javax.servlet.annotation.*;
import javax.servlet.http.*;
import java.util.*;

@WebServlet("/Hochladen")
public class UploadServlet extends HttpServlet
{
    protected void doPost(HttpServletRequest request,
                          HttpServletResponse response)
            throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Antwort auf Hochladen" + "</title>");
        out.println("</head>");
        out.println("<body>");
        out.println("<h1>Antwort auf Hochladen</h1>");
        out.println("<h3>Folgende Kopffelder sind " +
                   "angekommen:</h3>");
        out.println("<pre>");

        Enumeration<String> headers = request.getHeaderNames();
        while(headers.hasMoreElements())
        {
            String header = int seheaders.nextElement();
            String hvalue = request.getHeader(header);
            out.println(header + ": " + hvalue);
        }
        out.println("</pre>");

        out.println("<h3>Folgende Daten sind " +
                   "+ "angekommen:</h3>");
        out.println("<pre>");
        BufferedReader br = request.getReader();
        String line;
        while((line = br.readLine()) != null)
        {
            out.println(line); // --> Ausgabe der Datei
        }
        out.println("</pre>");
        out.println("</body></html>");
    }
}

```

Wie man dem Programmcode entnehmen kann, werden zuerst alle Kopffelder ausgelesen, deren Wert bestimmt und diese Daten in die erzeugte HTML-Seite übernommen. Im zweiten Teil des Servlets wird auf den Parameter `request` die Methode `getReader` angewendet, welche einen `BufferedReader` zurückliefert. Dies ist ganz analog zum Aufruf von `getWriter` auf den Parameter `response`. Wie man über den dadurch benutzbaren `PrintWriter` auf den Datenteil der HTTP-Antwort schreibend zugreifen kann, so kann man über den `Buffered-`

Reader den Datenteil der HTTP-Anfrage zeilenweise lesen. Die durch das Demo-Servlet gelesenen Daten werden einfach in die produzierte HTML-Antwortseite geschrieben. Ein Beispiel einer solchen Antwortseite zeigt Bild 7.6.

The screenshot shows a Firefox browser window with the title "Antwort auf Hochladen". The address bar shows "localhost:8080/puva/Hochladen". The page content is titled "Antwort auf Hochladen" and contains the following text:

**Folgende Kopffelder sind angekommen:**

```
host: localhost:8080
user-agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) Gecko/20100101 Firefox/55.0
accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
accept-language: de,en-US;q=0.7,en;q=0.3
accept-encoding: gzip, deflate
referer: http://localhost:8080/puva/Upload.html
content-type: multipart/form-data; boundary=-----41184676334
content-length: 423
cookie: JSESSIONID=881280D5E9CB559993E59FD5C8177E28
dnt: 1
connection: keep-alive
upgrade-insecure-requests: 1
```

**Folgende Daten sind angekommen:**

```
-----41184676334
Content-Disposition: form-data; name="vorname"
Rainer
-----41184676334
Content-Disposition: form-data; name="nachname"
Oechsle
-----41184676334
Content-Disposition: form-data; name="upload"; filename="lines.txt"
Content-Type: text/plain

Zeile 1
Zeile 2
Zeile 3
Zeile 4
-----41184676334--
```

**Bild 7.6** Antwortseite des Upload-Servlets

Wie man sieht, werden die drei Teile des Eingabeformulars durch eine spezielle Zeichenkette, die mit vielen „-“ beginnt, voneinander getrennt. Diese Zeichenkette wird im Kopfteil der HTTP-Anfrage in der Zeile „content-type“ als „boundary“ bekannt gegeben. Der Content-Type ist übrigens – wie im Formular eingestellt – als „multipart/form-data“ angegeben. Jeder der drei Teile, die durch die spezielle Zeichenkette voneinander getrennt sind, hat wiederum das Format: beliebig viele Kopfzeilen – Leerzeile – Datenteil. Die Kopfzeilen enthalten immer den Namen des jeweiligen Eingabefelds. Im dritten Teil ist außerdem noch der Dateiname der hochgeladenen Datei sowie in einer weiteren Kopfzeile der Typ der Datei („text/plain“) erkennbar. Nach der Leerzeile folgt der Inhalt der Datei.

Will man die Daten verarbeiten, müssen die gezeigten Eingabedaten, die man über den BufferedReader liest, entsprechend analysiert werden, was einen gewissen Aufwand verur-

sacht. Es gibt aber frei verfügbare Klassen, die das Parsen der Eingabedaten unterstützen und den Programmierern dabei viel Arbeit abnehmen.

## ■ 7.8 JSF (Java Server Faces)

Bei Servlets wird HTML-Text in Java-Code eingebettet. Bei größeren Webseiten bläht dies den Programmcode durch die vielen bzw. langen Ausgabeanweisungen auf. Außerdem wird dadurch verhindert, dass der HTML-Teil mit HTML-Editoren erstellt und gewartet werden kann. Da sich um den HTML-Teil häufig Personen kümmern, die keine größeren Kenntnisse im Bereich der Software-Entwicklung haben, ist eine deutlichere Trennung des HTML- und des Programmcode-Teils wünschenswert. Eine inzwischen veraltete Lösung zur Erreichung dieses Ziels stellt *JSP (Java Server Pages)* dar. In diesem Buch wird *JSF (Java Server Faces)* behandelt, welches die aktuell favorisierte Lösung zur Trennung von HTML und Java-Code darstellt.

Die Grundidee von JSF sieht so aus: In die HTML-Seiten, die in dem speziellen Dialekt namens *XHTML* formuliert werden, werden in der Sprache *EL (Expression Language)* spezielle Ausdrücke eingebettet, die mit einem dazugehörigen Java-Code zusammenwirken. Man kann so u.a. Felder eines Formulars an Attribute von Java-Objekten koppeln, was sich sowohl bei der Ausgabe als auch bei der Eingabe auswirkt. Bei der Ausgabe, also wenn die Web-Seite vom Web-Server erzeugt und zum Browser gesendet wird, werden die Attributwerte des Objekts in das Formular übernommen. Bei der Eingabe, also wenn die Benutzerin das Formular ausgefüllt und zum Web-Server zurückgeschickt hat, werden die Formularinhalte in die Attribute des Objekts übernommen. Dieses Prinzip schauen wir uns am besten anhand eines Beispiels an.

### 7.8.1 Einführendes Beispiel

Das folgende Beispiel ist eine sehr einfache Anwendung zum Anzeigen eines Formulars (s. Bild 7.7 links). Nach dem Ausfüllen des Formulars werden der Einfachheit lediglich die eingegebenen Daten auf einer Ergebnisseite angezeigt (s. Bild 7.7 rechts). Im Prinzip ist die Anwendung derjenigen aus Unterabschnitt 7.2.3 sehr ähnlich, bei der es um ein Teeformular ging und die mit Servlets realisiert wurde.

**Anmeldung für Studierende**

Vorname:

Nachname:

Geboren in: Dänemark

Wohnhaft in: Dänemark

Bevorzugte Programmiersprache:

C++  C#  Delphi  Java  JavaScript  Python

Programmiersprachen, die ebenfalls beherrscht werden:

C++  C#  Delphi  Java  JavaScript  Python

**Bestätigung der Anmeldung**

Name: Rainer Oechsle

Geboren in: Deutschland

Wohnhaft in: Deutschland

Bevorzugte Programmiersprache: Java

Andere Programmiersprachen:

- C++
- Python

**Bild 7.7** Web-Seiten für Formular (links) und Ergebnis nach dem Absenden (rechts)

Die HTML-Seite student-form.xhtml, welche das Formular anzeigt, sieht so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:h="http://java.sun.com/jsf/html">

    <h:head>
        <title>Anmeldung f&uuml;r Studierende</title>
    </h:head>
    <h:body>
        <h1>Anmeldung f&uuml;r Studierende</h1>
        <h:form>
            <p>Vorname: <h:inputText value="#{student.firstName}" /></p>
            <p>Nachname: <h:inputText value="#{student.lastName}" /></p>
            <p>Geboren in:<br/>
                <h:selectOneMenu value="#{student.bornInCountry}">
                    <f:selectItem itemLabel="Dänemark"
                                  itemValue="Dänemark"/>
                    <f:selectItem itemLabel="Deutschland"
                                  itemValue="Deutschland"/>
                    ...
                    <f:selectItem itemLabel="Spanien" itemValue="Spanien"/>
                </h:selectOneMenu>
            </p>
            <p>Wohnhaft in:<br/>
                <h:selectOneMenu value="#{student.livingInCountry}">
                    <f:selectItems value="#{options.countryOptions}" />
                </h:selectOneMenu>
            </p>
        </h:form>
    </h:body>
</html>
```

```

<p>Bevorzugte Programmiersprache:
    <h:selectOneRadio value="#{student.progLanguage}">
        <f:selectItems value="#{options.languageOptions}" />
    </h:selectOneRadio>
</p>
<p>Programmiersprachen, die ebenfalls beherrscht werden:
    <h:selectManyCheckbox
        value="#{student.otherProgLanguages}">
        <f:selectItems value="#{options.languageOptions}" />
    </h:selectManyCheckbox>
</p>
<p>
    <h:commandButton value="Absenden"
        action="student-response" />
</p>
</h:form>
</h:body>
</html>
```

Ohne auf die XHTML-spezifischen Details einzugehen sehen wir, dass im Wesentlichen ein Formular mit unterschiedlichen Eingabemöglichkeiten definiert wird. Durch h:inputText entsteht ein einzeiliges Texteingabefeld im Formular. Während wir bisher unseren Eingabefeldern Namen gaben, wird jetzt über value in der Sprache EL (Expression Language) ein Bezug zu einem bestimmten Attribut (im Beispiel firstName und lastName) eines Objekts student hergestellt. Damit dies funktioniert, benötigen wir in unserer Web-Anwendung eine sogenannte *ManagedBean-Klasse* (s. später) namens Student mit den Methoden getFirstName, getLastname, setFirstName und setLastName. Mit h:selectOneMenu wird eine ComboBox in das Formular gesetzt, die die Auswahl genau eines Werts erlaubt. Auch hier wird wieder eine Beziehung hergestellt zu Attributen des Objekts student. Beim ersten derartigen Auswahlelement werden die möglichen Werte explizit über Elemente der Art f:selectItem aufgezählt, wobei unter itemLabel immer der in der ComboBox dargestellte Text und unter itemValue der später dem Objekt student zugewiesene Wert, falls der betreffende Eintrag ausgewählt wurde, angegeben ist (hier in diesem Fall ist itemLabel und itemValue immer jeweils gleich). Bei der zweiten Verwendung von h:selectOneMenu werden die auswählbaren Einträge nicht aufgeführt, sondern über ein Java-Objekt beschafft. Damit der EL-Ausdruck #{options.countryOptions} wie erwartet funktioniert, brauchen wir eine ManagedBean-Klasse Options mit einer Methode getCountryOptions. Das folgende Element h:selectOneRadio ist h:selectOneMenu sehr ähnlich, nur dass die Einfachauswahlmöglichkeit über RadioButtons realisiert wird, bei denen höchstens einer selektiert sein kann. Mit h:selectManyCheckbox wird eine Mehrfachauswahlmöglichkeit mittels CheckBoxes in das Formular gesetzt. Schließlich wird durch h:commandButton ein Button zum Absenden des Formulars definiert. Neben der Beschriftung wird student-response als action spezifiziert. Dies bedeutet, dass beim Absenden das Dokument student-response.xhtml angefordert wird. Dieses sieht so aus:

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
    xmlns:f="http://java.sun.com/jsf/core"
    xmlns:h="http://java.sun.com/jsf/html"
```

```

xmlns:ui="http://xmlns.jcp.org/jsf/facelets"

<h:head>
    <title>Bestätigung der Anmeldung</title>
</h:head>
<h:body>
    <h1>Bestätigung der Anmeldung</h1>
    <p>Name: #{student.firstName} #{student.lastName}</p>
    <p>Geboren in: #{student.bornInCountry}</p>
    <p>Wohnhaft in: #{student.livingInCountry}</p>
    <p>Bevorzugte Programmiersprache: #{student.progLanguage}</p>
    <p>Andere Programmiersprachen:</p>
    <ul>
        <ui:repeat value="#{student.otherProgLanguages}"
            var="tempLang">
            <li>#{tempLang}</li>
        </ui:repeat>
    </ul>
</h:body>
</html>

```

Wir sehen, dass zur Ausgabe dieselben EL-Ausdrücke wie in der vorigen XHTML-Seite verwendet werden wie zum Beispiel `#{student.firstName}`. Von besonderem Interesse ist die durch `ui:repeat` definierte Schleife. Hier wird mit `#{student.otherProgLanguages}` ein String-Array angegeben, das durchlaufen wird. Bei jedem Durchlauf wird das aktuelle String-Element einer Variablen zugewiesen, die unter `var` festgelegt wird (im Beispiel ist es `tempLang`). Der jeweilige Wert dieser Variablen kann dann im Schleifenrumpf wieder in Form eines EL-Ausdrucks genutzt werden.

Damit der Bezug zwischen HTML und Java-Code funktioniert, benötigen wir in diesem Fall in unserer JSF-Web-Anwendung zwei Klassen namens `Student` und `Options`, die mit der Annotation `@ManagedBean` versehen sind. Der Code ist in gekürzter Form in Listing 7.19 und Listing 7.20 dargestellt.

### **Listing 7.19**

```

import javax.faces.bean.*;

@ManagedBean
public class Student
{
    private String firstName;
    private String lastName;
    private String bornInCountry;
    private String livingInCountry;
    private String progLanguage;
    private String[] otherProgLanguages;

    //Getter- und Setter-Methoden für alle Attribute
}

```

**Listing 7.20**

```

import java.util.ArrayList;
import java.util.List;
import javax.faces.bean.*;

