1 Einleitung

Computer-Nutzer dürften mit großer Wahrscheinlichkeit sowohl mit parallelen Abläufen auf ihrem eigenen Rechner als auch verteilten Anwendungen vertraut sein. So ist jeder Benutzer eines PC heutzutage gewohnt, dass z. B. gleichzeitig eine größere Video-Datei kopiert, ein Musikstück aus einer MP3-Datei abgespielt, ein Java-Programm übersetzt und ein Dokument in einem Editor oder Textverarbeitungsprogramm bearbeitet werden kann. Aufgrund der Tatsache, dass die Mehrzahl der genutzten Computer an das Internet angeschlossen ist und fast alle Menschen ein Handy nutzen, sind heute auch nahezu alle den Umgang mit verteilten Anwendungen wie der elektronischen Post, Messenger-Diensten oder dem World Wide Web gewohnt.

Dieses Buch handelt allerdings nicht von der Benutzung, sondern von der Entwicklung paralleler und verteilter Anwendungen mit Java. In diesem ersten einleitenden Kapitel werden zunächst einige wichtige Begriffe wie Parallelität, Nebenläufigkeit, Verteilung, Prozesse und Threads geklärt.

1.1 Parallelität, Nebenläufigkeit und Verteilung

Wenn mehrere Vorgänge gleichzeitig auf einem Rechner ablaufen, so sprechen wir von Parallelität oder Nebenläufigkeit (engl. concurrency). Diese Vorgänge können dabei echt gleichzeitig oder nur scheinbar gleichzeitig ablaufen: Wenn ein Rechner mehrere Prozessoren bzw. einen Mehrkernprozessor (Multicore-Prozessor) besitzt, dann ist echte Gleichzeitigkeit möglich. Man spricht in diesem Fall auch von echter Parallelität. Besitzt der Rechner aber nur einen einzigen Prozessor mit einem einzigen Kern, so wird die Gleichzeitigkeit der Abläufe nur vorgetäuscht, indem in sehr hoher Frequenz von einem Vorgang auf den nächsten umgeschaltet wird. Man spricht in diesem Fall von Pseudoparallelität oder Nebenläufigkeit. Die Begriffe Parallelität und Nebenläufigkeit werden in der Literatur nicht einheitlich verwendet: Einige Autoren verwenden den Begriff Nebenläufigkeit als Oberbegriff für echte Parallelität und Pseudoparallelität, für andere Autoren sind Nebenläufigkeit und Pseudoparallelität Synonyme. In diesem Buch wird der Einfachheit halber nicht zwischen Nebenläufigkeit und Parallelität unterschieden; mit beiden Begriffen sollen sowohl die echte als auch die Pseudoparallelität gemeint sein.

Wenn das gleichzeitige Ablaufen von Vorgängen auf mehreren Rechnern betrachtet wird, wobei die Rechner über ein Rechnernetz gekoppelt sind und darüber miteinander kommunizieren, spricht man von Verteilung (verteilte Systeme, verteilte Anwendungen).

Wir unterscheiden also, ob die Vorgänge auf einem Rechner oder auf mehreren Rechnern gleichzeitig ablaufen; im ersten Fall sprechen wir von Parallelität, im anderen Fall von Verteilung. Die Mehrzahl der Leserinnen und Leser dürfte vermutlich mit dieser Unterscheidung zufrieden sein. In manchen Fällen ist es aber gar nicht so einfach, zu entscheiden, ob ein gegebenes System einen einzigen Rechner oder eine Vielzahl von Rechnern darstellt. Betrachten Sie z. B. ein System zur Steuerung von Maschinen, wobei dieses System in einem Schaltschrank untergebracht ist, in dem sich mehrere Einschübe mit Prozessoren befinden. Handelt es sich hier um einen oder um mehrere kommunizierende Rechner? Zur Klärung dieser Frage wollen wir uns hier an die allgemein übliche Unterscheidung zwischen eng und lose gekoppelten Systemen halten: Ein eng gekoppeltes System ist ein Rechnersystem bestehend aus mehreren gekoppelten Prozessoren, wobei diese auf einen gemeinsamen Speicher (Hauptspeicher) zugreifen können. Ein lose gekoppeltes System (auch verteiltes System genannt) besteht aus mehreren gekoppelten Prozessoren ohne gemeinsamen Speicher (Hauptspeicher), die über ein Kommunikationssystem Nachrichten austauschen. Ein eng gekoppeltes System sehen wir als einen einzigen Rechner, während wir ein lose gekoppeltes System als einen Verbund mehrerer Rechner betrachten.

