PARALLELE UND VERTEILTE ANWENDUNGEN IN JAVA

3., erweiterte Auflage
Inhalt

1 Einleitung ........................................................................................................................................... 13
1.1 Parallelität, Nebenläufigkeit und Verteilung ................................................................................ 13
1.2 Programme, Prozesse und Threads ................................................................................................. 14

2 Grundlegende Synchronisationskonzepte in Java ............................................................................ 19
2.1 Erzeugung und Start von Java-Threads .......................................................................................... 19
   2.1.1 Ableiten der Klasse Thread ..................................................................................................... 19
   2.1.2 Implementieren der Schnittstelle Runnable ............................................................................ 21
   2.1.3 Einige Beispiele ..................................................................................................................... 22
2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte ............................................................. 28
   2.2.1 Erster Lösungsversuch ............................................................................................................. 31
   2.2.2 Zweiter Lösungsversuch ......................................................................................................... 33
2.3 Synchronized und volatile ............................................................................................................. 35
   2.3.1 Synchronized-Methoden ......................................................................................................... 35
   2.3.2 Synchronized-Blöcke ............................................................................................................. 36
   2.3.3 Wirkung von synchronized .................................................................................................. 37
   2.3.4 Notwendigkeit von synchronized ......................................................................................... 39
   2.3.5 volatile .................................................................................................................................. 40
   2.3.6 Regel für die Nutzung von synchronized ............................................................................... 40
2.4 Ende von Java-Threads ................................................................................................................... 42
   2.4.1 Asynchrone Beauftragung mit Abfragen der Ergebnisse ......................................................... 43
   2.4.2 Zwangsweises Beenden von Threads .................................................................................... 48
   2.4.3 Asynchrone Beauftragung mit befristetem Warten ............................................................... 53
   2.4.4 Asynchrone Beauftragung mit Rückruf (Callback) ............................................................... 54
2.5 Wait und notify ............................................................................................................................... 57
   2.5.1 Erster Lösungsversuch ............................................................................................................. 57
   2.5.2 Zweiter Lösungsversuch ......................................................................................................... 58
   2.5.3 Korrekte Lösung mit wait und notify ....................................................................................... 59
2.6 NotifyAll .......................................................................................................................................... 67
   2.6.1 Erzeuger-Verbraucher-Problem mit wait und notify ............................................................ 68
   2.6.2 Erzeuger-Verbraucher-Problem mit wait und notifyAll ...................................................... 71
   2.6.3 Faires Parkhaus mit wait und notifyAll .................................................................................. 74
2.7 Prioritäten von Threads ................................................................................................................... 75
<table>
<thead>
<tr>
<th>3.8 Ursachen für Verklemmungen</th>
<th>153</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8.1 Beispiele für Verklemmungen mit synchronized</td>
<td>154</td>
</tr>
<tr>
<td>3.8.2 Beispiele für Verklemmungen mit Semaphoren</td>
<td>157</td>
</tr>
<tr>
<td>3.8.3 Bedingungen für das Eintreten von Verklemmungen</td>
<td>158</td>
</tr>
<tr>
<td>3.9 Vermeidung von Verklemmungen</td>
<td>159</td>
</tr>
<tr>
<td>3.9.1 Anforderung von Betriebsmitteln „auf einen Schlag“</td>
<td>162</td>
</tr>
<tr>
<td>3.9.2 Anforderung von Betriebsmitteln gemäß einer vorgegebenen Ordnung</td>
<td>163</td>
</tr>
<tr>
<td>3.9.3 Anforderung von Betriebsmitteln mit Bedarfsanalyse</td>
<td>164</td>
</tr>
<tr>
<td>3.10 Modellierung mit Petri-Netzen</td>
<td>171</td>
</tr>
<tr>
<td>3.10.1 Petri-Netze</td>
<td>171</td>
</tr>
<tr>
<td>3.10.2 Modellierung der Nutzung von Synchronized-Methoden</td>
<td>173</td>
</tr>
<tr>
<td>3.10.3 Modellierung von wait, notify und notifyAll</td>
<td>176</td>
</tr>
<tr>
<td>3.11 Zusammenfassung</td>
<td>179</td>
</tr>
</tbody>
</table>
4 Parallelität und grafische Benutzeroberflächen ........................................ 181
4.1 Einführung in die Programmierung grafischer Benutzeroberflächen mit Swing ........................................ 182
  4.1.1 Einige erste Beispiele .................................................................................. 182
  4.1.2 Ereignisbehandlung .................................................................................... 186
  4.1.3 Container ..................................................................................................... 190
  4.1.4 Primitive Interaktionselemente ................................................................... 193
  4.1.5 Grafikprogrammierung .............................................................................. 194
  4.1.6 Applets ....................................................................................................... 199
4.2 MVC .................................................................................................................. 201
  4.2.1 Prinzip von MVC ...................................................................................... 202
  4.2.2 MVC für die Entwicklung eigener Programme .......................................... 205
  4.2.3 MVC in Swing .......................................................................................... 210
4.3 Threads und Swing .......................................................................................... 212
4.4 Zusammenfassung ........................................................................................... 223