@ManagedBean
@ApplicationScoped
public class Options
{
    private List<String> countryOptions;
    private List<String> languageOptions;

    public Options()
    {
        countryOptions = new ArrayList<>();
        countryOptions.add("D&uuml;nemark");
        countryOptions.add("Deutschland");
        ...
        countryOptions.add("Spanien");

        languageOptions = new ArrayList<>();
        languageOptions.add("C++");
        languageOptions.add("C#");
        ...
        languageOptions.add("Python");
    }

    public List<String> getCountryOptions()
    {
        return countryOptions;
    }

    public List<String> getLanguageOptions()
    {
        return languageOptions;
    }
}

```

Wir können übrigens noch folgendes Experiment durchführen und der Klasse Student aus Listing 7.19 einen Konstruktor hinzufügen, der beispielsweise so aussehen könnte:

```

public Student()
{
    firstName = "Vorname";
    lastName = "Nachname";
    bornInCountry = "Deutschland";
    livingInCountry = "Deutschland";
    progLanguage = "Java";
}

```

In diesem Fall wären dann in dem angeforderten Formular (s. Bild 7.7 links) schon die entsprechenden Werte in den Eingabefeldern eingetragen bzw. die entsprechenden Optionen bei den ComboBoxes und RadioButtons vorausgewählt.

Damit der Web-Server die XHTML-Dateien nicht einfach so an den Browser zurückliefert, wie sie im Dateisystem vorliegen, muss ein zusätzlicher Verarbeitungsvorgang dazwischen

geschaltet werden, in dem u.a. die speziellen Tags, die mit h:, f: oder ui: beginnen sowie die darin enthaltenen EL-Ausdrücke interpretiert werden. Aus diesem Grund sollte man die Datei nicht wie andere HTML- oder PNG-Dateien unter ihrer üblichen URL, sondern unter einer ganz speziellen URL beim Web-Server anfordern. In den Konfigurationseinstellungen des JSF-Web-Projekts kann üblicherweise das URL-Muster spezifiziert werden, für das HTTP-Requests an ein spezielles Servlet namens FacesServlet weitergeleitet werden. Dieses Servlet ist ein zentraler Bestandteil der Realisierung des JSF-Mechanismus und interpretiert die EL-Ausdrücke in den XHTML-Dateien. Die Voreinstellung für dieses Muster lautet /faces/\*. Wenn wir davon ausgehen, dass diese Voreinstellung in unserem Beispielprojekt gewählt wurde, der Web-Server auf dem Rechner www.abc.de unter dem Port 8080 erreichbar ist, das JSF-Web-Projekt puva heißt und unsere XHTML-Dateien im WebContent-Bereich in einem Unterverzeichnis namens student liegen, dann kann die Web-Seite zu student-form.xhtml unter der URL <http://www.abc.de:8080/puva/faces/student/student-form.xhtml> angefordert werden. In dieser Datei steht als action student-response, wobei dieser String automatisch zu student-response.xhtml ergänzt wird. Da diese Angabe immer noch eine relative URL ist, wird daraus eine absolute URL, indem der vordere Teil durch den vorderen Teil der URL ergänzt wird, von der die aktuelle Seite student-form.xhtml angefordert wurde. Aus student-response wird also <http://www.abc.de:8080/puva/faces/student/student-response.xhtml>.

Wenn man sich den Seitenquelltext auf seinem Browser anschaut, dann erkennt man, dass das FacesServlet aus der XHTML-Datei ganz normalen HTML-Text erzeugt. Aus dem Formular in student-form.xhtml (zwischen <h:form> und </h:form>) wird so beispielsweise (in gekürzter Form):

```
<form id="j_idt6" name="j_idt6" method="post"
action="/puva/faces/student/student-form.xhtml"
enctype="application/x-www-form-urlencoded">
<input type="hidden" name="j_idt6" value="j_idt6" />
<p>Vorname: <input type="text" name="j_idt6:j_idt8" /></p>
<p>Nachname: <input type="text" name="j_idt6:j_idt10" /></p>
<p>Geboren in:
<select name="j_idt6:j_idt12" size="1">
<option value="Dänemark">Dänemark</option>
...
<option value="Spanien">Spanien</option>
</select>
</p>
<p>Wohnhaft in:
<select name="j_idt6:j_idt23" size="1">
<option value="Dänemark">Dänemark</option>
...
<option value="Spanien">Spanien</option>
</select>
</p>
<p>Bevorzugte Programmiersprache:
<input type="radio" name="j_idt6:j_idt26"
id="j_idt6:j_idt26:0" value="C++" />
<label for="j_idt6:j_idt26:0"> C++</label>
...
<input type="radio" name="j_idt6:j_idt26"
id="j_idt6:j_idt26:5" value="Python" />
<label for="j_idt6:j_idt26:5"> Python</label></td>
```

```

</p>
<p>Programmiersprachen, die ebenfalls beherrscht werden:
    <input name="j_idt6:j_idt29" id="j_idt6:j_idt29:0"
        value="C++" type="checkbox" />
    <label for="j_idt6:j_idt29:0" class=""> C++</label>
    ...
    <input name="j_idt6:j_idt29" id="j_idt6:j_idt29:5"
        value="Python" type="checkbox" />
    <label for="j_idt6:j_idt29:5" class=""> Python</label>
</p>
<p><input type="submit" name="j_idt6:j_idt32" value="Absenden" />
</p>
<input type="hidden" name="javax.faces.ViewState"
    id="j_id1:javax.faces.ViewState:0" value="..." autocomplete="off" />
</form>

```

Man kann erkennen, dass außer der Wandlung in die üblichen Eingabemöglichkeiten für Formulare wie ComboBoxes, RadioButtons und CheckBoxes zwei zusätzliche versteckte Felder (Hidden Fields) über `<input type="hidden" ...>` in das Formular eingebaut werden. Das erste versteckte Feld befindet sich am Anfang gleich nach dem Tag `<form ...>`. Damit wird ein Name für das Formular definiert, wobei Name und Wert gleich sind. Am Ende wird ein zweites verstecktes Feld mit dem Namen `javax.faces.ViewState` angelegt, dessen Wert ein sehr langer String ist und somit schwer erraten werden kann (ähnlich wie eine Session-Kennung, oben durch `...` ersetzt). Außerdem ist zu sehen, dass dem Command Button ein Name gegeben wird, so dass Name und Wert des Command Buttons beim Klicken mit als Formulardaten an den Server übertragen werden.

Statt die Antwortseite fest in der XHTML-Seite zu codieren, wie dies durch `action="student-response"` erfolgt ist (bitte beachten Sie auch, was unter `action` im Tag `<form>` in der in HTML übersetzten Version oben steht), könnte man die Antwortseite auch dynamisch festlegen, indem man als `action` über einen EL-Ausdruck einen Methodenaufruf spezifiziert. Dazu wird die Definition des Command Buttons beispielsweise wie folgt geändert:

```
<h:commandButton value="Absenden" action="#{student.handleRequest}"/>
```

In diesem Fall brauchen wir in der Student-Klasse aus Listing 7.19 eine zusätzliche Methode namens `handleRequest`, die beispielsweise so aussehen könnte:

```

public String handleRequest()
{
    if(firstName.equals("Rainer") && lastName.equals("Oechsle"))
    {
        return "student-response-special";
    }
    else
    {
        return "student-response";
    }
}

```

Diese Methode muss einen String zurückliefern, der dann so weiterverarbeitet wird, als hätte er direkt als Action-String in der XHTML-Datei gestanden. Bei der oben beispielhaft gezeigten Methode gehen wir davon aus, dass außer `student-response.xhtml` zusätzlich

auch eine Datei namens student-response-special.xhtml in unserem JSF-Web-Projekt existiert.

Die Klasse Options ist zusätzlich noch mit @ApplicationScoped annotiert, während dies bei der Klasse Student nicht so ist. Um dies besser zu verstehen, schauen wir uns Managed Beans und ihre Scopes etwas näher an.

## 7.8.2 Managed Beans und deren Scopes

Unter Scope (Gültigkeitsbereich) wird in der Informatik im Allgemeinen die Lebenszeit und Sichtbarkeit von Elementen verstanden. Für die Managed Beans gibt es die folgenden Scopes, die durch Angabe der entsprechenden Annotation eingestellt werden kann:

- **@ApplicationScoped:** Damit kennzeichnet man anwendungsglobale Daten (s. Unterabschnitt 7.3.3). Das heißt: Es gibt ein einziges Objekt einer solchen ManagedBean-Klasse, das für alle Elemente der Anwendung sichtbar ist und so lange existiert, bis die Anwendung vom Web-Server entfernt wird (Undeployment) oder der Web-Server heruntergefahren wird.
- **@SessionScoped:** Damit kennzeichnet man ManagedBean-Klassen, wenn deren Objekte als Session-Daten benutzt werden sollen. Derartige Objekte werden an eine Session gebunden. Das heißt: Pro Session gibt es ein Objekt dieser Klasse, das so lange lebt wie die Session. Zur Erinnerung sei erwähnt, dass bei der Nutzung mehrerer Tabs in einem Browser dieselbe Session und damit dasselbe ManagedBean-Objekt benutzt wird.
- **@ViewScoped:** Hier gibt es ein Objekt pro Bildschirmseite. Das heißt: Wenn man mit mehreren Tabs im Browser arbeitet, dann wird dies auf Server-Seite unterschieden und bei der Bearbeitung der Requests der unterschiedlichen Tabs wird jeweils mit einem eigenen ManagedBean-Objekt gearbeitet.
- **@RequestScoped:** In diesem Fall gibt es ein Objekt pro Request. Dies ist der Scope, der verwendet wird, falls kein Scope explizit angegeben wird.
- **@NoneScoped:** Dieser Scope wird nur benötigt, falls Managed Beans zur Initialisierung anderer Managed Beans benutzt werden. Darauf gehen wir nicht ein.

Im obigen Beispiel war es nun so, dass für die Klasse Student kein Scope angegeben war. Deshalb wurde hier standardmäßig @RequestScoped gesetzt, was für diese Anwendung ausreichend war. Es genügt, wenn das ManagedBean-Objekt und damit dessen Daten nur bis zum Ende der Bearbeitung des laufenden Requests vorhanden sind. Eine längerfristige Speicherung etwa über @SessionScoped war für dieses Beispiel nicht nötig. Wenn man @ApplicationScoped für Student gesetzt hätte, wäre dies sogar kontraproduktiv gewesen, weil sich bei gleichzeitiger Nutzung der Anwendung durch mehrere Kunden die Ein- und Ausgabedaten hätten vermischen können. Die Anwendung hätte aber funktioniert, wenn umgekehrt die ManagedBean-Klasse Options wie Student keine Scope-Angabe gehabt hätte und damit @RequestScoped gewesen wäre. Da aber ein Options-Objekt unveränderlich ist, wäre es unnötig aufwändig gewesen, bei jedem Request ein neues Options-Objekt zu erzeugen. Es reicht hier, wenn ein solches Objekt ein einziges Mal erzeugt und dann immer wieder verwendet wird.

Bei der Annotation @ManagedBean kann optional noch ein Name angegeben werden, der in den XHTML-Dateien benutzt wird. Wird dieser Name wie im obigen Beispiel nicht angegeben, so ist der Name, mit dem man sich auf die Objekte in den XHTML-Dateien bezieht, der Klassename, wobei allerdings der erste Buchstabe des Klassennamens, der nach der in Java üblichen Konvention ein Großbuchstabe sein sollte, zu einem Kleinbuchstaben wird. Die Klasse kann sich in einem beliebigen Package befinden; der Package-Name spielt keine Rolle. Ein weitere mögliche Angabe in der Annotation @ManagedBean ist eager (fleißig), dessen Wert true oder false sein kann. Wie gleich verständlich sein wird, macht diese Angabe nur Sinn für ManagedBeans, die ApplicationScoped sind. Ein True-Wert bedeutet nämlich, dass das Objekt schon beim Starten der Anwendung (Deployment) bzw. des Servers erzeugt wird. Bei false, was der Standard ist und gilt, wenn die Angabe weggelassen wird, wird das Objekt erst erzeugt, wenn es benötigt wird. Für die anderen Scopes macht diese Angabe keinen Sinn, da zum Beispiel SessionScoped-Objekte immer erst mit dem Entstehen einer neuen Session erzeugt werden können und nie im Voraus. Entsprechendes gilt für View- und RequestScoped. Die explizite Angabe von name und eager für die Klasse Options könnte dann beispielsweise so aussehen:

```
@ManagedBean(name="choices", eager=true)
@ApplicationScoped
public class Options ...
```

Wie auch bei den Servlets muss der Zugriff auf anwendungsglobale Daten und auf Session-Daten synchronisiert werden, falls auch schreibende Zugriffe vorkommen. Für ViewScoped und RequestScoped ist keine Synchronisation nötig. Dies soll anhand des mittlerweile wohl vertrauten Zählerbeispiels verdeutlicht werden.

Als ManagedBeans verwenden wir in unserem Beispiel die vier Counter-Klassen ApplicationCounter, SessionCounter, ViewCounter und RequestCounter, die alle einen Zähler mit den parameterlosen Methoden increment und reset (Rückgabetyp void für beide) sowie getCounter (Rückgabetyp int) darstellen, allerdings entsprechend ihres Namens unterschiedlich annotiert sind bezüglich ihrer Scopes. Zumindest beim ApplicationCounter und SessionCounter müssen alle Methoden synchronisiert sein. Die XHTML-Datei enthält pro ManagedBean-Zähler je ein Formular (s. Bild 7.8).

The screenshot shows a web application titled "Zähler mit unterschiedlichen Geschmacksrichtungen". It contains four sections, each with a title, a counter value, and two buttons: "Erhöhen!" and "Zurücksetzen!".

- Application Scope:** Aktueller Zählerstand: 0. Buttons: Erhöhen!, Zurücksetzen!.
- Session Scope:** Aktueller Zählerstand: 0. Buttons: Erhöhen!, Zurücksetzen!.
- View Scope:** Aktueller Zählerstand: 0. Buttons: Erhöhen!, Zurücksetzen!.
- Request Scope:** Aktueller Zählerstand: 0. Buttons: Erhöhen!, Zurücksetzen!.

**Bild 7.8** Seite mit Formularen für Managed Beans mit unterschiedlichen Scopes

Da die vier Formulare für die ManagedBeans alle ähnlich sind, ist im Folgenden nur das erste Formular für den Zähler mit ApplicationScope dargestellt. Die anderen Formulare sind analog:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html xmlns="http://www.w3.org/1999/xhtml"
xmlns:f="http://java.sun.com/jsf/core"
xmlns:h="http://java.sun.com/jsf/html">

<h:head>
    <title>Zähler mit unterschiedlichen
    Geschmacksrichtungen</title>
</h:head>
<h:body>
    <h1>Zähler mit unterschiedlichen Geschmacksrichtungen</h1>
    <h2>Application Scope</h2>
    <h:form>
        <p>Aktueller Zählerstand:
            <h:outputText value="#{applicationCounter.counter}" />
        </p>
        <p>
            <h:commandButton value="Erhöhen!" />
            <h:commandButton value="Zurücksetzen!" />
        </p>
    </h:form>
</h:body>
</html>
```

```

        actionListener="#{applicationCounter.increment}"/>
        <h:commandButton value="Zurücksetzen!" 
        actionListener="#{applicationCounter.reset}"/>
    </p>
</h:form>
<h2>Session Scope</h2> ...
<h2>View Scope</h2> ...
<h2>Request Scope</h2> ...
</h:body>
</html>
```