Parallelität und Verteilung schließen sich nicht gegenseitig aus, sondern hängen im Gegenteil eng miteinander zusammen: In einem verteilten System laufen auf jedem einzelnen Rechner mehrere Vorgänge parallel (echt parallel oder pseudoparallel) ab. Wie auch in diesem Buch noch ausführlich diskutiert wird, arbeitet ein Server im Rahmen eines Client-Server-Szenarios häufig parallel, um mehrere Clients gleichzeitig zu bedienen. Außerdem können verteilte Anwendungen, die für den Ablauf auf unterschiedlichen Rechnern vorgesehen sind, im Spezialfall auf einem einzigen Rechner parallel ausgeführt werden.

Sowohl Parallelität als auch Verteilung werden durch Hard- und Software realisiert. Bei der Software spielt das Betriebssystem eine entscheidende Rolle. Das Betriebssystem verteilt u. a. die auf einem Rechner gleichzeitig möglichen Abläufe auf die vorhandenen Prozessoren bzw. die vorhandenen Kerne des Rechners. Auf diese Art vervielfacht also das Betriebssystem die Anzahl der vorhandenen Prozessoren bzw. der vorhandenen Kerne virtuell. Diese Virtualisierung ist eines der wichtigen Prinzipien von Betriebssystemen, die auch für andere Ressourcen realisiert wird. So wird z. B. durch das Konzept des virtuellen Speichers ein größerer Hauptspeicher vorgegaukelt als tatsächlich vorhanden. Erreicht wird dies, indem immer die gerade benötigten Daten vom Hintergrundspeicher (Platte) in den Hauptspeicher transferiert werden.

■ 1.2 Programme, Prozesse und Threads

Im Zusammenhang mit Parallelität bzw. Nebenläufigkeit und Verteilung muss zwischen den Begriffen Programm, Prozess und Thread (Ausführungsfaden) unterschieden werden. Da es einen engen Zusammenhang zu den Themen Betriebssysteme, Rechner und verteilte Systeme gibt, sollen alle diese Begriffe anhand einer Metapher verdeutlicht werden:

- Ein Programm entspricht einem Rezept in einem Kochbuch. Es ist statisch. Es hat keine Wirkung, solange es nicht ausgeführt wird. Dass man von einem Rezept nicht satt wird, ist hinlänglich bekannt.
- Einen Prozess kann man sich vorstellen als eine Küche und einen Thread als einen Koch. Ein Koch kann nur in einer Küche existieren, aber nie außerhalb davon. Umgekehrt muss sich in einer Küche immer mindestens ein Koch befinden. Alle Köche gehen streng nach Rezepten vor, wobei unterschiedliche Köche nach demselben oder nach unterschiedlichen Rezepten kochen können. Jede Küche hat ihre eigenen Pfannen, Schüsseln, Herde, Waschbecken, Messer, Gewürze, Lebensmittel usw. Köche in unterschiedlichen Küchen können sich gegenseitig nicht in die Quere kommen, wohl aber die Köche in einer Küche. Diese müssen den Zugriff auf die Materialien und Geräte der Küche koordinieren.
- Ein Rechner ist in dieser Metapher ein Haus, in dem sich mehrere Küchen befinden.
- Ein Betriebssystem lässt sich mit einem Hausmeister eines Hauses vergleichen, der dafür sorgt, dass alles funktioniert (z. B. dass immer Strom für den Herd da ist). Der Hausmeister übernimmt u. a. auch die Rolle eines Boten zwischen den Küchen, um Gegenstände oder Informationen zwischen den Küchen auszutauschen. Auch kann er eine Küche durch einen Anbau vergrößern, wenn eine Küche zu klein geworden ist.