5 Verteilte Anwendungen mit Sockets ................................................................. 225
5.1 Einführung in das Themengebiet der Rechnernetze ........................................ 226
  5.1.1 Schichtenmodell ....................................................................................... 226
  5.1.2 IP-Adressen und DNS-Namen ................................................................. 230
  5.1.3 Das Transportprotokoll UDP ................................................................. 231
  5.1.4 Das Transportprotokoll TCP ................................................................... 233
5.2 Socket-Schnittstelle ....................................................................................... 234
  5.2.1 Socket-Schnittstelle zu UDP ..................................................................... 234
  5.2.2 Socket-Schnittstelle zu TCP ..................................................................... 235
  5.2.3 Socket-Schnittstelle für Java ...................................................................... 238
5.3 Kommunikation über UDP mit Java-Sockets .............................................. 239
5.4 Multicast-Kommunikation mit Java-Sockets .............................................. 246
5.5 Kommunikation über TCP mit Java-Sockets .............................................. 250
5.6 Sequenzielle und parallele Server ............................................................... 260
  5.6.1 Server mit dynamischer Parallelität .......................................................... 262
  5.6.2 Server mit statischer Parallelität ............................................................... 266
5.7 Zusammenfassung ......................................................................................... 271

6 Verteilte Anwendungen mit RMI ................................................................. 273
6.1 Prinzip von RMI ............................................................................................. 273
6.2 Einführendes RMI-Beispiel .......................................................................... 276
  6.2.1 Basisprogramm ......................................................................................... 276
  6.2.2 RMI-Client mit grafischer Benutzeroberfläche .......................................... 280
  6.2.3 RMI-Registry ............................................................................................. 283
6.3 Parallelität bei RMI-Methodenaufrufen ...................................................... 287
6.4 Wertübergabe für Parameter und Rückgabewerte ...................................... 291
  6.4.1 Serialisierung und Deserialisierung von Objekten .................................... 292
  6.4.2 Serialisierung und Deserialisierung bei RMI ............................................. 296
6.5 Referenzübergabe für Parameter und Rückgabewerte ................................ 300
6.6 Transformation lokaler in verteilte Anwendungen ...................................... 316
  6.6.1 Rechnergrenzen überschreitende Synchronisation mit RMI .................... 317
6.6.2 Asynchrone Kommunikation mit RMI .................................................................319
6.6.3 Verteilte MVC-Anwendungen mit RMI ............................................................319
6.7 Dynamisches Umschalten zwischen Wert- und Referenzübergabe – Migration von Objekten ..321
  6.7.1 Das Exportieren und „Unexportieren“ von Objekten ........................................321
  6.7.2 Migration von Objekten ....................................................................................324
  6.7.3 Eintrag eines Nicht-Stub-Objekts in die RMI-Registry .......................................331
6.8 Laden von Klassen über das Netz ...........................................................................332
6.9 Realisierung von Stubs und Skeletons ..................................................................333
  6.9.1 Realisierung von Skeletons ...............................................................................333
  6.9.2 Realisierung von Stubs .....................................................................................334
6.10 Verschiedenes ......................................................................................................336
6.11 Zusammenfassung .................................................................................................337