In der XHTML-Datei wird das Tag h:outputText verwendet, das selbsterklärend sein sollte. Als Wert wird der EL-Ausdruck #{applicationCounter.counter} angegeben. Hierzu wird ein ManagedBean mit passendem Namen gesucht bzw. erzeugt. Auf dieses Objekt wird dann die Methode getCounter angewendet. In den CommandButtons ist dieses Mal nicht action angegeben wie zuvor, sondern actionListener. Die als EL-Ausdruck angegebene Methode sollte anders als bei action keinen String zurückliefern für die Antwortseite, sondern void sein. Aus diesem Grund sind in den ManagedBean-Klassen increment und reset void, was die zusätzliche Methode getCounter notwendig macht. Als Antwortseite wird die Seite, aus der der Methodenaufruf getriggert wurde, noch einmal neu generiert und an den Browser zurückgegeben.

Wenn man diese Anwendung auf unterschiedlichen Rechnern, mit mehreren Browsern auf einem Rechner und mit mehreren Tabs pro Browser ausprobiert, dann wird die Bedeutung der Scopes erlebbar:

- Den ApplicationScoped-Zähler gibt es ein einziges Mal. Wenn man diesen von irgendwo aus erhöht hat, dann spiegelt sich dies beim nächsten Erhöhen wider, gleichgültig, auf welchem Rechner, in welchem Browser und Tab man die nächste Erhöhung durchführt.
- Beim SessionScoped-Zähler erkennt man, dass man auf unterschiedlichen Rechnern oder bei der Nutzung unterschiedlicher Browser-Typen auf einem Rechner unabhängig voneinander einen Zähler hochzählen kann, dass man sich aber bei der Nutzung mehrerer Tabs oder mehrerer Fenster beispielsweise des Firefox-Browsers einen Zähler teilt wie beim ApplicationScoped-Zähler.
- Beim ViewScoped-Zähler kann man wirklich in jedem Browser und Tab unabhängig von den anderen Browsern und Tabs hochzählen. Wenn man also schon auf 10 gezählt hat und man wechselt den Tab, dann beginnt die Zählung dort wieder bei 1 und wird nicht bei 11 fortgesetzt wie beim ApplicationScoped- und SessionScoped-Zähler.
- Der RequestScoped-Zähler ist nur aus Demonstrationsgründen vorhanden. Er ist eigentlich sinnlos, denn nach jedem Drücken des Erhöhen-Buttons wird 1 angezeigt. Dies ist nicht verwunderlich, da bei jedem Request ein neues Zähler-Objekt mit dem Anfangswert 0 erzeugt wird, dessen Wert anschließend erhöht wird. Somit ist der angezeigte neue Zählerstand immer 1. Das Zurücksetzen funktioniert wie erwartet.

Da bei dieser Anwendung die Änderung des Zählers und die Abfrage des aktuellen Werts durch zwei unterschiedliche Methodenaufrufe realisiert wird, ist bei Nutzung des ApplicationScoped- und SessionScoped-Zählers nicht garantiert, dass man auf der im Folgenden angezeigten Web-Seite den Wert sieht, auf den man selber den Zähler geändert hat. Das heißt, dass beispielsweise nach dem Zurücksetzen nicht in jedem Fall 0 angezeigt werden muss. Zwischen Änderung und Abfrage könnten nämlich eine oder sogar mehrere wei-

tere Änderungen erfolgen, denn selbstverständlich läuft auch bei JSF die Verarbeitung der Requests parallel ab, was man bei Bedarf jederzeit selber ausprobieren kann, indem man die Laufzeit von Methoden der ManagedBeans wieder künstlich mit Sleep verlängert.

### 7.8.3 MVP-Prinzip mit JSF

Wie sich bei einer Recherche im Internet zeigt, gibt es sehr viele unterschiedliche Auffassungen darüber, wie die einzelnen Bestandteile einer JSF-Anwendung nach dem MVP-Architekturmuster gestaltet werden sollen bzw. welche Bestandteile einer JSF-Anwendung den M-, V- und P-Komponenten entsprechen. Es herrscht noch nicht einmal Einigkeit darüber, ob JSF überhaupt zu MVP passt oder eher zu einem der alternativen Architekturmuster MVC oder MVVM, die zu Beginn des Abschnitts 4.4 kurz erwähnt, aber nicht näher erläutert wurden. Eine abschließende Klärung dieser Frage scheint nicht möglich, aber auch nicht wichtig zu sein. Wichtig ist lediglich, dass man seine Anwendung klar nach einem verständlichen und nachvollziehbaren Prinzip strukturiert. Dazu werden im Folgenden einige Ideen vorgestellt, wobei wir uns am MVP-Muster orientieren, da dieses Muster in Kapitel 4 im Kontext von JavaFX besprochen wurde.

Durch die deutliche Trennung von HTML und Java-Code hat man mit JSF bereits einen Schritt in Richtung MVP gemacht. Der XHTML-Teil stellt ganz offensichtlich die View-Komponente dar. Manche sind der Meinung, dass das FacesServlet, das oben erwähnt wurde und ein zentraler Baustein zur Realisierung des JSF-Frameworks darstellt, die Presenter-Komponente ist. Dieser Auffassung schließen wir uns in diesem Buch nicht an, sondern wir legen den Fokus darauf, den eigenen Programmcode sinnvoll in einen Teil zu trennen, der für die Ablaufsteuerung verantwortlich ist (P-Komponente), und in einen Teil, der mit den relevanten Daten hantiert und die Geschäftslogik realisiert (M-Komponente).

Managed Beans mit Request Scope sollten gemäß dieser Auffassung eher zur P-Komponente gezählt werden, denn die von einem Formular übernommenen Daten werden nur zur Bearbeitung eines einzigen Requests in das Objekt übernommen, um im folgenden Ablauf genutzt zu werden. Auch die Übernahme der Daten in die im Folgenden produzierte Webseite bedeutet die Anzeige temporärer Daten. In unserem ersten JSF-Beispiel (Formular für Studierende) sollte man die Klasse Student somit eher als Presenter interpretieren, insbesondere die Variante, in der die Methode handleRequest ergänzt wurde. In handleRequest wurde nämlich anhand der aktuellen Eingabedaten, die zuvor bereits in die Attribute übernommen worden sind, der weitere Ablauf gesteuert, indem entschieden wird, welche Antwortseite angezeigt werden soll. Eine Modellkomponente gab es in diesem einfachen Beispiel nicht.

Managed Beans mit View, Session oder Application Scope kann man als Teil der M-Komponente auffassen. In vielen Anwendungen werden aber Daten nicht nur in Objekten, deren Zustand beim Herunterfahren des Web-Servers verloren ist, sondern persistent im Dateisystem oder in Datenbanken gespeichert, wobei der Zugriff auf eine relationale Datenbank häufig über *Objekt-Relationale-Mapper (ORM)* wie etwa Hibernate erfolgt. In manchen Fällen ist zwischen die Web-Anwendung und die Datenbank auch noch ein Anwendungs-Server mit *Enterprise Java Beans (EJB)* geschaltet. Man könnte diesen ganzen Bereich als Modellkomponente auffassen. Manchmal wird hierfür auch der Begriff Backend verwendet, wäh-

rend die Web-Seite das Frontend darstellt. In unserem Zählerbeispiel könnte man so die Zählerklassen (mit Ausnahme des RequestScoped-Zählers, der ohnehin sinnlos war) als Modellkomponente sehen, die die relevanten Daten über einen Request hinaus speichern und die Logik zur Änderung dieser Daten enthalten. Hier gab es dafür im Gegensatz über das FacesServlet hinaus keine Ablaufsteuerung in unserem Programmcode und damit auch keine P-Komponente. Dass in der XHTML-Datei, welche die View darstellt, die Daten direkt aus dem Modell gelesen werden, also ein direktes Zusammenwirken von V und M existiert, was bei MVP im Prinzip verboten ist, sollte bei diesem einfachen Beispiel nicht groß stören.

Es ist im Java-Code von JSF möglich, sich die Referenzen von Managed Beans zu beschaffen und auf diese dann zuzugreifen. Hierfür gibt es mehrere Möglichkeiten. Im Folgenden wird beispielhaft gezeigt, wie man sich in der Zähler-Anwendung die Referenz auf den ApplicationScoped-Zähler in einer der anderen ManagedBean-Klassen beschaffen könnte:

```
FacesContext c = FacesContext.getCurrentInstance();
Application a = context.getApplication();
ApplicationCounter ac = a.evaluateExpressionGet(context,
                                                "#{applicationCounter}",
                                                ApplicationCounter.class);
```

Man könnte so zum Beispiel in der Ablaufsteuerung, also von einem Managed Bean, das RequestScoped ist, auf ein anderes Managed Bean, das ApplicationScoped ist und zur Modellkomponente gehört, zugreifen.

#### 7.8.4 AJAX mit JSF

Im JSF-Zähler-Beispiel wurde nach jeder Aktion auf dem Web-Server die vollständige Web-Seite erneut vom Web-Server auf den Browser übertragen. Da diese Web-Seite sehr klein ist, müssen dafür nicht viele Daten übertragen werden. Auch hat der Browser mit der Interpretation der Daten und der Aktualisierung seiner Anzeige nicht allzu viel Arbeit. Im Allgemeinen, bei sehr umfangreichen Web-Seiten mit eventuell vielen zusätzlichen Inhalten wie Bildern, ist dies aber ein unnötiger Aufwand, den man sich gern ersparen möchte. In unserem vorigen Beispiel ändert sich auf der Web-Seite durch jeden Klick auf einen Command Button nur einer der Zählerstände. Der Rest der Seite bleibt gleich. Schön wäre es, wenn man nur die veränderten Inhalte vom Web-Server zum Browser übertragen könnte und wenn der Browser dann in der Lage wäre, die vorhandene Web-Seite an entsprechender Stelle zu ändern. Mit AJAX ist genau dies möglich.

AJAX steht für *Asynchronous JavaScript And XML*. Zur Realisierung von AJAX nutzt man in der Regel JavaScript. Wie Bilder können JavaScript-Programme in Web-Seiten, die vom Web-Server geladen werden, eingebettet sein. Diese Programme werden vom Browser ausgeführt. An die Buttons von Formularen lassen sich zum Beispiel JavaScript-Funktionen als Listener anhängen. Wenn dann der Button geklickt wird, wird die JavaScript-Funktion auf dem Browser ausgeführt, die beispielsweise eine spezielle POST-Anfrage an den Web-Server senden könnte, um nur den Zähler hochzuzählen und den aktuellen Zählerstand in Form eines verhältnismäßig kleinen XML-Dokuments anzufordern. Alle JavaScript-Funktionen werden von einem einzigen Thread des Browsers ausgeführt, der nicht blockiert werden sollte, da sonst während dieser Blockade keine anderen Aktionen auf dem Browser

ausgeführt und damit z. B. nicht auf weitere Ereignisse reagiert werden kann (vgl. JavaFX). Die JavaScript-Funktion wartet somit besser nicht auf eine Antwort vom Server. Der Aufruf wird also asynchron durchgeführt, womit nun schließlich auch noch ein Bezug zum ersten Buchstaben im Akronym AJAX hergestellt ist. Statt des Wartens auf die Antwort wird eine andere JavaScript-Funktion angegeben, die wieder vom JavaScript-Interpreter-Thread des Browsers ausgeführt wird, nachdem die Antwort vom Server eingetroffen ist. Diese zweite JavaScript-Funktion kann dann die aktuelle Web-Seite verändern. Das gesamte HTML-Dokument hat eine baumartige Struktur. Man bezeichnet diesen Baum als *DOM (Document Object Model)*. Mit JavaScript ist es einfach möglich, Teile des DOM-Baums in beliebiger Weise zu verändern, worauf sich die Anzeige im Browser entsprechend aktualisiert.