Ein verteiltes System besteht entsprechend aus mehreren solcher Häuser mit Küchen, wobei die Hausmeister der einzelnen Häuser z. B. über Telefon oder über hin- und herlaufende Boten untereinander kommunizieren können. Somit können Köche, die in unterschiedlichen Häusern arbeiten, Gegenstände oder Informationen austauschen, indem sie ihre jeweiligen Hausmeister damit beauftragen.

Diese Begriffe und ihre Beziehung sind in Bild 1.1 zusammenfassend dargestellt.

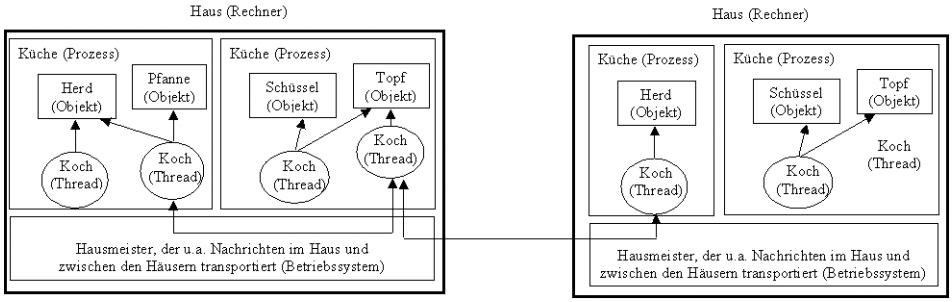

Bild 1.1 Häuser, Küchen, Köche und Hausmeister als Metapher für Rechner, Prozesse, Threads und Betriebssysteme

Am Beispiel der Programmiersprache Java und der Ausführung von Java-Programmen lässt sich diese Metapher nun auf die Welt der Informatik übertragen:

- Ein Programm (Kochrezept) ist in einer Datei abgelegt: als Quelltext in einer oder mehreren Java-Dateien und als übersetztes Programm (Byte-Code) in einer oder mehreren Class-Dateien.
- Zum Ausführen eines Programms mithilfe des Kommandos java wird eine JVM (Java Virtual Machine) gestartet. Bei jedem Erteilen des Java-Kommandos wird ein neuer Prozess

(Küche) erzeugt. Ein Prozess stellt im Wesentlichen einen Adressraum für den Programmcode und die Daten dar. Der Programmcode, der sich in einer oder mehreren Dateien befindet, wird in den Adressraum des Prozesses geladen. Es ist möglich, mehrere JVMs zu starten, sodass die entsprechenden Prozesse alle gleichzeitig existieren, wobei jeder Prozess seinen eigenen Adressraum besitzt.

- Jeder Prozess und damit auch jede JVM hat als Aktivitätsträger mindestens einen Thread (Koch). Neben den sogenannten Hintergrund-Threads, die z. B. für die Speicherbereinigung (Garbage Collection) zuständig sind, gibt es einen Thread, der die Main-Methode der im Java-Kommando angegebenen Klasse ausführt. Dieser Thread kann durch Aufruf entsprechender Methoden weitere Threads starten. Die Threads innerhalb desselben Prozesses können auf dieselben Objekte (Gegenstände in einer Küche wie Herd, Pfannen, Töpfe, Schüsseln usw.) lesend und schreibend zugreifen, nicht aber auf die Objekte, die sich in anderen Prozessen befinden.
- Das Betriebssystem (Hausmeister) verwaltet die Adressräume der Prozesse und teilt den Threads abwechselnd die vorhandenen Prozessoren bzw. den vorhandenen Kernen zu. Das Betriebssystem garantiert gemeinsam mit der Hardware, dass ein Prozess keinen Zugriff auf den Adressraum eines anderen Prozesses auf demselben Rechner besitzt. Damit sind die Prozesse eines Rechners voneinander isoliert. Zwei Prozesse auf unterschiedlichen Rechnern sind ebenfalls voneinander isoliert, da ein verteiltes System laut Definition ein lose gekoppeltes System ist, das keinen gemeinsamen Speicher hat.