7 Webbasierte Anwendungen mit Servlets und JSP ....................................................339
  7.1 HTTP ...................................................................................................................340
    7.1.1 GET ................................................................................................................340
    7.1.2 Formulare .....................................................................................................343
    7.1.3 POST ............................................................................................................345
    7.1.4 Format von HTTP-Anfragen und -Antworten ...................................................346
  7.2 Einführende Servlet-Beispiele ..............................................................................347
    7.2.1 Allgemeine Vorgehensweise .........................................................................347
    7.2.2 Erstes Servlet-Beispiel ..................................................................................348
    7.2.3 Zugriff auf Formulardaten ..............................................................................355
    7.2.4 Zugriff auf die Daten der HTTP-Anfrage und -Antwort ....................................356
  7.3 Parallelität bei Servlets .........................................................................................357
    7.3.1 Demonstration der Parallelität von Servlets .....................................................357
    7.3.2 Paralleler Zugriff auf Daten ..........................................................................359
    7.3.3 Anwendungsglobale Daten ..........................................................................362
  7.4 Sessions und Cookies .........................................................................................365
    7.4.1 Sessions ........................................................................................................366
    7.4.2 Realisierung von Sessions mit Cookies ..........................................................371
    7.4.3 Direkter Zugriff auf Cookies .........................................................................372
    7.4.4 Servlets mit länger dauernden Aufträgen .......................................................374
  7.5 Übertragung von Dateien mit Servlets .................................................................378
    7.5.1 Herunterladen von Dateien ............................................................................378
    7.5.2 Hochladen von Dateien ..................................................................................381
  7.6 JSP (Java Server Pages) .....................................................................................384
    7.6.1 Scripting-Elemente .......................................................................................384
    7.6.2 Direktiven .....................................................................................................386
    7.6.3 Aktionen .......................................................................................................387
  7.7 MVC-Prinzip mit Servlets und JSPs .....................................................................390
  7.8 MVC-Prinzip mit AJAX und GWT .......................................................................397
  7.9 Zusammenfassung ...............................................................................................405

Literatur .........................................................................................................................407

Register ..........................................................................................................................409
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, 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:

```java
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 eine Klasse des Package java.lang (so wie String) 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:**

```java
public class MyThread extends Thread {
    public void run()
    {
        System.out.println("Hello World");
    }

    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.

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


Dieses kleine nur wenige Zeilen umfassende Beispielprogramm enthält noch ein weiteres Verständnisproblem für viele Neulinge: Warum muss ein Thread-Objekt (genauer: ein Ob-

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 anwen- den).

### 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 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.

```java
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:

```java
public class SomethingToRun implements Runnable
{
    public void run()
    {
        System.out.println("Hello World");
    }
    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:

```java
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:

```java
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).

### 2.1.3 Einige Beispiele

Um das bisher Gelernte etwas zu vertiefen, betrachten wir das Beispielprogramm aus Listing 2.3:
2.1 Erzeugung und Start von Java-Threads

Listing 2.3:

```java
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 Methodenauftrag 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)
```

![Diagramm der Threads](image)

**Abbildung 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. Taktrate des Prozessors) 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),
2.1 Erzeugung und Start von Java-Threads

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 kön-
nen.

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 be-
kannten Konstruktoren der Klasse Thread ohne Argument und mit einem Runnable-Argu-
ment gibt es Konstruktoren mit einem zusätzlichen Namensargument. Damit kennen wir
nun vier Konstruktoren der Klasse Thread. Zusätzlich werden die neuen Methoden set-
Name 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 Attributes 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:

```java
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:

```java
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 nicht mehr berücksichtigt und verbraucht in dieser Phase keine Rechenzeit.


```
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)
```

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) 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.


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
2.2 Probleme beim Zugriff auf gemeinsam genutzte Objekte