Auch wenn der Name JavaScript eine große Ähnlichkeit mit Java suggeriert, so gibt es trotz einiger Ähnlichkeiten auch eine ganze Reihe starker Unterschiede. Aus didaktischer Sicht wäre die Nutzung einer zusätzlichen Programmiersprache auf den letzten Seiten dieses Buches nicht besonders günstig. Zum Glück gibt es aber eine einfache Möglichkeit, AJAX in JSF zu benutzen. Für die Zähler-Anwendung müssen wir dazu lediglich die XHTML-Datei ein klein wenig anpassen. Im Folgenden werden die Änderungen (fett dargestellt) stellvertretend nur für das Formular des ApplicationScoped-Zählers gezeigt. Die Formulare für die anderen Zähler können entsprechend angepasst werden:

```
<h:form id="appform">
    <p>Aktueller Z&auml;hlerstand:
        <h:outputText id="counter" value="#{applicationCounter.counter}" />
    </p>
    <p>
        <h:commandButton value="Erh&ouml;hen!" 
            actionListener="#{applicationCounter.increment}">
            <f:ajax render="appform:counter"/>
        </h:commandButton>
        <h:commandButton value="Zur&uuml;cksetzen!" 
            actionListener="#{applicationCounter.reset}">
            <f:ajax render="appform:counter"/>
        </h:commandButton>
    </p>
</h:form>
```

Sowohl das Formular als auch die Zähleranzeige im Formular bekommt jeweils eine Kennung. Da der Zähleranzeige die Kennung counter im Formular mit der Kennung appform zugewiesen wird, lautet die vollständige Kennung des Zählerausgabefelds appform:counter. Die CommandButtons wurden bisher über `<h:commandButton .../>` mit einem kombinierten öffnenden und schließenden Tag definiert. Jetzt gibt es separat das öffnende Tag `<h:commandButton ...>` und das schließende Tag `</h:commandButton>`, um dazwischen das neue Tag `<f:ajax .../>` einzufügen zu können. Dadurch wird festgelegt, dass beim Klicken des jeweiligen Buttons ein (asynchroner) AJAX-Aufruf stattfinden soll. Durch die Angabe in render wird der Teil des DOM-Baums festgelegt, der nach dem Eintreffen der Antwort des AJAX-Aufrufs aktualisiert werden soll. In diesem Fall ist das natürlich die Zähleranzeige. Die Nutzung von AJAX wird dadurch extrem einfach, denn weitere Änderungen sind nicht nötig. Wer Interesse daran hat, was aus dem Tag `<f:ajax ...>` wird, braucht sich einfach nur einmal den Quelltext der im Browser angezeigten HTML-Seite anzusehen.

Für den Benutzer ist der Unterschied ohne und mit AJAX kaum erkennbar. Deutlicher wird es allerdings, wenn man sich die zwischen Browser und Web-Server übertragenen Daten ansieht. Browser wie Firefox bieten die Möglichkeit, den Netzverkehr anzuzeigen. Alternativ kann man mit einem Programm wie Wireshark auch den Datenverkehr belauschen oder einen Proxy zwischen Browser und Web-Server setzen, der den durchgeleiteten Datenverkehr ausgibt. Wie man auch immer an die Daten gelangt: Wenn AJAX nicht verwendet wird, kann man sehen, dass der Browser beim Klicken des Erhöhen-Buttons folgenden HTTP-Request absendet:

```
POST /puva/faces/counter/c.xhtml HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) ... Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/puva/faces/counter/c.xhtml
Content-Type: application/x-www-form-urlencoded
Content-Length: 1271
Cookie: JSESSIONID=07C3DDCBAB9E8F9FFF871B0FBE974821
DNT: 1
Connection: keep-alive
Upgrade-Insecure-Requests: 1

j_idt6=j_idt6&j_idt6%3Aj_idt10=Erh%C3%B6hen%21&javax.faces.ViewState=...
```

Es handelt sich hierbei um einen normalen POST-Request. Wir haben zuvor gesehen, dass bei der Wandlung von XHTML zu HTML zwei versteckte Felder in das Formular eingebaut werden, zum einen ein Feld zur Kennzeichnung des Formulars (`j_idt6=j_idt6`) und zum anderen eine lange, schwer zu ratende Kennung unter dem Namen `javax.faces.ViewState`. Außerdem wird jedem Button ein Name zugewiesen (in diesem Fall `j_idt6:j_idt10`). Somit wird beim Klicken des Buttons sein Name und Wert, der in diesem Fall die Beschriftung des Buttons ist („Erhöhen!“), übertragen. Da Umlaute und Sonderzeichen wie Doppelpunkte und Ausrufezeichen durch bestimmte Zeichenfolgen codiert werden, sind die drei Name-Wert-Paare nicht leicht auf den ersten Blick zu erkennen.

Als Antwort kommt die vollständige HTML-Seite wieder zurück, wie zu erwarten war:

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: text/html; charset=UTF-8
Content-Length: 7130
Date: Sat, 16 Sep 2017 09:19:17 GMT

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
    <head id="j_idt2">
        <title>Zähler mit unterschiedlichen ...</title>
    </head>
    <body>
        <h1>Zähler mit unterschiedlichen Geschmacksrichtungen</h1>
        <h2>Application Scope</h2>
        <form id="j_idt6" name="j_idt6" method="post"
            action="/puva/faces/counter/c.xhtml"
```

```

        enctype="application/x-www-form-urlencoded">
        <input type="hidden" name="j_idt6" value="j_idt6" />
        <p>Aktueller Zählerstand: 10
        </p>
        <p>
            <input type="submit" name="j_idt6:j_idt10"
                   value="Erhöhen!" />
            <input type="submit" name="j_idt6:j_idt11"
                   value="Zurücksetzen!" />
        </p>
        <input type="hidden" name="javax.faces.ViewState"
               id="j_id1:javax.faces.ViewState:0"
               value="92...q2M=" autocomplete="off" />
    </form>
    ...
</body>
</html>
```

Im AJAX-Fall sieht der vom Browser abgesendete Request dagegen so aus:

```

POST /puva/faces/counter/c.xhtml HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) ... Firefox/55.0
Accept: /*
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Referer: http://localhost:8080/puva/faces/counter/c-ajax.xhtml
Faces-Request: partial/ajax
Content-type: application/x-www-form-urlencoded; charset=UTF-8
Content-Length: 1645
Cookie: JSESSIONID=FF55D61A70B77390C164DF02A258A7C6
DNT: 1
Connection: keep-alive

appform=appform&
javax.faces.ViewState=...&
javax.faces.source=appform%3Aj_idt8&
javax.faces.partial.event=click&
javax.faces.partial.execute=appform%3Aj_idt8%20appform%3Aj_idt8&
javax.faces.partial.render=appform%3Acounter&
javax.faces.behavior.event=action&
javax.faces.partial.ajax=true
```

Auch hier sehen wir wieder einen normalen POST-Request mit der URL der XHTML-Seite, in der das komplette Formular steht. Die übertragenen Daten sehen allerdings ganz anders aus. Der besseren Lesbarkeit halber wurden die übertragenen Daten so editiert, dass die einzelnen Name-Wert-Paare, die durch & getrennt sind, in unterschiedlichen Zeilen stehen. Wir erkennen, dass die in der XHTML-Datei angegebene Information appform:counter als Render-Wert (wieder in Ersatzdarstellung für den Doppelpunkt) an den Server übertragen wird. Was die restlichen Daten bedeuten, soll hier nicht interessieren. Wichtiger ist zu erkennen, dass der Web-Server in diesem Fall anders als zuvor antwortet:

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Cache-Control: no-cache
```

```
Content-Type: text/xml; charset=UTF-8
Content-Length: 1535
Date: Sat, 16 Sep 2017 10:52:50 GMT

<?xml version='1.0' encoding='UTF-8'?>
<partial-response id="j_id1">
    <changes>
        <update id="appform:counter">
            <! [CDATA[<span id="appform:counter">4</span>]]>
        </update>
        <update id="j_id1:javax.faces.ViewState:0">
            <! [CDATA[qzfMb...oz]]>
        </update>
    </changes>
</partial-response>
```

Es kommt also in diesem Fall nicht die komplette Web-Seite vom Server zurück, sondern im Wesentlichen nur die Information zu appform:counter. Diese Information ist in diesem Fall „4“. Mit diesem Wert wird dann durch eine bereitgestellte JavaScript-Funktion der DOM-Baum verändert, damit die Zahl 4 auf der Web-Seite erscheint.

Dies war ein äußerst knapper Einstieg in das Thema JSF, zu dem es sehr viel mehr zu sagen gäbe. Wir wenden uns nun aber einem anderen Thema zu.

## ■ 7.9 RESTful WebServices

In diesem Kapitel ging es bisher fast ausschließlich um die Produktion von HTML-Web-Seiten, die direkt vom Browser interpretiert und angezeigt werden können. AJAX war eine Ausnahme, denn die übertragenen XML-Daten wurden auf Client-Seite von einer JavaScript-Methode im Browser verarbeitet und nicht vom Standard-Code des Browsers. In diesem Abschnitt geht es jetzt um die Nutzung des HTTP-Protokolls zur Kommunikation zwischen einem Server und irgendeinem Client-Programm. Das Client-Programm kann als JavaScript-Funktion in einem Browser laufen. Es kann aber auch eine eigenständige Anwendung sein, in der zur Kommunikation mit dem Server statt Sockets oder RMI das HTTP-Protokoll benutzt wird. In diesem Buch wird unser Client-Programm natürlich auch in Java programmiert sein. Die über HTTP ausgetauschten Daten sind in diesem Fall typischerweise keine HTML-Daten, denn sie sind nicht für die Anzeige in einem Browser gedacht. Man spricht in diesem Fall von sogenannten *WebServices*.

Früher waren WebServices nahezu gleichbedeutend mit *SOAP (Simple Object Access Protocol)*. SOAP ist eine Form eines WebService, der (obwohl nicht zwingend) in der Regel auf HTTP basiert. Der Body-Teil besteht aus XML-Daten in einem speziellen XML-Dialekt. Als XML aufkam, gab es eine große Euphorie und alles, auch jede Konfigurationsdatei, musste im XML-Format sein. Inzwischen ist diese Modewelle abgeebbt. Was zuvor als die allein seligmachende Lösung zur Kommunikation im Internet gepriesen wurde, wird mittlerweile als zu kompliziert und zu aufwändig gesehen. *RESTful WebServices* (*REST: Representational State Transfer*) sind eine Art Gegenbewegung zu SOAP. Auch hier geht es um die Kommunikation zwischen einem Client-Programm und einem Server unter Verwendung von HTTP.

Die damit ausgetauschten Daten sollten unterschiedliches Format haben können, allerdings wesentlich schlanker sein. Seit einiger Zeit erfreut sich *JSON (JavaScript Object Notation, s.unten)* größerer Beliebtheit. Dieses Format wird sehr häufig im Zusammenhang mit RESTful WebServices verwendet. Auch wenn das Akronym JSON darauf hinweist, dass das Format aus der JavaScript-Welt stammt, so ist JSON ein Format, das mittlerweile in sehr vielen Programmiersprachen unterstützt und somit häufig auch in Java-Anwendungen eingesetzt wird. Bevor wir uns ein Beispiel für einen solchen WebService in Java ansehen, schauen wir uns etwas genauer an, was man unter RESTful WebServices versteht und wie das JSON-Format aussieht.

### 7.9.1 Definition von RESTful WebServices

Die Grundidee von RESTful WebServices besteht darin, im Rahmen einer Client-Server-Kommunikation ein Protokoll zum Zugriff auf Ressourcen anzubieten. Die ausgetauschten Daten werden in der Regel nicht wie HTML-Text in einem Browser für menschliche Benutzer angezeigt, sondern von Programmen verarbeitet. Man spricht deshalb auch von Maschine-zu-Maschine-Kommunikation, obwohl der Begriff streng genommen nichts aussagt, denn auch bei der Kommunikation zwischen Browser und Web-Server geht es um eine Kommunikation zwischen zwei Computern. Wichtige Merkmale der RESTful WebServices sind:

- Man geht von einer Menge von Ressourcen aus. Jede Ressource besitzt eine eindeutige URL, unter der diese erreichbar ist.
- Zum Zugriff auf die Ressourcen wird in der Regel das HTTP-Protokoll verwendet. Neben den Kommandos GET und POST kommen auch weitere in diesem Protokoll definierten Kommandos zum Einsatz, insbesondere PUT und DELETE. Wie der Name andeutet, soll GET eine rein lesende Operation sein, die den Zustand der Ressource nicht verändert. Ebenfalls am Namen ablesbar ist die Wirkung von DELETE, bei der eine Ressource gelöscht wird. POST und PUT sind beides ändernde Operationen. PUT soll dabei idempotent sein. Das heißt, dass die mehrfache Ausführung des PUT-Kommandos dieselbe Wirkung hat wie die einfache Ausführung. Hierzu gehört das Setzen der Eigenschaften einer Ressource auf bestimmte Werte. Als einfaches Beispiel kann man sich das Setzen eines Zählers auf einen bestimmten Wert vorstellen. Unsere Methode reset des häufig benutzten Zählerbeispiels ist idempotent, denn wenn der Zähler einmal auf 0 gesetzt wird, hat das denselben Effekt, wie wenn man ihn mehrfach auf 0 setzt. POST kann für andere Änderungen verwendet werden, zum Beispiel auch für das Erzeugen neuer Ressourcen, was im Allgemeinen nicht idempotent ist. Beim Zählerbeispiel wäre increment eine nicht-idempotente Aktion, denn das mehrfache Erhöhen des Zählers ist etwas anderes als das einfache Erhöhen.
- Die im Body-Teil übertragenen Daten sind auf kein bestimmtes Format festgelegt. Häufig wird jedoch JSON verwendet. Bei einer GET-Anfrage kann angegeben werden, welches Format man sich als Ergebnis wünscht. Im Allgemeinen wird bei HTTP immer angegeben, welches Format die im Body-Teil enthaltenen Daten haben (z.B. in POST-Kommandos oder in Antworten auf GET-Anfragen).

- Das Prinzip der miteinander vernetzten Ressourcen, die es im Web bezüglich der Web-Seiten gibt, kann auch hier weiterhin genutzt werden, denn in den Nachrichten können URLs anderer Ressourcen angegeben werden, z.B. als Antwort auf eine PUT-Anfrage, bei der eine neue Ressource erzeugt wird. Die URL dieser neuen Ressource kann als Antwort zurückgeliefert und anschließend in den folgenden Requests verwendet werden.
- Die Kommunikation ist Zustandslos. Dies heißt, dass keine z.B. über Cookies realisierte Sitzungen existieren, sondern dass bei jeder Anfrage alle notwendigen Informationen wieder neu mitgeschickt werden müssen.