Die Isolierung der Prozessadressräume kann durch Inanspruchnahme von Leistungen des Betriebssystems über Systemaufrufe in kontrollierter Weise durchbrochen werden. Damit können die Prozesse miteinander interagieren. Betriebssysteme bieten Dienste zur Synchronisation und Kommunikation zwischen Prozessen sowie zur gemeinsamen Nutzung von speziellen Speicherbereichen an. Ferner stellen Betriebssysteme Funktionen bereit, um über ein Rechnernetz Daten an Prozesse anderer Rechner zu senden oder eingetroffene Nachrichten entgegenzunehmen. Durch Systemaufrufe kann das Betriebssystem auch beauftragt werden, den Adressraum eines Prozesses zu vergrößern.

In diesem Buch geht es um zwei wesentliche Aspekte:

- Parallelität innerhalb eines Prozesses: Die Leserinnen und Leser sollen das Konzept der Parallelität innerhalb eines Prozesses aus Sicht einer Programmiererin bzw. eines Programmierers mit Java-Threads beherrschen lernen. Sie sollen erkennen, welche Probleme entstehen, wenn mehrere Threads auf dieselben Objekte zugreifen und wie diese Probleme gelöst werden können.
- Verteilung: Darüber hinaus zeigt das Buch, wie verteilte Anwendungen mit Java entwickelt werden. Wir unterscheiden dabei eigenständige Client-Server-Anwendungen und webbasierte Anwendungen. Bei eigenständigen Client-Server-Anwendungen entwickeln wir sowohl die Client- als auch die Server-Programme selbst. Client und Server kommunizieren dabei über die Socket-Schnittstelle, über RMI (Remote Method Invocation) oder indirekt über einen Vermittler. Bei webbasierten Anwendungen benutzen wir als Client einen Browser. Die Server-Seite besteht aus einem Web-Server, der durch selbst entwickelte Programme erweitert werden kann. Sowohl bei den eigenständigen Client-Server-Anwendungen als auch bei den webbasierten Anwendungen spielt die Parallelität insbesondere auf Server-Seite eine wichtige Rolle. Immer wichtiger wird die Nutzung von Cloud-Diensten durch verteilte Anwendungen. Auch dieses Thema wird behandelt.

Wir betrachten hier nicht gesondert die Parallelität, Interaktion und Synchronisation von Threads unterschiedlicher Prozesse desselben Rechners. Dies liegt vor allem daran, dass es hierzu keine speziellen Java-Klassen gibt. Dies bedeutet aber keine Einschränkung, denn alle in diesem Buch vorgestellten Kommunikationskonzepte zwischen dem Client- und Server-Prozess einer verteilten Anwendung können auch angewendet werden, wenn sich Client und Server auf demselben Rechner befinden. Das heißt: Bezüglich der Kommunikation zwischen Threads unterschiedlicher Prozesse unterscheiden wir nicht, ob sich die Prozesse auf demselben oder auf unterschiedlichen Rechnern befinden.

Die Synchronisations- und Kommunikationskonzepte, die anhand von Java-Threads innerhalb eines Prozesses vorgestellt werden, gibt es in ähnlicher Weise auch für das Zusammenspiel von Threads unterschiedlicher Prozesse auf einem Rechner. Wie schon erwähnt, gibt es zwar hierfür keine spezielle Java-Schnittstelle, aber die erlernten Konzepte wie Semaphore, Message Queues und Pipes bilden eine gute Grundlage für das Verständnis der Dienste, die ein Betriebssystem wie Linux zur Synchronisation und Kommunikation zwischen unterschiedlichen Prozessen anbietet.