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
{
    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++)
        {

```

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...


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 Programmiererteam 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 Programmiererteam 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 kennenlernen.

### 2.2.1 Erster Lösungsversuch

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

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;


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.
Register

A
accept 251
acquire 99, 149
ActionEvent 189, 193
ActionListener 187, 189
actionPerformed 187, 189
activeCount 84
activeGroupCount 84
add 152, 182
addActionListener 187, 189
addChangeListener 189
addCookie 373
addListSelectionListener 189
addMouseListener 189
Adresseabildung 366
AJAX 398
Aktion 387
aktive Klasse 95
aktives Objekt 95
aktives Warten 34, 42, 58, 374
AlreadyBoundException 284
Animation 221
Annotation 352, 363, 365, 399
Anwendungsprotokoll 242
Anwendungsschicht 230
Applet 199, 397
Applet-Tag 200
Application Layer 230
ApplicationContext 362
ArrayBlockingQueue 152
ASCII-Protokoll 256, 260, 340
AsyncCallback 400
Asynchronous JavaScript And XML 398
atomar 39, 148
Atomarität 39
AtomicInteger 148
Audio-Video-Konferenz 230, 232
Auftragnehmer 235
Auskunftsdiensst 276
await 143, 144, 149
awaitTermination 138

B
Bankier-Algorithmus 167
Bean 388
Bedarfsanalyse 167
Betriebsmittel 153
Betriebsmittelgraph 158
Betriebsmitteltyp 160
Betriebsmittelverwalter 160
Betriebssystem 15, 16
bind 284
Bitübertragungsschicht 229
BlockingQueue 139, 151
BorderLayout 183, 190
BoundedRangeModel 210
BufferedInputStream 254
BufferedOutputStream 254
BufferedReader 254, 256
BufferedWriter 254, 255, 256
busy waiting 34
ButtonGroup 193
ByteArrayInputStream 253
Register

ByteArrayOutputStream 253

C
call 136
Call by Reference 291, 300
Call by Value 291
Callable 137
Callback 54, 300, 400
cancel 137
ChangeEvent 189
ChangeListener 189
CharacterArrayReader 253
CharacterArrayWriter 253
Client 235
Client-Server-Anwendung 235, 236
close 240, 251, 252
Comparable 152
Comparator 152
compareAndSet 148
concurrency 13
Condition 143
connect 240
Consumer 67
Container-Klasse 190
Cookie 371
Cookie (Klasse) 373
countDown 149
CountDownLatch 149
createRegistry 285
createServerSocket 336
createSocket 336
currentThread 79
currentTimeMillis 76
CyclicBarrier 149

D
Daemon 83
Daemon Threads 87
Data Link Layer 229
Datagramm 232
datagrammorientiert 232
datagrammverlust 232
DatagramPacket 238, 239
DatagramSocket 238, 239, 240
DataInputStream 254
DataOutputStream 254
datenstromorientiert 233, 255
datenstromorientierte Kommunikation 113
DefaultBoundedRangeModel 210
DefaultTableModel 210
DefaultTreeModel 210
dekomprimieren 255
Delayed 152
DelayQueue 152
denial-of-Service 266
deprecated 48, 83, 88
deserialisierung 292
destroy 347
dienstbringer 235
dienstschnittstelle 226
direktive 386
DISPOSE_ON_CLOSE 186
DNS 231
DO NOTHING ON CLOSE 186
Document 210
document Object Model 398
doGet 347
dOM 398
domain Name System 231
doPost 347
down 99
drahtloses Funknetz 229
drawOval 197
dumpStack 88
DynamicProxy 334

E
EA-intensive Threads 81
eingabestrom 252
elektronische Post 230
Entschlüsseln 255
ereignisbehandlungs-Thread 215
Erzeuger 67, 109
Erzeuger-Verbraucher-Prinzip 151
Erzeuger-Verbraucher-Problem 111, 144
Ethernet 229
Event-Dispatcher 215
EventQueue 216
exchange 150
Exchanger 150
execute 135
Executor 135
ExecutorService 137
EXIT_ON_CLOSE 186
Exportieren 321
exportObject 321

F
fair 74, 142, 149
Fern-Methodenauftruf 274
FileInputStream 253
FileOutputStream 253
FileReader 253
FileWriter 253
fillOval 197
Firewall 239
FlowLayout 183, 190
flush 256
Flusskontrolle 233