## 7.9.2 JSON

JSON ist ein sehr einfaches Format, mit dem Objektstrukturen beschrieben werden können. Der Zustand eines Objekts besteht bekanntlich aus den aktuellen Werten seiner Attribute. Mit JSON wird der Zustand eines Objekts als eine Folge von Name-Wert-Paaren in geschweiften Klammern notiert. Die Paare sind jeweils durch ein Komma voneinander getrennt, Name und Wert durch einen Doppelpunkt. Der Name des Attributs wird als Zeichenkette in Anführungszeichen angegeben. Werte können sein: Strings (ebenfalls Zeichenketten in Anführungszeichen), Zahlen, boolesche Werte (true, false), der Wert null, eine wiederum in geschweiften Klammern geschachtelte Beschreibung eines weiteren Objekts sowie Felder (in eckigen Klammern gefasste Folge von Werten oder Objekten, jeweils durch Komma voneinander getrennt). Die folgenden Zeilen enthalten ein Beispiel einer Objektbeschreibung im JSON-Format:

```
{
  "name": "Van Morrison",
  "born": 1945,
  "cds": [
    {
      "title": "Astral Weeks",
      "year": 1968
    },
    {
      "title": "Moondance",
      "year": 1970
    },
    {
      "title": "Keep Me Singing",
      "year": 2016
    }
  ]
}
```

Wie zu sehen ist, ist der Wert des Attributs cds ein Feld mit drei Elementen, wobei jedes Element ein Objekt mit den Attributen title und year ist. Es gibt Bibliotheken, mit denen man sehr einfach solche JSON-Strings aus einer vorhandenen Objektstruktur erzeugen kann. Man bezeichnet dies als Serialisierung (siehe auch Unterabschnitt 6.4.1), weil man eine Objektstruktur in eine serielle Folge von Zeichen transformiert. Aber auch das Umkehrte, die Deserialisierung, ist möglich: Aus einem JSON-String werden die entsprechenden Objekte erzeugt und entsprechend miteinander vernetzt, wie in Listing 7.21 zu sehen

ist. In diesem Programm wird zur Serialisierung und Deserialisierung die GSON-Bibliothek verwendet wird, die ursprünglich von Google stammt, wie am Package-Namen noch zu erkennen ist.

### Listing 7.21

```
import com.google.gson.Gson;

class Cd
{
    private String title;
    private int year;
    //Konstruktor, Getter- und Setter-Methoden hier nicht gezeigt: ...
}

class Artist
{
    private String name;
    private int born;
    private Cd[] cds;
    //Konstruktor, Getter- und Setter-Methoden hier nicht gezeigt: ...
}

public class JsonExample
{
    public static void main(String[] args)
    {
        Cd cd1 = new Cd("Astral Weeks", 1968);
        Cd cd2 = new Cd("Moondance", 1970);
        Cd cd3 = new Cd("Keep Me Singing", 2016);
        Artist van = new Artist("Van Morrison", 1945,
                               new Cd[]{cd1, cd2, cd3});
        Gson gson = new Gson();
        String jsonString = gson.toJson(van);
        System.out.println(jsonString);

        Artist van2 = gson.fromJson(jsonString, Artist.class);
        System.out.println(van2.getName() + "/" + van2.getBorn());
        for(Cd cd: van2.getCds())
        {
            System.out.println(" " + cd.getTitle() + "/" + cd.getYear());
        }
    }
}
```

Im ersten Teil der Main-Methode erfolgt eine Serialisierung. Es wird eine Objektstruktur aufgebaut, bestehend aus einem Artist-Objekt, in dem sich eine Referenz auf ein Array befindet, das drei Referenzen auf unterschiedliche Cd-Objekte enthält. Mit der Methode `toJson` wird der JSON-String erzeugt und zurückgegeben. Dieser wird anschließend ausgegeben. Wenn man den JSON-String im Folgenden nicht mehr brauchen würde, sondern den JSON-String nur ausgeben möchte, dann hätte man auch schreiben können:

```
gson.toJson(van, System.out);
```

Die Ausgabe ist eine unformatierte Folge von Zeichen ohne Leerzeichen und Zeilenumbrüche, die sich gut zum Versenden über das Netz eignet:

```
{ "name": "Van Morrison", "born": 1945, "cds": [{"title": "Astral Weeks", ...}
```

Falls man eine formatierte Variante eines JSON-Strings benötigt, der für Menschen angenehmer zu lesen ist, so ist dies mit Gson auch möglich. Die folgenden Zeilen erzeugen genau die anfangs angegebene Beispieldaten aus:

```
Gson gsonPretty = new GsonBuilder().setPrettyPrinting().create();
gsonPretty.toJson(van, System.out);
```

Im zweiten Teil der Main-Methode wird mit Hilfe der Methode `fromJson` aus einem JSON-String eine Objektstruktur aufgebaut. Die nachfolgende Ausgabe soll demonstrieren, dass wir über die Referenz `van2` eine komplette Kopie der im Programm aufgebauten Objektstruktur erhalten haben. Das zweite Argument `Artist.class` beim Aufruf der Methode `fromJson` ermöglicht, bei der Zuweisung des Rückgabewerts auf ein Casting zu verzichten.

### 7.9.3 Beispiel

JAX-RS ist eine Java-Programmierschnittstelle für RESTful WebServices und Jersey ist eine Referenzimplementierung dieser Schnittstelle, die im folgenden Beispiel verwendet wird. Ähnlich wie bei JSF wird auf dem Web-Server ein Web-Projekt angelegt. Darin wird für einen bestimmten Pfad (in unserem Beispiel ist es `/rest/*`) ein Servlet namens `ServletContainer` aus der Jersey-Bibliothek festgelegt. Die Programmierung der Server-Seite (also die Programmierung, wie auf eingehende WebService-Anfragen reagiert wird) erfolgt ähnlich wie bei JSF und den Managed Beans auch hier mit Java-Klassen, die speziell annotiert sind. Die wichtigsten Annotationen sind:

- `@Path`: Dies ist eine Annotation für eine Klasse oder eine Methode. Die Bedeutung ist vergleichbar mit `@WebServlet`. Hier wird der hintere Teil der URL angegeben, unter der ein Objekt der Klasse erreichbar ist (der vordere Teil ist die URL des oben erwähnten Servlets aus der Jersey-Bibliothek). Wenn die Annotation an der Klasse steht, gilt der Pfad für alle Methoden. Einzelne Methoden können durch eine zusätzliche `@Path`-Annotation den Pfad verlängern und darin auch einen Parameter codieren, der mit der Annotation `@PathParam` zusammenwirkt.
- `@GET, @POST, @PUT, @DELETE`: Dies sind Annotationen für Methoden, die bei Eintreffen der dazugehörigen HTTP-Anfrage aufgerufen werden.
- `@Produces, @Consumes`: Als Parameter werden ein oder mehrere Media-Types wie beispielsweise XML oder JSON angegeben. Dies sind Annotationen für Methoden, mit denen man festlegen kann, welches Datenformat eine bestimmte Anfrage mitbringen soll und von der Methode verarbeitet werden kann bzw. welches Datenformat eine Methode produziert und damit in der Antwort auf eine Anfrage zurückgeliefert wird.
- `@PathParam`: Dies ist eine Annotation für ein Argument einer Methode. Der Teil der URL, der durch `@Path` spezifiziert ist, wird der betreffenden Methode als Parameter übergeben. Das wird sicher nachher anhand eines Beispiels besser verständlich werden.

In Listing 7.22 findet sich eine Klasse namens `ArtistManager`, mit der man Künstler der Klasse `Artist` (s. Listing 7.21) anlegen und löschen kann. Außerdem kann man alle gespei-

cherten Künstler oder einzelne Künstler abfragen. Die Künstler werden in einer HashMap gehalten. Als Schlüssel (Key) wird eine String-Kennung verwendet. Diese Kennung ist ein Zahlen-String, wobei bei jedem Eintragen eines neuen Künstlers die Zahl hochgezählt wird, sodass die Kennungen fortlaufend vergeben werden. Da bei jedem Request ein neues Artist-Manager-Objekt angelegt wird, wurden der Einfachheit halber die Attribute alle statisch angelegt. Die Methoden sind dementsprechend statisch und synchronized, da auch hier wieder automatisch Parallelität im Spiel ist.

### Listing 7.22

```
import java.util.HashMap;
import javax.ws.rs.*;
import javax.ws.rs.core.MediaType;
import com.google.gson.Gson;

@Path("/artists")
public class ArtistManager
{
    private static HashMap<String, Artist> allArtists = new HashMap<>();
    private static int number = 0;
    private static Gson gson = new Gson();

    @GET
    @Produces(MediaType.APPLICATION_JSON )
    public static synchronized String getAll()
    {
        return gson.toJson(allArtists);
    }

    @GET
    @Produces(MediaType.APPLICATION_JSON)
    @Path("{id}")
    public static synchronized String getArtist(@PathParam("id") String id)
    {
        return gson.toJson(allArtists.get(id));
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public static synchronized String addArtist(String json)
    {
        Artist a = gson.fromJson(json, Artist.class);
        if(a != null)
        {
            number++;
            allArtists.put("" + number, a);
            return "" + number;
        }
        else
        {
            return null;
        }
    }

    @DELETE
    @Path("{id}")
}
```

```

@Produces(MediaType.APPLICATION_JSON)
public static synchronized String deleteArtist(@PathParam("id")
                                              String id)
{
    if(allArtists.remove(id) != null)
    {
        return id;
    }
    else
    {
        return "null";
    }
}
}

```

Es gibt zwei mit @GET annotierte Methoden. Die erste liefert die komplette HashMap als JSON-String zurück. Die Notation ist ähnlich wie bei einem Objekt, wobei die Schlüssel den Attributnamen und die Werte der schon gesehenen JSON-Darstellung eines Künstlers entsprechen:

```

{
    "1": {"name": "Van Morrison", "born": 1945, "cds": [...]},
    "2": {"name": "Mark Knopfler", "born": 1949, "cds": [...]}
}

```

Die zweite mit @GET annotierte Methode hat zusätzlich eine @Path-Annotation. Darin wird ein Bezug hergestellt zum Parameter dieser Methode, der mit @PathParam annotiert ist. Es kann somit z.B. folgende URL angefordert werden:

```
/puva/rest/artists/1
```

Dies hat den Effekt, dass die betreffende Methode mit „1“ als Parameter aufgerufen wird.

Die mit @POST annotierte Methode empfängt einen JSON-String, aus dem ein Artist-Objekt erzeugt wird. Das Objekt wird in die HashMap übernommen und der dafür erzeugte Schlüssel wird zurückgeliefert. Schließlich gibt es noch eine mit @DELETE gekennzeichnete Methode zum Löschen eines Künstlers. Hier wird auf dieselbe Art wie in der zweiten GET-Methode die Kennung des zu löschen Künstlers mit in der URL übertragen.

Wie schon erwähnt ist die Client-Seite bei WebServices im typischen Fall kein Browser. In Listing 7.23 ist ein Client-Programm zu sehen, das Methoden hat mit denselben Namen wie soeben in Listing 7.22 gesehen. Der Inhalt ist allerdings ein anderer, denn mit diesen Methoden wird nicht auf Requests reagiert, sondern es werden Anfragen abgesendet und die Antworten aufbereitet und zurückgegeben. Mit getAll werden alle Künstler in Form eines JSON-Strings angefordert. Dieser wird in eine HashMap transformiert und zurückgegeben. Mit getArtist erhält man einen JSON-String, der einen einzelnen Künstler repräsentiert. Dieser wird in ein Artist-Objekt gewandelt und zurückgegeben. Mit addArtist wird aus einem übergebenen Artist-Objekt ein JSON-String gemacht. Dieser wird in einem POST-Request übertragen. Diese Methode liefert die Kennung des auf dem Server angelegten Objekts zurück. Mit deleteArtist wird schließlich ein DELETE-Request zum Löschen eines Künstlers gesendet.

**Listing 7.23**

```
import java.net.URI;
import java.util.HashMap;
import javax.ws.rs.client.*;
import javax.ws.rs.core.*;
import com.google.gson.Gson;

public class ArtistClient
{
    private static WebTarget target =
        ClientBuilder.newClient().target(getBaseURI());
    private static Gson gson = new Gson();

    private static URI getBaseURI()
    {
        return UriBuilder.
            fromUri("http://localhost:8080/puva/rest/artists").build();
    }

    @SuppressWarnings("unchecked")
    private static HashMap<String, Artist> getAll()
    {
        String response = target.request().
            accept(MediaType.APPLICATION_JSON).
            get(String.class);
        System.out.println("getAll(): " + response);
        return gson.fromJson(response, HashMap.class);
    }

    private static Artist getArtist(String id)
    {
        String response = target.path(id).request().
            accept(MediaType.APPLICATION_JSON).
            get(String.class);
        System.out.println("getArtist(" + id + "): " + response);
        return gson.fromJson(response, Artist.class);
    }

    private static String addArtist(Artist a)
    {
        String input = gson.toJson(a);
        String response = target.request().
            post(Entity.entity(input, MediaType.APPLICATION_JSON)).
            readEntity(String.class);
        System.out.println("addArtist(...): " + response);
        return response;
    }

    private static Artist buildSample(int type)
    {
        return ...;
    }

    private static String deleteArtist(String id)
    {
        String result = target.path(id).request().delete(String.class);
        System.out.println("deleteArtist(" + id + "): " + result);
    }
}
```

```

        return result;
    }

    public static void main(String[] args)
    {
        getAll();

        Artist a1 = buildSample(0);
        String id1 = addArtist(a1);

        Artist a2 = buildSample(1);
        String id2 = addArtist(a2);

        getAll();
        getArtist(id1);
        getArtist(id2);

        deleteArtist(id2);
        getAll();
        deleteArtist(id1);
        getAll();
    }
}

```

Das Client-Programm sollte im Groben verständlich sein. Statt auf die Details des Client-Programms einzugehen, schauen wir uns lieber an, was auf der Leitung passiert. Bei getAll wird folgender Request vom Client abgesendet:

```

GET /puva/rest/artists HTTP/1.1
Accept: application/json
User-Agent: Jersey/2.22.1 (HttpURLConnection 1.8.0_65)
Host: localhost:8080
Connection: keep-alive

```

Die Antwort des Servers lautet, wobei die Chunked-Transfer-Codierung der besseren Lesbarkeit wegen herauseditiert wurde:

```

HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json
Date: Mon, 18 Sep 2017 16:19:51 GMT

{"1":{"name":"Van Morrison"}, {"2":{"name":"Mark Knopfler"} }

```

Bei Aufruf der Client-Methode getArtist passiert etwas ganz Ähnliches. Der Unterschied ist, dass die URL zusätzlich die Kennung des Künstlers enthält, wie oben schon gezeigt wurde, und dass ein JSON-String zurückkommt für genau einen Künstler.