2 Grundlegende Synchronisationskonzepte in Java

In diesem Kapitel geht es um die grundlegenden Synchronisationskonzepte in Java. Diese bestehen im Wesentlichen aus dem Schlüsselwort synchronized sowie den Methoden wait, notify und notifyAll der Klasse Object. Es wird erläutert, welche Wirkung synchronized, wait, notify und notifyAll haben und wie sie eingesetzt werden sollen. Außerdem spielt die Klasse Thread eine zentrale Rolle. Diese Klasse wird benötigt, um Threads zu erzeugen und zu starten.

2.1 Erzeugung und Start von Java-Threads

Wie schon im einleitenden Kapitel erläutert wurde, wird beim Start eines Java-Programms (z. B. mittels des Kommandos java) ein Prozess erzeugt, der u. a. einen Thread enthält, der die Main-Methode der angegebenen Klasse ausführt. Der Programmcode weiterer vom Anwendungsprogrammierer definierter Threads muss sich in Methoden namens *run* befinden:

```
public void run ()
{
    // Code, der in eigenem Thread ausgeführt wird
}
```

Es gibt zwei Möglichkeiten, in welcher Art von Klasse diese Run-Methode definiert wird.

2.1.1 Ableiten der Klasse Thread

Die erste Möglichkeit besteht darin, aus der Klasse *Thread*, die bereits eine leere Run-Methode besitzt, eine neue Klasse abzuleiten und darin die Run-Methode zu überschreiben. Die Klasse Thread ist (wie String) eine Klasse des Package java.lang und kann deshalb ohne Import-Anweisung in jedem Java-Programm verwendet werden. Hat man eine derartige Klasse definiert, so muss noch ein Objekt dieser Klasse erzeugt und dieses Objekt (das ja ein Thread ist, da es von Thread abgeleitet wurde) mit der *Start-Methode* gestartet werden. Das Programm in Listing 2.1 zeigt dies anhand eines Beispiels.

Listing 2.1

```
public class MyThread extends Thread
{
    public void run()
    {
        System.out.println("Hallo Welt");
    }
    public static void main(String[] args)
    {
        MyThread t = new MyThread();
        t.start();
    }
}
```

An diesem ersten Programmbeispiel mag auf den ersten Blick verwirrend sein, dass in der Klasse MyThread zwar eine Run-Methode definiert wird, dass aber in der Main-Methode eine Methode namens *start* auf das Objekt der Klasse MyThread angewendet wird. Die Methode start ist in der Klasse Thread definiert und wird somit auf die Klasse MyThread vererbt.

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

Natürlich könnte man statt start auch die Methode run auf das erzeugte Objekt anwenden. Der Benutzer würde keinen Unterschied zwischen den beiden Programmen feststellen können, denn in beiden Fällen wird „Hallo Welt“ ausgegeben. Allerdings ist der Ablauf in beiden Fällen deutlich verschieden: In der Metapher der Küchen und Köche passiert bei dem oben angegebenen Programm Folgendes: Der bereits vorhandene Koch, der nach dem Rezept der Main-Methode kocht, erzeugt einen neuen Koch und erweckt diesen mithilfe der Start-Methode zum Leben. Dieser neue Koch geht nach dem Rezept der entsprechenden Run-Methode vor und gibt „Hallo Welt“ aus. Würde dagegen der Aufruf der Start-Methode durch einen Aufruf der Run-Methode in obigem Programm ersetzt, so wäre dies ein gewöhnlicher Methodenaufruf, wie Sie das aus der bisherigen sequenziellen Programmierung bereits kennen. Die Ausgabe „Hallo Welt“ erfolgt also in diesem Fall durch den Thread, der die Main-Methode ausführt, und nicht durch einen neuen Thread. In der Metapher der Küchen und Köche könnte man einen Methodenaufruf so sehen wie einen Hinweis in einem Kochbuch, in dem in einem Rezept die Anweisung „Hefeteig zubereiten“ (s. Seite 456) steht. Derselbe Koch, der diese Anweisung liest, würde dann auf die Seite 456 blättern, die dort stehenden Anweisungen befolgen und anschließend zum ursprünglichen Rezept zurückkehren.