Forward-Aktion 387
Future 137

G
gegenseitiger Ausschluss 99, 158
getActionCommand 198
getAllByName 239
getAttribute 362
getByName 238
getCompletedTaskCount 139
getCookies 373
getchannel 152
gethostByAddress 356
gethostbyname 356
gethostbyname 143
gethostbyname 238
gethostbyname 238
getid 372
getInetAddress 251
getInputStream 252, 255
GET-Kommando 341
getLargestPoolSize 139
getLocalAddress 240
getLocalHost 239
getLocalPort 240, 251
getMethod 357
getAddress 25
getOutputStream 252, 255
getParameter 355
getParameterNames 356
getParameterValue 356
getPriority 75
getProperty-Aktion 390
getQueueLength 142
getRegistry 286
getRemoteAddr 357
getRemoteHost 357
getservletcontext 362
getSession 366
getSofTimeout 240
getSource 189
getState 88
getThreadGroup 83, 84
getWriter 349
getX 189
getY 189
Google Web Toolkit 398
Grafikkontext 194
Graphics 194
GridLayout 185, 190
GWT 398

H
Herunterladen von Dateien 378
HIDE_ON_CLOSE 185
Hintergrund-Threads 87
Hochladen von Dateien 378, 381
holdLock 88
HTMLDocument 210
HTTP 256, 340
HTTP-Anfrage 341, 346
HTTP-Antwort 341, 346
HttpServlet 347
<table>
<thead>
<tr>
<th><strong>Register</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>HttpServletRequest 356</td>
</tr>
<tr>
<td>HttpServletResponse 356</td>
</tr>
<tr>
<td>HttpSession 367</td>
</tr>
<tr>
<td>HyperText Transfer Protocol 340</td>
</tr>
</tbody>
</table>

**I**

ICEfaces 396
IllegalStateException 60, 72
Include-Aktion 387
Include-Direktive 387
InetAddress 238
init 199, 347
InputStream 252
InputStreamReader 255
Intermediate Container 190
Internet Protocol 229
interrupt 50, 141
interrupted 51
InterruptedException 27, 43, 51
Interrupt-Flag 50
invalidate 368
Invariante 48, 57, 131
InvocationHandler 334
invoke 335
invokeAll 138, 139
invokeAndWait 216
invokeAny 138
invokeLater 216
IP 229
IP-Adresse 229, 230
IPv4 230
IPv6 231
isAlive 42
isCancelled 137
isDaemon 87
isDone 137
isInterrupted 50
isReachable 239

**J**

JApplet 199
Jar-Datei 200
Java Server Faces 396
Java Server Pages 384
JavaScript 398
JavaScript Object Notation 403
JButton 187
JCheckBox 193
JCheckBoxMenuItem 194
JComboBox 19
JFrame 182
JLabel 182
JList 194
JMenu 194
JMenuBar 194
 JMenuItem 194
join 42
joinGroup 246
JPanel 193
JPasswordField 193
JRadioButton 193
JRadioButtonMenuItem 194
 JScrollPane 193
JSeparator 194
JSplitPane 193
JSeparatorMenuItem 194
JSidePane 384
JTabbedPane 193
JTable 194
JTextArea 194
JTextField 193
JTree 194

**K**

Kommunikationsprotokoll 226
Komprimieren 255
konsistenter Zustand 57, 131
Konsistenz 57, 131
Konsistenzbedingung 57, 131
kritischer Abschnitt 100
Kunde 235
L
Layout-Manager 183
leaveGroup 246
Leitungsschicht 229
Leser-Schreiber-Problem 126
Lesesperre 143
LinkedBlockingQueue 152
list 284
ListSelectionEvent 189
ListSelectionListener 189
localhost 231
LocateRegistry 285
lock 141
Lock 141
lockInterruptibly 142
lookup 278, 284

M
MAC 229
MAC-Adresse 229
MAX_PRIORITY 75
Medium Access Control 229
Medium-Zugangskontrolle 229
Mehrkernprozessor 13
Message Queue 108, 111
Message-Oriented Middleware 319
Migration 324
migrieren 324
MIN_PRIORITY 75
Model – View – Controller 201
MOM 319
mouseClicked 189
mouseEntered 189
MouseEvent 189
mouseExited 189
MouseListener 189
mousePressed 189
mouseReleased 189
Multicast 246
Multicast-Adresse 231
MulticastSocket 246
Multicore-Prozessor 13
Mutex 99
mutual exclusion 99
MVC 201, 319, 391, 405

N
nachrichtenorientierte Kommunikation 113
Naming 278, 284
nanoTime 76
NAT 366
Nebenläufigkeit 13
Network Address Translation 366
Network Layer 229
newCondition 142, 144
newLine 256
NORM_PRIORITY 75
notify 59
notifyAll 71

O
ObjectInputStream 294
ObjectOutputStream 294
offer 152
OutputStream 252
OutputStreamWriter 255

P
p 99
Page-Direktive 387
paintComponent 194
Parallelität 13, 14
dynamisch 261
