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

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 CompletableFutures behandelt, ohne auf alle Details dieser Konzepte einzugehen (Kapitel 3).
- Entwicklung von Anwendungen mit grafischer Benutzeroberfläche in Java (Kapitel 4): Grafische Benutzeroberflächen scheinen auf den ersten Blick nichts mit parallelen und verteilten Anwendungen zu tun zu haben. Bei näherem Hinsehen erkennt man aber durchaus Zusammenhänge. So wird zum Beispiel in diesem Buch ausführlich erläutert, welche negativen Effekte es bei naiver Programmierung für Anwendungen mit grafischer Benutzeroberfläche gibt, wenn eine länger dauernde Aktivität aufgrund einer Interaktion mit der grafischen Benutzerobefläche 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 Benutzerobefläche 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 CompletableFutures, 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: **p**arallele **u**nd **v**erteilte **A**nwendungen) 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

2 Grundlegende Synchronisations-konzepte 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 Methodenname eindeutig aus der funktionalen Schnittstelle I1 herleiten lassen:

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

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

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

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

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

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

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

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

Jetzt könnten wir zum Beispiel schreiben:

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

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

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

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

2.1.3 Einige Beispiele

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

Listing 2.3

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

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

    public void run()
    {
        for(int i = 1; i <= 100; i++)
```

```
        {
            System.out.println(myName + " (" + i + ")");
        }
    }

    public static void main(String[] args)
    {
        Loop1 t1 = new Loop1("Thread 1");
        Loop1 t2 = new Loop1("Thread 2");
        Loop1 t3 = new Loop1("Thread 3");
        t1.start();
        t2.start();
        t3.start();
    }
}
```

In diesem Beispiel werden drei zusätzliche Threads gestartet. Die dazugehörigen Thread-Objekte gehören alle derselben Klasse an, 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.

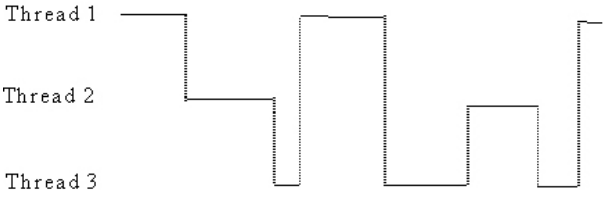

Bild 2.1
Ausführungsintervalle von drei Threads

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

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z