Dieses kleine, nur wenige Zeilen umfassende Beispielprogramm enthält noch ein weiteres Verständnisproblem für viele Neulinge: Warum muss ein Thread-Objekt (genauer: ein Objekt der aus Thread abgeleiteten Klasse MyThread) mit new erzeugt und warum muss dieses dann noch zusätzlich mit der Start-Methode gestartet werden? Diese Verständnisschwierigkeit kann beseitigt werden, indem man sich klar macht, dass es einen Unterschied zwischen einem Thread-Objekt und dem eigentlichen Thread im Sinne einer selbst-

ständig ablaufenden Aktivität gibt. In unserer Küchen-Köche-Metapher entspricht das Thread-Objekt dem Körper eines Kochs. Ein solcher Körper wird mit new erzeugt. Man kann bei diesem Objekt wie bei anderen Objekten üblich Attribute lesen und verändern, also z. B. Name, Personalnummer und Schuhgröße des Kochs. Dieses Objekt ist aber leblos wie andere Objekte bei der sequenziellen Programmierung auch. Erst durch Aufruf der Start-Methode wird dem Koch der Odem eingehaucht; er beginnt zu atmen und eigenständig gemäß seines Run-Rezepts zu handeln. Dieses Leben des Kochs ist als Objekt im Programm nicht repräsentiert, sondern lediglich der Körper des Kochs. Das Leben des Kochs ist beim Ablauf des Programms durch die vorhandene Aktivität zu erkennen.

Wie im richtigen Leben kann auf ein Thread-Objekt nur ein einziges Mal die Start-Methode angewendet werden. Wenn mehrere gleichartige Threads gestartet werden sollen, dann müssen entsprechend viele Thread-Objekte erzeugt werden (s. Abschnitt 2.1.3).

Ist das Run-Rezept eines Kochs abgehandelt (d. h. ist die Run-Methode zu Ende), so stirbt dieser Koch wieder (der Thread ist als Aktivität nicht mehr vorhanden). Damit muss aber der Körper des Kochs nicht auch verschwinden, sondern dieser kann weiter existieren (falls es noch Referenzen auf das entsprechende Thread-Objekt gibt, ist dieses Objekt noch vorhanden; die verbleibenden Threads können weitere Methoden auf dieses Objekt anwenden).

2.1.2 Implementieren der Schnittstelle Runnable

Falls sich im Rahmen eines größeren Programms die Run-Methode in einer Klasse befinden soll, die bereits aus einer anderen Klasse abgeleitet ist, so kann diese Klasse nicht auch zusätzlich aus Thread abgeleitet werden, da es in Java keine Mehrfachvererbung für Klassen gibt. Als Ersatz für die Mehrfachvererbung existieren in Java Schnittstellen (Interfaces). Es gibt eine Schnittstelle namens *Runnable* (wie die Klasse Thread im Package java.lang), die nur die schon oben vorgestellte Run-Methode enthält.

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

Will man nun die Run-Methode in einer nicht aus Thread abgeleiteten Klasse definieren, so sollte diese Klasse stattdessen die Schnittstelle Runnable implementieren. Wenn ein Objekt einer solchen Klasse, die diese Schnittstelle implementiert, dem Thread-Konstruktor als Parameter übergeben wird, dann wird die Run-Methode dieses Objekts nach dem Starten des Threads ausgeführt. Das Programm in Listing 2.2 zeigt diese Vorgehensweise anhand eines Beispiels.