echt 13
statisch 261
passive Klasse 95
passives Objekt 95
Petri-Netz 171
Philosophen-Problem 116
Physical Layer 229
ping 239
Pipe 113
PlainDocument 210
poll 152
Polling 34, 374, 404
POP 256
Regis
ter

<table>
<thead>
<tr>
<th>Portnummer</th>
<th>230, 232, 237, 239, 251</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST-Kommando</td>
<td>345, 403</td>
</tr>
<tr>
<td>print</td>
<td>349</td>
</tr>
<tr>
<td>println</td>
<td>349</td>
</tr>
<tr>
<td>PrintWriter</td>
<td>349</td>
</tr>
<tr>
<td>Prioritäten</td>
<td>75</td>
</tr>
<tr>
<td>PriorityBlockingQueue</td>
<td>152</td>
</tr>
<tr>
<td>Producer</td>
<td>67</td>
</tr>
<tr>
<td>Programm</td>
<td>15</td>
</tr>
<tr>
<td>Protokoll</td>
<td>226</td>
</tr>
<tr>
<td>Proxy</td>
<td>366</td>
</tr>
<tr>
<td>Prozess</td>
<td>15, 16</td>
</tr>
<tr>
<td>Prozessorzuteilungsstrategie</td>
<td>75</td>
</tr>
<tr>
<td>Pseudoparallelität</td>
<td>13</td>
</tr>
<tr>
<td>Pull-Prinzip</td>
<td>404</td>
</tr>
<tr>
<td>punktierte Dezimalnotation</td>
<td>230</td>
</tr>
<tr>
<td>Push-Prinzip</td>
<td>404</td>
</tr>
<tr>
<td>put</td>
<td>139, 151</td>
</tr>
</tbody>
</table>

**R**

| read             | 252                     |
| Reader           | 252                     |
| readLine         | 256                     |
| readObject       | 294                     |
| ReadWriteLock    | 143                     |
| rebind           | 278, 284                |
| receive          | 240                     |
| rechenintensive Threads | 81                 |
| Rechner          | 15, 16                  |
| ReentrantLock    | 142                     |
| ReentrantReadWriteLock | 143            |
| Referenzübergabe | 300                     |
| Reflection API   | 333                     |
| Registry         | 286                     |
| Reihenfolgevertauschung | 230, 232     |
| release          | 99, 149                 |
| Remote           | 275                     |
| Remote Method Invocation | 273             |
| RemoteException  | 275, 276                |
| RemoteServiceRelativePath | 399        |
| removeActionListener | 187           |
| removeAttribute  | 363                     |
| repaint          | 198                     |
| RequestDispatcher| 387                     |
| resume           | 83, 88                  |
| Return by Reference | 291               |
| Return by Value  | 291                     |
| RMI              | 273                     |
| rmic             | 280, 333                |
| RMIClientSocketFactory | 336        |
| rmid             | 336                     |
| rmiregistry      | 280                     |
| RMI-Registry     | 276, 280, 284, 285      |
| RMIServerSocketFactory | 336       |
| RMISocketFactory | 336                     |
| run              | 19                      |
| Runnable         | 21                      |

**S**

| Sandbox          | 201                     |
| Sandkasten       | 201                     |
| ScheduledExecutorService | 141              |
| ScheduledThreadPoolExecutor | 141 |
| Scheduling       | 75                      |
| Schicht          | 226                     |
| Schichtenmodell  | 226                     |
| Schreibsperrre   | 143                     |
| Scope            | 388                     |
| Secure Socket Layer | 336             |
| Semaphor         | 98, 149                 |
| additiv          | 104                     |
| binär            | 104                     |
| Semaphorgruppe   | 106                     |
| send             | 240                     |
| sendRedirect     | 388                     |
| serialisierbar   | 292                     |
| serialisieren    | 292                     |
| Serialisierung   | 292                     |
| Serializable     | 292                     |