Durch addArtist wird diese Anfrage versendet:

```

POST /puva/rest/artists HTTP/1.1
Content-Type: application/json
User-Agent: Jersey/2.22.1 (HttpURLConnection 1.8.0_65)
Host: localhost:8080
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive

```

```
Content-Length: 832
```

```
{"name": "Neil Young", "born": 1945, "cds": [...]}
```

Diese wird vom Server wie folgt beantwortet (wieder etwas vereinfacht durch Entfernung der Chunked-Transfer-Codierung):

```
HTTP/1.1 200 OK
Server: Apache-Coyote/1.1
Content-Type: application/json
Date: Mon, 18 Sep 2017 16:19:51 GMT
```

```
3
```

Das Löschen eines Künstlers wird durch folgende Nachricht in die Wege geleitet:

```
DELETE /Restful/rest/artists/3 HTTP/1.1
User-Agent: Jersey/2.22.1 (HttpURLConnection 1.8.0_65)
Host: localhost:8080
Accept: text/html, image/gif, image/jpeg, *; q=.2, */*; q=.2
Connection: keep-alive
```

Die Antwort ist wie bei POST.

Zum Abschluss dieser Erläuterungen zu den RESTful WebServices kann ich mir die Bemerkung nicht ganz verkneifen, dass es doch sehr viel einfacher gewesen wäre, dieselbe Anwendung mit RMI zu entwickeln.

## ■ 7.10 WebSockets

Bei den RESTful WebServices wurde HTTP verwendet. Da HTTP ein Request-Response-Protokoll ist, hat dies zur Folge, dass der Server nur etwas senden kann, wenn der Client zuvor einen Request abgesendet hat. In einer Chat-Anwendung, wie wir sie bei RMI gesehen haben, muss aber der Server eine Nachricht an einen Client senden, nachdem ein anderer Client einen neuen Chat-Beitrag dem Server bereitgestellt hat. Zur Lösung dieses Problems müssen die Clients ein Polling durchführen. Das heißt, sie müssen ständig einen Request an den Server senden, bei dem sie fragen, ob es neue Beiträge gibt. Um die Rate der Polling-Requests zu reduzieren, kann der Server die Antwort verzögern. Erst, wenn ein Beitrag vorliegt oder eine bestimmte Zeit vergangen ist, wird geantwortet. Man nennt dieses Prinzip Long Polling. Ideal ist auch dieses Vorgehen nicht. *WebSockets* stellen eine Lösung für diese Problemstellung dar.

WebSockets können wie AJAX und RESTful WebServices nur dann genutzt werden, wenn auf Client-Seite ein Programm ausgeführt wird. Wie im vorigen Abschnitt erläutert wurde, können dies z.B. JavaScript-Funktionen sein, die von einem Browser ausgeführt werden, oder es kann eine eigenständige Java-Anwendung sein. Wird JavaScript verwendet, dann kann der JavaScript-Code von Hand geschrieben sein. Alternativ könnte er auch aus einem Framework stammen wie die AJAX-Funktionen bei JSF. Auch es ist möglich, ein Java-Pro-

gramm, das die GWT-Bibliothek nutzt (GWT: Google Web Toolkit), in JavaScript-Code zu übersetzen. In JavaScript gibt es beispielsweise eine Funktion, mit der man eine neue HTTP-Verbindung zu einem Web-Server aufbauen kann. Wenn man WebSockets nutzen möchte, dann beginnt die anzugebende URL nicht wie üblich mit http (bzw. https für verschlüsselte Verbindungen), sondern mit ws (bzw. mit wss für verschlüsselte Verbindungen). Eine URL könnte also beispielsweise so aussehen:

```
ws://localhost:8080/puva/counter
```

Wird z.B. mit JavaScript-Code eine WebSocket-Verbindung mit dieser URL geöffnet, wird nach dem Aufbau der TCP-Verbindung folgende GET-Anfrage an den Server geschickt:

```
GET /puva/counter HTTP/1.1
Host: localhost:8080
User-Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:55.0) ... Firefox/55.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
Accept-Language: de,en-US;q=0.7,en;q=0.3
Accept-Encoding: gzip, deflate
Sec-WebSocket-Version: 13
Origin: http://localhost:8080
Sec-WebSocket-Extensions: permessage-deflate
Sec-WebSocket-Key: LET07hhYpVFuGjd9zFB6sw==
DNT: 1
Connection: keep-alive, Upgrade
Pragma: no-cache
Cache-Control: no-cache
Upgrade: websocket
```

In der GET-Anfrage taucht u.a. die vorgegebene URL auf. Die entscheidende zusätzliche Information ist Upgrade, womit der Client den Wunsch äußert, diese Verbindung für WebSockets zu verwenden. Wenn der Server WebSockets unterstützt, akzeptiert er diesen Wunsch durch folgende Antwort:

```
HTTP/1.1 101 Switching Protocols
Server: Apache-Coyote/1.1
Upgrade: websocket
Connection: upgrade
Sec-WebSocket-Accept: VgXTUyhd3tpcTV3wz+WaRcSH8b8=
Sec-WebSocket-Extensions: permessage-deflate
Date: Mon, 18 Sep 2017 19:47:34 GMT
```

Ab diesem Zeitpunkt wird auf dieser Verbindung nicht länger das HTTP-Protokoll verwendet. Die gesendeten Daten kommen ab jetzt ausschließlich von den Client- und Server-Anwendungen. Diese sind somit vollkommen frei, in welchem Format sie ihre Daten senden. Wichtig ist natürlich wie bei jedem Protokoll, dass sich Client und Server über die ausgetauschten Datenformate einig sind. Eine von sehr vielen Möglichkeiten wäre zum Beispiel wieder die Nutzung des JSON-Formats. Außerdem kann der Server nicht nur auf erhaltene Anfragen eines Clients antworten, sondern er kann zu jeder Zeit von sich aus Daten an den Client senden.

Zur Realisierung einer WebSocket-Anwendung in Java werden auf der Server-Seite Klassen bereitgestellt, die wie die Klassen bei den RESTful WebServices aus keiner anderen Klasse abgeleitet sein müssen (im Gegensatz zu Servlet-Klassen, die aus HttpServlet abge-

leitet werden), sondern mit bestimmten Annotationen gekennzeichnet werden. Bei den WebSockets muss eine Java-Klasse mit `@ServerEndpoint` annotiert werden, wobei auch hier wie in `@WebServlet` bzw. in `@Path` der Bezeichner angegeben wird, auf den mit den Methoden dieser Klasse reagiert werden soll. Für die oben angegebene URL sieht diese Annotation in einer Anwendung namens puva damit so aus:

```
@ServerEndpoint("/counter")
```

Genauso wenig, wie die WebSocket-Klasse auf dem Server aus einer spezifischen Klasse abgeleitet sein muss, müssen spezielle Methoden implementiert werden. Als Reaktion auf die Ereignisse des Verbindungsauflaufs und -abbaus sowie des Eintreffens neuer Nachrichten können Methoden bereitgestellt werden, die mit `@OnOpen`, `@OnClose` und `@OnMessage` annotiert sind. Die mit `@OnOpen` annotierte Methode sollte einen Parameter des Typs `Session` (aus dem Package `javax.websocket`, nicht zu verwechseln mit einer `HttpSession`) haben. Der beim Aufruf der Methode übergebene Parameter repräsentiert die WebSocket-Verbindung. Dieses Session-Objekt kann dann zum Senden von Daten verwendet werden. Tritt ein Fehler ein, so wird eine mit `@OnError` gekennzeichnete Methode aufgerufen, falls es eine solche gibt.

Auch mit WebSockets soll wieder der vertraute Zähler realisiert werden. Dabei soll es einen gemeinsamen Zähler für alle Clients geben. Im Gegensatz zu den Servlets ist es bei WebSockets nun allerdings so, dass für jeden Client ein neues Objekt unserer mit `@ServerEndpoint` annotierten Klasse auf dem Server erzeugt wird. Wir können somit den Zähler also beispielsweise nicht als Attribut unserer Klasse realisieren. Wir verwenden der Einfachheit halber (auch wenn es nicht schön ist) in diesem Fall ein statisches Attribut, das auf ein Zähler-Objekt zeigt, dessen Methoden alle `synchronized` sind. Da alle Clients über eine Änderung des Zählers benachrichtigt werden sollen, müssen alle Objekte der mit `@ServerEndpoint` annotierten Klasse auf alle Session-Objekte Zugriff haben. Wie für den Zähler verwenden wir auch in diesem Fall ein statisches Attribut, dieses Mal in Form einer Liste, die für den parallelen Zugriff geeignet (also `thread-safe`) ist.

Den (gekürzten) Programmcode für die Server-Seite zeigt Listing 7.24.

#### **Listing 7.24**

```
import java.io.IOException;
import java.util.*;
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;

class Counter
{
    private int counter;
    //Methoden incremenr, reset und get (alle synchronized): ...
}

@ServerEndpoint("/counter")
public class WebSocketCounter
{
    private static Counter c = new Counter();
    private static List<Session> connections =
        Collections.synchronizedList(new LinkedList<Session>());
    private Session session;
```

```
@OnOpen
public void start(Session s)
{
    session = s;
    connections.add(session);
    broadcast("someone joined");
}

@OnClose
public void end()
{
    connections.remove(session);
    session = null;
    broadcast("someone left");
}

@OnMessage
public void incoming(String message)
{
    String filteredMessage = message.toString();
    int value;
    if(filteredMessage.equals("increment"))
    {
        value = c.increment();
    }
    else if(filteredMessage.equals("reset"))
    {
        value = c.reset();
    }
    else
    {
        value = c.get();
    }
    broadcast("counter = " + value);
}

private static void broadcast(String msg)
{
    for(Session s: connections)
    {
        try
        {
            s.getBasicRemote().sendText(msg);
        }
        catch (IOException e)
        {
            connections.remove(s);
            try
            {
                s.close();
            }
            catch (IOException e1)
            {
            }
            broadcast("someone has been disconnected");
        }
    }
}
```

```
        }  
    }  
}
```

Der JavaScript-Code auf Client-Seite ist bezüglich der Nutzung der WebSocket-Verbindung ähnlich. Auch hier kann man mit Methoden auf das Öffnen und das Schließen der Verbindung sowie auf das Eintreffen von Daten reagieren. Typischerweise werden auf dem Client bei einer Benutzeraktion (z.B. das Anklicken eines Buttons) Daten gesendet, während beim Empfangen von Daten diese auf der Webseite angezeigt werden. Der JavaScript-Code zu dieser Anwendung kann auf der Webseite zu diesem Buch heruntergeladen werden.

## ■ 7.11 Zusammenfassung

In diesem Kapitel wurde in die Entwicklung webbasierter Anwendungen mit Servlets und JSF eingeführt. Eine webbasierte Anwendung besteht im einfachsten Fall aus Programmcode, der nur auf der Serverseite ausgeführt wird und im Regelfall HTML-Seiten generiert. Dieser Programmcode ist keine vollständige Anwendung, sondern ergänzt einen Web-Server wie den Tomcat-Server von Apache um anwendungsspezifische Teile. Auf der Clientseite wird ein Browser wie z.B. Internet Explorer oder Firefox verwendet. In vielen Fällen wird heute zum Browser nicht reiner HTML-Text gesendet, sondern in den HTML-Text sind neben Bildern auch kleine Programmteile eingebaut (in der Regel in JavaScript geschrieben), die dann vom Browser ausgeführt werden.

Ein Servlet wird aus der Klasse HttpServlet abgeleitet und kann eine oder mehrere der Methoden doGet, doPost, init und destroy überschreiben. Wenn ein Servlet durch eine HTTP-GET- oder HTTP-POST-Anfrage angesprochen wird, dann wird die entsprechende Methode doGet bzw. doPost aufgerufen. Über die beiden Parameter dieser Methoden hat man Zugriff auf den Kopf- und Datenteil der HTTP-Anfrage und der zu erzeugenden HTTP-Antwort. So kann man aus der Anfrage auf bequeme Art und Weise mit Hilfe geeigneter Methoden die in einem Formular eingegebenen Daten auslesen. Oder man kann im Kopfteil der Antwort den Statuscode und den MIME-Typ festlegen sowie den HTML-Text für den Datenteil der HTTP-Antwort angeben. Durch Servlet-Methoden, deren Ausführung durch Aufruf der Thread.sleep-Methode künstlich verlängert wird, kann man erkennen, dass jede HTTP-Anfrage in einem eigenen Thread ausgeführt wird. Für alle Anfragen wird dasselbe Servlet-Objekt benutzt. Wie bei RMI ist bei der Nutzung gemeinsamer Daten auf korrekte Synchronisation zu achten.

Für Daten werden im Rahmen webbasierter Anwendungen mehrere Geltungsbereiche (Scopes) unterschieden:

- Daten, die allen Anwendern in gleicher Weise zur Verfügung stehen, können als Attribute von Servlet-Klassen realisiert werden. Wenn allerdings dieselben Daten von mehreren Servlets benötigt werden, empfiehlt sich eine Speicherung der Daten im ServletContext. Falls die Daten auch verändert werden, müssen alle lesenden und schreibenden Zugriffe synchronisiert werden.