Listing 2.2

```
public class SomethingToRun implements Runnable
{
    public void run()
    {
        System.out.println("Hallo Welt");
    }
```

```
    public static void main(String[] args)
    {
        SomethingToRun runner = new SomethingToRun();
        Thread t = new Thread(runner);
        t.start();
    }
}
```

Voraussetzung für die korrekte Übersetzung beider Beispielprogramme ist, dass die Klasse Thread u. a. folgende Konstruktoren besitzen muss:

```
public class Thread
{
    public Thread() {...}
    public Thread(Runnable r) {...}
   ...
}
```

Der zweite Konstruktor ist offenbar für das zweite Beispiel nötig. Die Nutzung des ersten Konstruktors im ersten Beispiel ist weniger offensichtlich. Da in der Klasse MyThread kein Konstruktor definiert wurde, ist automatisch der folgende Standardkonstruktor vorhanden:

```
public MyThread()
{
    super();
}
```

Der Super-Aufruf bezieht sich auf den parameterlosen Konstruktor der Basisklasse Thread. Einen solchen muss es geben, damit das Programm übersetzbar ist.

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

Auch wenn wie oben beschrieben ein Thread nur einmal gestartet werden kann, kann hier dennoch dasselbe Runnable-Objekt mehrmals als Parameter an Thread-Konstruktoren übergeben werden. Es wird ja jedes Mal ein neues Thread-Objekt erzeugt, das dann nur einmal gestartet wird. Unter Umständen kann dies aber zu Synchronisationsproblemen führen (s. Abschnitt 2.2 und Abschnitt 2.3).

Seit Java 8 gibt es sogenannte *Lambda-Ausdrücke*. Die Definition der Klasse SomethingToRun in Listing 2.2, welche die Schnittstelle Runnable implementiert, sowie das Erzeugen eines Objekts dieser Klasse kann mit einem Lambda-Ausdruck durch eine einzige Anweisung ersetzt werden:

```
Runnable runner = () -> System.out.println("Hallo Welt");
```

Wenn man das Runnable-Objekt, das man dem Konstruktor von Thread übergibt, in keine lokale Variable speichern möchte, dann kann man die Thread-Erzeugung noch kürzer auch so schreiben:

```
Thread t = new Thread(() -> System.out.println("Hallo Welt"));
```

Und wenn man auf die lokale Thread-Variable t auch noch verzichten möchte, dann schrumpft der Inhalt der Main-Methode auf diese eine Zeile zusammen:

```
new Thread(() -> System.out.println("Hallo Welt")).start();
```

Lambda-Ausdrücke können im Programmcode immer dort angegeben werden, wo ein Objekt vom Typ einer sogenannten *funktionalen Schnittstelle* erwartet wird. Eine funktionale Schnittstelle (Functional Interface) ist eine Schnittstelle mit einer einzigen Methode (genauer müsste man sagen: eine Schnittstelle mit einer einzigen *abstrakten* Methode, denn seit Java 8 können Schnittstellen auch nicht-abstrakte Methoden, sogenannte Default-Methoden, besitzen, für die in der Schnittstelle eine Implementierung angegeben ist). Die Schnittstelle Runnable ist ganz offensichtlich eine funktionale Schnittstelle. Also kann auf der rechten Seite einer Zuweisung an eine Runnable-Variable (Runnable runner = ...) oder als Parameterwert eines Thread-Konstruktors mit Runnable-Parameter ein Lambda-Ausdruck eingesetzt werden.

Allgemein hat ein Lambda-Ausdruck folgende Form:

```
Parameterliste -> Code
```

Der Name der implementierten Methode der Schnittstelle muss (und darf auch) bei einem Lambda-Ausdruck nicht angegeben werden; der Typ des Lambda-Ausdrucks ist nämlich eine funktionale Schnittstelle mit einer einzigen abstrakten Methode, und genau diese Methode wird implementiert. Für die Parameter müssen Bezeichner und optional der jeweilige Typ angegeben werden. Sie können im Code-Teil verwendet werden. Betrachten wir dazu z. B. folgende funktionale Schnittstelle I1:

```
public interface I1
{
    public void m(String s, boolean b);
}
```

Nun könnte man beispielsweise schreiben:

```
I1 i11 = (String s, boolean b) -> System.out.println(s + ", " + b);
```

Oder auch kürzer durch Weglassen der Parametertypen, die sich wie der Methodenname eindeutig aus der funktionalen Schnittstelle I1 herleiten lassen:

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

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

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

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

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