| Server           | 235                     |
| ServerSocket     | 238, 250, 251           |
| Servlet          | 347                     |
| ServletContext   | 362                     |
| Session          | 366                     |
| setAttribute     | 362                     |
| setColor         | 198                     |
Register

setContentType 349
setDaemon 87
setDefaultCloseOperation 185
setDefaultUncaughtExceptionHandler 88
setHeader 357
setLayout 185
setLocation 182
setMaxAge 373
setMaxInactiveInterval 369
setMaxPriority 83
setName 25
setPriority 75
SetProperty-Aktion 390
setSize 182
setSoTimeout 240
setStatus 357
setText 188
setUncaughtExceptionHandler 88
setVisible 183
shutdown 138
shutdownInput 252
shutdownNow 138
shutdownOutput 252
signal 143
signalAll 143
Skeleton 274, 275, 333
sleep 26
SMTP 256
Socket 238, 250, 251
Socket-Schnittstelle 234
Sperre 35, 141
SSL 336
SslRMIClientSocketFactory 336
SslRMIReadOnlySocketFactory 336
start 20
stateChanged 189
stop 48, 83, 88
Struts 396
Stub 274, 275, 333, 334
submit 138
suspend 83, 88
Swing 181
synchronized 35
Synchronized-Block 36
Synchronized-Methode 37
SynchronousQueue 152
System
   eng gekoppelt 14
   lose gekoppelt 14

T
TableModel 210, 211
Taglib-Direktive 387
take 139, 151
TCP 230, 233, 250
TCPSocket 256
Thread 15, 16, 261
   (Klasse) 19
ThreadGroup 83
Thread-Gruppe 82
ThreadLocal 89
Thread-Pool 136, 138
ThreadPoolExecutor 138
thread-safe 148, 216, 220
Timer 220
Time-To-Live 247
TimeUnit 137
Top-Level-Container 190
transient 295
Transmission Control Protocol 230, 233
transparent 273
Transparenz 275
Transport Layer 229
Transportschicht 229
TreeModel 210
Treiber 230
tryLock 142

U
Überlastkontrolle 233
UDP 230, 232, 239
unbind 284
uncaughtException 88
Unexportieren 323
unexportObject 322
Unicast-Adresse 230
<table>
<thead>
<tr>
<th>UnicastRemoteObject</th>
<th>275, 321</th>
</tr>
</thead>
<tbody>
<tr>
<td>Universal Resource Locator</td>
<td>340</td>
</tr>
<tr>
<td>unlock</td>
<td>141</td>
</tr>
<tr>
<td>unteilbar</td>
<td>39, 148</td>
</tr>
<tr>
<td>unzuverlässig</td>
<td>230</td>
</tr>
<tr>
<td>up</td>
<td>99</td>
</tr>
<tr>
<td>URL</td>
<td>271, 340</td>
</tr>
<tr>
<td>URLConnection</td>
<td>271</td>
</tr>
<tr>
<td>UseBean-Aktion</td>
<td>388</td>
</tr>
<tr>
<td>User Datagram Protocol</td>
<td>230, 232</td>
</tr>
<tr>
<td>User Threads</td>
<td>87</td>
</tr>
</tbody>
</table>

**V**

<table>
<thead>
<tr>
<th>v</th>
<th>99</th>
</tr>
</thead>
<tbody>
<tr>
<td>valueChanged</td>
<td>189</td>
</tr>
<tr>
<td>verbindungslös</td>
<td>230, 232</td>
</tr>
<tr>
<td>verbindungsoorientiert</td>
<td>233</td>
</tr>
<tr>
<td>Verbraucher</td>
<td>67, 109</td>
</tr>
<tr>
<td>Verklemmung</td>
<td>65, 106, 120, 153</td>
</tr>
<tr>
<td>Verlust</td>
<td>230</td>
</tr>
<tr>
<td>Vermittlungsschicht</td>
<td>229</td>
</tr>
<tr>
<td>Verschlüsseln</td>
<td>255</td>
</tr>
<tr>
<td>verteilte Anwendungen</td>
<td>14</td>
</tr>
<tr>
<td>verteilte Systeme</td>
<td>14</td>
</tr>
<tr>
<td>verteiltes System</td>
<td>15, 16</td>
</tr>
</tbody>
</table>

| Verteilung          | 14       |
| Verteilungstransparenz | 274    |
| Virtualisierung     | 14       |
| volatile            | 40       |
| Vordergrund-Threads | 87       |

**W**

| wait                 | 59, 66   |
| well-known port number | 235    |
| Wertübergabe         | 291      |
| Wiki                 | 377      |
| wohlbekannte Portnummer | 235, 236 |
| World Wide Web       | 230      |
| write                | 252, 255 |
| writeInt             | 254      |
| writeObject          | 294      |
| Writer               | 252      |
| WWW                  | 230      |

**Y**

| yield                | 88       |

**Z**

| Zustandsübergangsdiagramm | 93       |
| zuverlässig              | 233      |