- Daten, die für jede Benutzerin unterschieden werden müssen wie z.B. in einer Web-Shop-Anwendung die Inhalte von Warenkörben, können in so genannten Session-Objekten abgelegt werden. Mit Sessions können Benutzer unterschieden werden. Sessions werden in der Regel mit Cookies realisiert, auf die man als Servlet-Entwickler auch direkten Zugriff hat. Über Registerkarten eines Browsers kann man erreichen, dass mehrere HTTP-Anfragen für dieselbe Session parallel abgewickelt werden. Aus diesem Grund muss auch der Zugriff auf Daten, deren Geltungsbereich eine Session ist, synchronisiert erfolgen.
- Durch Include- und Forward-Anweisungen kann eine HTTP-Anfrage an andere Servlets delegiert werden (dies wurde in diesem Buch nicht behandelt). Vorverarbeitete Daten können dabei im Request-Objekt abgelegt werden und so an das nächste Servlet übergeben werden. Wenn die Daten nur über dieses Request-Objekt erreichbar sind und explizit keine weiteren Threads gestartet werden, muss der Zugriff auf diese Daten nicht synchronisiert werden.
- Schließlich gibt es noch Daten, die nur über lokale Variablen erreichbar sind. Der Zugriff auf solche Objekte muss nicht synchronisiert werden.

Längere HTML-Texte sind nur mit Mühe in Servlets zu erstellen. Durch JSF ist eine wesentlich bessere Trennung zwischen HTML und Java-Code möglich. Statt HTML wird ein spezieller Dialekt namens XHTML verwendet, der in HTML übersetzt wird. Darin können Methoden von Java-Klassen angesprochen werden, die durch Annotationen als Managed Beans gekennzeichnet sind. Für die Managed Beans kann man unterschiedliche Scopes festlegen: Application Scope, Session Scope, View Scope oder Request Scope. Mit Hilfe von JSF lassen sich webbasierte Anwendungen relativ leicht gemäß des MVP-Musters gestalten.

Ein Nachteil dieser traditionellen, webbasierten Anwendungen ist, dass bei jedem Client-Request der Server immer eine komplett neue HTML-Seite produziert, mit der er antwortet. Dieser Nachteil lässt sich durch den Einsatz von AJAX beheben. Damit können nur die Daten vom Server abgerufen werden, die sich geändert haben. Anschließend können diese neuen Daten durch einen Zugriff auf den DOM-Baum in die HTML-Seite eingebaut werden. Mit JSF wird die Verwendung von AJAX extrem erleichtert. Es ist nicht nötig, dazu eigene JavaScript-Funktionen zu programmieren.

WebServices nutzen das HTTP-Protokoll für nahezu beliebige Anwendungen. Auf Client-Seite befindet sich kein Browser, sondern im Allgemeinen eine spezielle Anwendung, die in nahezu jeder beliebigen Programmiersprache geschrieben sein kann. Nachdem SOAP als WebService mit seinen komplizierten XML-Inhalten aus der Mode gekommen ist, haben sich RESTful WebServices etabliert, die unterschiedliche Datenformate unterstützen. Besonders häufig wird dabei JSON verwendet.

Auch bei WebServices erfolgt die Kommunikation nach dem Request-Response-Prinzip. WebSockets erlauben den Aufbau einer Verbindung mit einem ersten Austausch von Informationen über HTTP. Anschließend haben Client und Server völligen Freiraum, wie sie die vorhandene Verbindung weiter nutzen möchten. Die Nutzung von HTTP ist nicht mehr zwingend. Es ist damit auch möglich, dass der Server eine Nachricht an den Client senden kann, ohne dass der Client zuvor einen Request an den Server gesendet hat.

Lizenziert für tuan.hoang@mail.de.

© 2018 Carl Hanser Fachbuchverlag. Alle Rechte vorbehalten. Keine unerlaubte Weitergabe oder Vervielfältigung.

# Literatur

Das Literaturverzeichnis ist thematisch gegliedert. Zuerst werden Bücher empfohlen, welche helfen sollen, eventuell vorhandene Lücken bei den Java-Grundlagen zu schließen, insbesondere was die Neuerungen von Java 8 und 9 anbelangt. Diese Bücher behandeln darüber hinaus Themen dieses Buches, allerdings für die meisten Themen in etwas kompakterer Form. Ferner bieten sie Erweiterungen und Vertiefungen in anderen Gebieten.

Anschließend folgen drei Blöcke von Literaturhinweisen, welche die in diesem Buch behandelten Themen der parallelen, der Oberflächen- und der verteilten Programmierung in Java vertiefen, erweitern oder aus einer anderen Perspektive beleuchten.

## Java allgemein

- *Michael Inden*: Der Weg zum Java-Profi: Konzepte und Techniken für die professionelle Java-Entwicklung.  
4. Auflage, dpunkt.verlag, 2018.
- *Michael Inden*: Java 8 – Die Neuerungen: Lambdas, Streams, Date and Time API und JavaFX im Überblick.  
2. Auflage, dpunkt.verlag, 2015.
- *Michael Inden*: Java 9 – Die Neuerungen: Syntax- und API-Erweiterungen und Modularisierung im Überblick.  
1. Auflage, dpunkt.verlag, 2017.
- *Christian Ullenboom*: Java ist auch eine Insel.  
13. Auflage, Rheinwerk Computing, 2017.

## Parallele Programmierung mit Java

- *Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, David Holmes, Tim Peierls*: Java Concurrency in Practice.  
1<sup>st</sup> Edition, Addison-Wesley Longman, 2006.
- *Jörg Hettel, Manh Tien Tran*: Nebenläufige Programmierung mit Java – Konzepte und Programmiermodelle für Multicore-Systeme.  
1. Auflage, dpunkt.verlag, 2016.

## JavaFX

- *Anton Epple*: JavaFX: Grundlagen und fortgeschrittene Techniken.  
1. Auflage, dpunkt.verlag, 2015.
- *Kishori Sharan*: Learn JavaFX: Building User Experience and Interfaces with Java 8.  
1. Auflage, Apress, 2015.
- *Ralph Steyer*: Einführung in JavaFX: Moderne GUIs für RIAs und Java-Applikationen.  
1. Auflage, Springer Vieweg, 2014.

## Verteilte Programmierung mit Java

- *Dietmar Abts*: Grundkurs Java: Von den Grundlagen bis zu Datenbank- und Netzanwendungen.  
9. Auflage, Springer Vieweg, 2016.
- *Michael Inden*: Der Java-Profi: Persistenzlösungen und REST-Services: Datenaustauschformate, Datenbankentwicklung und verteilte Anwendungen.  
1. Auflage, dpunkt.verlag, 2016.
- *Bernd Müller*: Java Server Faces 2.0.  
2. Auflage, Hanser Verlag, 2010.
- *Marcus Schießer, Martin Schmollinger*: Workshop Java EE 7: Ein praktischer Einstieg in die Java Enterprise Edition mit dem Web Profile.  
2. Auflage, dpunkt.verlag, 2014.

# Index

## Symbole

@ApplicationScoped 438  
@Consumes 451  
@DELETE 451  
@GET 451  
@ManagedBean 433f.  
@NoneScoped 438  
@OnClose 458  
@OnError 458  
@OnMessage 458  
@OnOpen 458  
@Path 451  
@PathParam 451  
@POST 451  
@Produces 451  
@PUT 451  
@RequestScoped 438  
@ServerEndpoint 458  
@SessionScoped 438  
@ViewScoped 438  
@WebFilter 424  
@WebListener 405  
@WebServlet 392  
@XHTML 431

## A

Abstract Window Toolkit 210  
accept 184, 290  
acquire 110, 169  
ActionEvent 215  
activeCount 95  
activeGroupCount 95  
add 172, 212  
addCookie 414  
Adressenabbildung 406  
AJAX 443  
aktive Klasse 107  
aktives Objekt 107  
aktives Warten 37, 46, 67, 415  
Alert 232  
allMatch 185

allOf 192  
AlreadyBoundException 328  
AnchorPane 223  
Animation 233  
Annotation 403  
Anwendungsprotokoll 282  
Anwendungsschicht 268  
anyMatch 185  
anyOf 192  
Application 211  
ApplicationContext 403  
Application Layer 268  
applyToEitherAsync 192  
Arc 227  
Architekturmuster 233  
ArrayBlockingQueue 172  
ASCII-Protokoll 295, 300, 384  
Asynchronous JavaScript And XML 443  
atomar 43, 165  
Atomarität 43  
Audio-Video-Konferenz 268, 271  
Auftragnehmer 273  
Auskundsdienst 317  
AutoCloseable 281  
average 182, 185  
await 160, 170  
awaitTermination 154  
AWT 210

## B

Beobachter 216  
Betriebsmittel 194  
Betriebsmittelgraph 199  
Betriebsmitteltyp 201  
Betriebsmittelverwalter 201  
Betriebssystem 17f.  
Big Data 180  
bind 328  
Binding 219  
- bidirektional 220  
- unidirektional 219  
Bitübertragungsschicht 267

BlockingQueue 154, 172  
BorderPane 223  
BufferedInputStream 293  
BufferedOutputStream 293  
BufferedReader 293, 295  
BufferedWriter 293, 295  
Bühne 211  
busy waiting 37  
Button 210, 214, 225  
ByteArrayInputStream 293  
ByteArrayOutputStream 293

## C

call 152, 255  
Callable 152  
Callback 59, 345  
Call by Reference 335, 345  
Call by Value 335  
cancel 153, 255  
Cascading Style Sheets 233  
changed 218  
ChangeListener 225  
Channel 311  
CharacterArrayReader 293  
CharacterArrayWriter 293  
CheckBox 210, 225  
ChoiceBox 225  
Choice Button 225  
ChoiceDialog 232  
Circle 227  
Client 273  
Client-Server-Anwendung 273f.  
Clipping 231  
close 278, 290f.  
Closeable 281  
collect 186  
Collection 221  
Color 226  
ColorPicker 226  
ComboBox 225  
Command Button 225  
CommonPool 189  
Comparable 173  
Comparator 173  
CompletableFuture 187  
complete 187  
completeExceptionally 187  
concurrency 15  
Condition 160  
connect 279  
Consumer 77, 184  
Container 210  
Control 224  
Cookie 411  
Cookie (Klasse) 413  
count 185  
countDown 170

CountDownLatch 170  
createRegistry 329  
createServerSocket 381  
createSocket 381  
CSS 233  
CubicCurve 227  
currentThread 89  
currentTimeMillis 86  
CyclicBarrier 170

## D

Daemon 94  
Daemon Threads 98  
Datagramm 270  
datagrammorientiert 270  
Datagrammverlust 270  
DatagramPacket 276 ff.  
DatagramSocket 276 ff.  
DataInputStream 293  
Data Link Layer 267  
DataOutputStream 293  
Data-Streaming 180  
Datenstrom 180  
datenstromorientiert 271, 295  
datenstromorientierte Kommunikation 126  
DatePicker 226  
Dekomprimieren 294  
Delayed 173  
DelayQueue 173  
Denial-of-Service 305  
deprecated 52, 94, 100  
Deserialisierung 336, 449  
destroy 391  
Diagramm 233  
Dialog 232  
Diensterbringer 273  
Dienstschnittstelle 265  
DNS 269  
Document Object Model 444  
doGet 391  
DOM 444  
Domain Name System 269  
doPost 391  
DoubleStream 182  
down 110  
Downcall 216  
Drag and Drop 232  
drahtloses Funknetz 267  
dumpStack 99  
DynamicProxy 378

## E

EA-intensive Threads 93  
Eingabestrom 291  
EJB 442  
EL 431

elektronische Post 268  
 Ellipse 227  
 Enterprise Java Beans 442  
 Entschlüsseln 294  
 Entwurfsmuster 216  
 Erzeuger 77, 121  
 Erzeuger-Verbraucher-Prinzip 172  
 Erzeuger-Verbraucher-Problem 123, 160  
 Ethernet 267  
 EventHandler 215  
 exchange 170  
 Exchanger 170  
 execute 151  
 Executor 151, 189  
 ExecutorService 153  
 Exportieren 365  
 exportObject 365  
 Expression Language 431

## F

fair 84, 159, 169  
 Fern-Methodenaufruf 315  
 FileChooser 226  
 FileInputStream 293  
 FileOutputStream 293  
 FileReader 293  
 FileWriter 293  
 filter 181  
 Firewall 277  
 Fließband 180  
 FlowPane 222  
 flush 295  
 Flusskontrolle 271  
 forEach 185  
 Fork-Join-Framework 173  
 ForkJoinPool 173  
 ForkJoinTask 175  
 fromJson 451  
 Function 188  
 Functional Interface 24  
 funktionale Schnittstelle 24  
 Future 152, 187  
 FutureTask 187  
 FXML 232

## G

gegenseitiger Ausschluss 199,  
 generate 183  
 getAllByName 277  
 getAttribute 403  
 getByIdName 277  
 getChildren 212  
 getCompletedTaskCount 155  
 getCookies 414  
 getDelay 173  
 getHeader 396

getHeaderNames 396  
 getHoldCount 159  
 getHostAddress 276  
 getHostName 276  
 getIId 413  
 getInetAddress 290  
 getInputStream 291, 294  
 GET-Kommando 385  
 getLargestPoolSize 155  
 getLocalAddress 278  
 getLocalHost 277  
 getLocalPort 278, 290  
 getMethod 397  
 getName 28  
 getOutputStream 291, 294  
 getParameter 394  
 getParameterNames 396  
 getParameterValues 396  
 getPriority 86  
 getQueueLength 159  
 getRegistry 330  
 getRemoteAddr 397  
 getRemoteHost 397  
 getServletContext 403  
 getSession 407  
 getSoTimeout 278  
 getState 99  
 getThreadGroup 95f.  
 getWriter 393  
 Google Web Toolkit 457  
 GridPane 223  
 GSON 450  
 GWT 457

## H

handle 215  
 HBox 211, 222  
 Herunterladen von Dateien 425  
 Hibernate 442  
 Hintergrund-Threads 98  
 Hochladen von Dateien 425, 428  
 holdsLock 99  
 HTTP 295, 384  
 HTTP-Anfrage 384, 390  
 HTTP-Antwort 385, 391  
 HttpServlet 391  
 HttpServletRequest 396  
 HttpServletResponse 396  
 HttpSession 407  
 Hyperlink 225  
 HyperText Transfer Protocol 384

