

HANSER



Leseprobe

zu

„Parallele und verteilte Anwendungen in Java, 5.A.“

von Rainer Oechsle

ISBN (Buch): 978-3-446-45118-6

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

Weitere Informationen und Bestellungen unter
<http://www.hanser-fachbuch.de/978-3-446-45118-6>

sowie im Buchhandel

© Carl Hanser Verlag, München

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`s 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

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`s, 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 Programmiertechniken (FOPT)“ im Rahmen des Informatik-

Fernstudiums an der Hochschule Trier zu belegen. Hier können Sie zu den Themen dieses Buches Einsendeaufgaben bearbeiten, an zusätzlichen Tutorien (per Videokonferenz) teilnehmen sowie ein einwöchiges Präsenzpraktikum absolvieren. Nähere Informationen hierzu, insbesondere über die Voraussetzungen für die Belegung, über die Kosten sowie über die weiteren Module des Fernstudiums, finden Sie unter 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 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.

Konz-Oberemmel, im Januar 2018

Rainer Oechsle

Inhalt

1	Einleitung	15
1.1	Parallelität, Nebenläufigkeit und Verteilung	15
1.2	Programme, Prozesse und Threads	16
2	Grundlegende Synchronisationskonzepte in Java	20
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	Zwangsweises 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
3	Fortgeschrittene Synchronisationskonzepte in Java	108
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 Completable Futures 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
5	Verteilte Anwendungen mit Sockets	264
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
6	Verteilte Anwendungen mit RMI	315
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
7	Webbasierte Anwendungen mit Servlets und JSF	383
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
Literatur	463
Index	465

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 übersetzbar ist.

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

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

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

```

Thread 2 (7)
Thread 3 (8)
Thread 2 (8)
Thread 3 (9)
Thread 2 (9)
Thread 1 (53)
Thread 2 (10)
Thread 1 (54)
Thread 2 (11)
Thread 1 (55)
Thread 2 (12)
Thread 1 (56)
Thread 3 (10)
Thread 2 (13)
Thread 1 (57)
Thread 3 (11)
Thread 2 (14)
...

```

Die Threads laufen scheinbar oder echt gleichzeitig. Welcher Fall vorliegt, kann anhand der Ausgabe nicht eindeutig unterschieden werden. Falls der Ablauf nur scheinbar gleichzeitig ist (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.

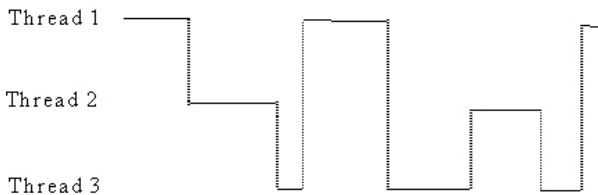


Bild 2.1
Ausführungsintervalle von
drei Threads

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

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

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

■ 2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte

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

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
Auskunftsdienst 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 273 f.
 Clipping 231
 close 278, 290 f.
 Closeable 281
 collect 186
 Collection 221
 Color 226
 ColorPicker 226
 ComboBox 225
 Command Button 225
 CommonPool 189
 Comparable 173
 Comparator 173
 Completable Futures 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
 datagramorientiert 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
- getByName 277
- getChildren 212
- getCompletedTaskCount 155
- getCookies 414
- getDelay 173
- getHeader 396

- getHeaderNames 396
- getHoldCount 159
- getHostAddress 276
- getHostName 276
- getId 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 95 f.
- 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 295
 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
 Schreibsperre 159
 Secure Socket Layer 381
 select 311
 Selector 311
 Semaphor 109, 169
 – additiv 115
 – binär 115
 Semaphorgruppe 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
 Suchooperationen 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 254 f.
 TBB 179
 TCP 268, 271, 290
 TCPSocket 296
 Text 227
 TextArea 226
 TextField 226
 TextInputControl 226
 thenApplyAsync 188
 thenCombineAsync 191
 Thread 17 f., 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
 ToIntFunction 181
 toJson 450
 transient 340
 Transmission Control Protocol 268, 271
 transparent 315
 Transparenz 317
 Transport Layer 268
 Transportschicht 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 16 ff.
 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
 wohlbekanntes Portnummer 273 f.
 Worker 254
 World Wide Web (WWW) 268
 write 291, 295

writeInt 293
writeObject 338
Writer 292
WWW 268

Y

yield 99

Z

Zustandsübergangsdigramm 105
zuverlässig 271