## I

IllegalMonitorStateException 70, 82  
 ImagePattern 227  
 InetAddress 276

init 391  
InputStream 291f.  
InputStreamReader 294  
Interaktionselement 210  
Internet Protocol 268  
interrupt 54, 158  
interrupted 55  
InterruptedException 30, 46, 55  
Interrupt-Flag 54  
IntStream 182  
invalidate 409  
Invariante 52, 65, 146  
InvocationHandler 379  
invoke 379  
invokeAll 153, 155  
invokeAny 153  
IP 268  
IP-Adresse 268f.  
IPv4 269  
IPv6 269  
isAlive 45  
isCancelled 153, 256  
isDaemon 98  
isDone 153  
isInterrupted 54  
isReachable 277  
isRunning 258  
iterate 183

## J

JavaFX 210  
JavaFX Application Thread 243  
JavaFX-Collection 221  
JavaScript Object Notation 340, 448  
Java Server Faces 431  
Java Server Pages 431  
JAX-RS 451  
Jersey 451  
join 46, 190  
joinGroup 286  
JSF 431  
JSON 340, 448f.  
JSP 431

## K

Kommunikationsprotokoll 265  
Komprimieren 294  
konsistenter Zustand 65, 146  
Konsistenz 65, 146  
Konsistenzbedingung 65, 146  
kritischer Abschnitt 111  
Kunde 273

## L

Label 210f., 225  
Labeled 224  
Lambda-Ausdruck 23  
launch 211  
Layout 210  
leaveGroup 286  
Leitungsschicht 267  
Leser-Schreiber-Problem 140  
Lesesperre 159  
limit 184  
Line 227  
LinearGradient 226  
LinkedBlockingQueue 172  
list 328  
ListView 226  
localhost 269  
LocateRegistry 329  
lock 157f.  
Lock 157f.  
lock-free 169  
lockInterruptibly 158  
LongStream 182  
lookup 320, 328

## M

MAC 267  
MAC-Adresse 267  
mapToInt 181  
MAX\_PRIORITY 86  
Medium Access Control 267  
Medium-Zugangskontrolle 267  
Mehrkernprozessor 15  
Message-Oriented Middleware 363  
Message Queue 121, 123  
Migration 368  
migrieren 368  
MIN\_PRIORITY 86  
Model - View - Controller 233  
Model - View - Presenter 233  
Model - View - ViewModel 233  
MOM 363  
MouseEvent 229  
Multicast 286  
Multicast-Adresse 269  
MulticastSocket 286  
Multicore-Prozessor 15  
Mutex 110  
mutual exclusion 110  
MVC 233  
MVP 233, 364  
MVVM 233

**N**

nachrichtenorientierte Kommunikation 126  
 Naming 319f., 328  
 nanoTime 86  
 NAT 406  
 Nebenläufigkeit 15  
 Network Address Translation 406  
 Network Layer 268  
 newCondition 158, 160  
 New Input/Output 311  
 newLine 295  
 NIO-Bibliothek 311  
 Node 221  
 NORM\_PRIORITY 86  
 notify 69  
 notifyAll 82  
 Number 218

**O**

ObjectInputStream 338  
 ObjectOutputStream 338  
 Objekt-Relationale-Mapper 442  
 ObservableList 221  
 ObservableMap 221  
 ObservableSet 221  
 ObservableValue 218  
 Observer 216  
 offer 172  
 ORM 442  
 OutputStream 291f.  
 OutputStreamWriter 294

**P**

p 110  
 Paint 226  
 Pane 221  
 parallel 186  
 Parallelität 15f.  
 - dynamisch 301  
 - echt 15  
 - statisch 301  
 parallelStream 186  
 passive Klasse 107  
 passives Objekt 107  
 PasswordField 226  
 peek 184  
 Philosophen-Problem 129  
 Physical Layer 267  
 ping 277  
 Pipe 126  
 Platform 246  
 poll 172  
 Polling 37, 415  
 Polygon 227  
 Polyline 227

**POP**

Portnummer 268, 270, 275, 277, 290  
 POST-Kommando 389  
 Predicate 181  
 print 393  
 println 393  
 PrintWriter 393  
 Prioritäten 86  
 PriorityBlockingQueue 173  
 Producer 77  
 Programm 17  
 ProgressBar 226  
 ProgressIndicator 226  
 Property 217  
 Protokoll 265  
 Proxy 406  
 Prozess 17  
 Prozessorzuteilungsstrategie 86  
 Pseudoparallelität 15  
 punktierte Dezimalnotation 269  
 put 154, 172

**Q**

QuadCurve 227

**R**

RadialGradient 226  
 RadioButton 210, 225  
 read 291  
 Reader 292  
 readLine 295  
 readObject 338  
 ReadOnlyBooleanProperty 218  
 ReadOnlyBooleanWrapper 218  
 ReadOnlyIntegerProperty 218  
 ReadOnlyIntegerWrapper 218  
 ReadOnlyProperty 218  
 ReadOnlyWrapper 218  
 ReadWriteLock 159  
 rebind 319, 328  
 receive 278  
 rechenintensive Threads 93  
 Rechner 17f.  
 Rectangle 227  
 RecursiveAction 175  
 RecursiveTask 175  
 reduce 185  
 Reduzieroperationen 185  
 reentrant 42  
 ReentrantLock 159  
 ReentrantReadWriteLock 159  
 Referenzübergabe 345  
 Reflection API 378  
 Registry 330  
 Reihenfolgevertauschung 268, 270  
 release 110, 169

Remote 317  
RemoteException 317  
Remote Method Invocation 315  
removeAttribute 403  
Representational State Transfer 447  
reset 258  
restart 258  
RESTful WebServices 447  
resume 94, 100  
Return by Reference 335  
Return by Value 335  
RMI 315  
rmic 322, 377  
RMIClientSocketFactory 381  
rmid 381  
rmiregistry 322  
RMI-Registry 318, 322, 328f.  
RMIServerSocketFactory 381  
RMISocketFactory 381  
run 20  
runLater 246  
Runnable 22

## S

Sammeloperationen 185  
Scene 211  
Scenebuilder 232  
ScheduledExecutorService 157  
ScheduledService 254, 260  
ScheduledThreadPoolExecutor 157  
Scheduling 86  
Schicht 265  
Schichtenmodell 265  
Schreibsperrre 159  
Secure Socket Layer 381  
select 311  
Selector 311  
Semaphor 109, 169  
– additiv 115  
– binär 115  
Semaphoregruppe 118  
send 278  
serialisierbar 336  
serialisieren 337  
Serialisierung 336, 449  
Serializable 336  
Server 273  
ServerSocket 276, 290  
ServerSocketChannel 311  
Service 254, 258  
Servlet 391  
ServletContext 403  
Session 407  
set 187, 217  
setAttribute 403  
setContentType 393  
setDaemon 98  
setDefaultUncaughtExceptionHandler 99  
setDisable 248  
setException 187  
setHeader 397  
setLayoutX 221  
setLayoutY 221  
setMaxAge 414  
setMaxInactiveInterval 409  
setMaxPriority 94  
setName 28  
setOnAction 214  
setOnMouseDragged 229  
setOnMouseMoved 229  
setOnMousePressed 229  
setOnMouseReleased 229  
setPriority 86  
setSoTimeout 278  
setStatus 397  
setText 216, 219  
setUncaughtExceptionHandler 99  
Shape 226  
show 211  
shutdown 154  
shutdownInput 291  
shutdownNow 154  
shutdownOutput 291  
signal 160  
signalAll 160  
SimpleBooleanProperty 217  
SimpleDoubleProperty 217  
SimpleIntegerProperty 217  
SimpleLongProperty 217  
Simple Object Access Protocol 447  
SimpleObjectProperty 217  
SimpleStringProperty 217  
Skeleton 377f., ,  
sleep 29  
Slider 210, 226  
SMTP 295  
SOAP 447  
Socket 276, 290f.  
Socket-Schnittstelle 272  
sorted 187  
Sperre 38, 157  
SSL 381  
SslRMIClientSocketFactory 381  
SslRMIServerSocketFactory 381  
StackPane 223  
Stage 211  
Standard Window Toolkit 210  
start 20, 258,  
stop 52, 94, 100  
stream 181  
Stream 182  
Stub 316, 377f.,  
submit 153, 187  
Suchoperationen 185  
sum 182, 185

Supplier 188  
 supplyAsync 188  
 suspend 94, 100  
 Swing 210  
 SWT 210  
 synchronized 38  
 Synchronized-Block 39  
 Synchronized-Methode 40  
 SynchronousQueue 173  
 System  
 - eng gekoppelt 16  
 - lose gekoppelt 16  
 Szene 211

**T**

TableView 226  
 take 154, 172  
 Task 254f.  
 TBB 179  
 TCP 268, 271, 290  
 TCPSocket 296  
 Text 227  
 TextArea 226  
 TextField 226  
 TextInputControl 226  
 thenApplyAsync 188  
 thenCombineAsync 191  
 Thread 17f., 301  
 - (Klasse) 20  
 ThreadGroup 94  
 Thread-Gruppe 93  
 Threading Building Blocks 179  
 ThreadLocal 101  
 Thread-Pool 152, 154, 187  
 ThreadPoolExecutor 154  
 thread-safe 165  
 thread-sicher 165  
 TilePane 223  
 Time-To-Live 286  
 TimeUnit 153  
 ToggleButton 225  
 ToggleGroup 225  
 TolntFunction 181  
 toJson 450  
 transient 340  
 Transmission Control Protocol 268, 271  
 transparent 315  
 Transparenz 317  
 Transport Layer 268  
 Transportsschicht 268  
 Traversierungsoperationen 185  
 TreeTableView 226  
 TreeView 226  
 Treiber 268  
 tryLock 158  
 try-with-resources 279, 281

**U**

Überlastkontrolle 271  
 UDP 268, 270, 277  
 UI Control 210  
 unbind 328  
 uncaughtException 99  
 Unexportieren 366  
 unexportObject 366  
 Unicast-Adresse 269  
 UnicastRemoteObject 317, 365  
 Universal Resource Locator 384  
 unlock 157  
 unteilbar 43, 165  
 unzuverlässig 268  
 up 110  
 Upcall 216  
 updateMessage 255  
 updateProgress 255  
 updateTitle 255  
 updateValue 255  
 URL 314, 384  
 URLConnection 314  
 User Datagram Protocol 268, 270  
 User Threads 98

**V**

v 110  
 VBox 211, 222  
 verbindungslos 268, 270  
 verbindungsorientiert 271  
 Verbraucher 77, 121  
 Verklemmung 75, 118, 134, 194  
 Verlust 268  
 Vermittlungsschicht 268  
 Verschlüsseln 294  
 verteilte Anwendungen 16  
 verteilte Systeme 16ff.  
 Verteilung 16  
 Verteilungstransparenz 316  
 Virtualisierung 16  
 volatile 43  
 Vordergrund-Threads 98

**W**

wait 69, 76  
 WebServices 447  
 WebSockets 456  
 well-known port number 273  
 Wertübergabe 335  
 Widget 210  
 Wiki 419  
 wohlbekannte Portnummer 273f.  
 Worker 254  
 World Wide Web (WWW) 268  
 write 291, 295

writeln 293  
writeObject 338  
Writer 292  
WWW 268

## Y

yield 99

## Z

Zustandsübergangsdiagramm 105  
zuverlässig 27

# PARALLELE UND VERTEILTE ANWENDUNGEN IN JAVA //

- Einstieg in zwei grundlegende Konzepte der Informatik
- Inkl. Programmierung grafischer Benutzeroberflächen
- Alle Programme des Buches auf der Webseite: [puva.hochschule-trier.de](http://puva.hochschule-trier.de)

Für Nutzer ist es selbstverständlich, dass sie mehrere Programme gleichzeitig verwenden können oder dass Programme so komplex sind, dass sie auf mehrere Rechner zugreifen müssen. Aber wie werden solche Anwendungen programmiert?

Das Lehrbuch behandelt zwei eng miteinander verknüpfte Basisthemen der Informatik: die Programmierung paralleler (nebenläufiger) und verteilter Anwendungen. Es werden zunächst anhand zahlreicher Beispiele grundlegende Synchronisationskonzepte für die Programmierung paralleler Abläufe präsentiert. Neben den »klassischen« Synchronisationsmechanismen von Java werden auch die Konzepte aus der Java-Concurrency-Klassenbibliothek vorgestellt.

Weiteres Basiswissen, das zum Verständnis des Buchs notwendig ist, etwa über grafische Benutzeroberflächen, das MVC-Entwurfsmuster oder Rechnernetze, wird im Buch anschaulich und praxisnah vermittelt.

Die 5. Auflage wurde neu bearbeitet und aktualisiert. Insbesondere wurde von Swing auf JavaFX und von JSP auf JSF umgestellt. Außerdem wurde das Buch um neue Themenbereiche wie das Data-Streaming-Framework, CompletableFuture und RESTful WebServices erweitert.

Das Lehrbuch wendet sich an Studierende der Informatik und ingenieurwissenschaftlicher Studiengänge mit Grundkenntnissen in Java und Objektorientierung sowie Softwareentwickler.

**Prof. Dr. Rainer OECHSLE** lehrt an der Hochschule Trier am Fachbereich Informatik und vertritt die Fachgebiete parallele, verteilte, mobile und komponentenbasierte Software-Systeme.

## AUS DEM INHALT //

- Grundlegende Synchronisationskonzepte in Java
- Fortgeschrittene Synchronisationskonzepte in Java
- Parallelität und grafische Benutzeroberflächen
- Verteilte Anwendungen mit Sockets
- Verteilte Anwendungen mit RMI
- Webbasierte Anwendungen mit Servlets und JSF

## UNSER BUCHTIPP FÜR SIE //



Ratz, Grundkurs  
Programmieren in Java  
2014, 745 Seiten, FlexCover € 34,99.  
ISBN 978-3-446-44073-9

HANSER

