Inhalt

15 Parallele Programmierung

In diesem Kapitel sollen Möglichkeiten der **parallelen Programmierung** betrachtet werden. Eine *echte* parallele Ausführung kann natürlich nur auf *einem* Rechner, der mit mehreren Prozessoren ausgestattet ist oder auf *mehreren* miteinander vernetzten Rechnern realisiert werden. Ansonsten kann die **Nebenläufigkeit** von Programmen nur *simuliert* werden, indem der Prozessor während der Laufzeit zwischen diesen wechselt. Für den Benutzer entsteht der *Eindruck* einer gleichzeitigen Ausführung. Man spricht von **Quasiparallelität**.

In Java wird ein nebenläufiger Programmfluss durch ein Objekt der Klasse **`Thread`** realisiert. Unbewusst haben wir bereits mit Threads (Fäden) gearbeitet. Eine Anwendung startet mit einem **`main`**-Thread. In Programmen mit grafischer Oberfläche wird ein weiterer Thread erzeugt, der sich automatisch um alle Ereignisse im Zusammenhang mit den verschiedenen Komponenten der Oberfläche kümmert.

15.1 Die Klasse `java.lang.Thread`

Die Klasse **`Thread`** implementiert ein Interface **`Runnable`**. Dieses Interface hat nur eine Methode, die Methode **`run`**. Ein spezieller Mechanismus der Klasse **`Thread`** wird durch eine Methode **`start`** eingeleitet. Er sorgt dafür, dass die Methode **`run`** nebenläufig ausgeführt werden kann. Parallele Programmteile werden folglich in der **`run`**-Methode eines Threads implementiert. Die **`start`**-Methode unterbricht das laufende Programm, stößt die **`run`**-Methode an und setzt die Ausführung unmittelbar danach fort.

Konstruktor

- **`Thread()`**, erzeugt **`Thread`**-Objekt.

Klassenvariable der Klasse `Thread`

`static int MIN_PRIORITY`	geringste Priorität eines Threads
`static int MAX_PRIORITY`	höchste Priorität eines Threads
`static int NORM_PRIORITY`	default Priorität eines Threads

Methoden/Klassenmethoden der Klasse `Thread` (Auswahl)

Name	Parameter-anzahl	Parameter-typ	Ergebnis-typ	Beschreibung
`run`	0			fasst nebenläufig ausführbare Anweisungen zusammen
`start`	0			stößt **`run`**-Methode nebenläufig an
`isAlive`	0		`boolean`	Thread gestartet, nicht beendet
`interrupt`	0			setzt Abbruchflag
`isInterrupted`	0		`boolean`	gibt Abbruchflag zurück
`getPriority`	0		`int`	gibt Priorität zurück
`setPriority`	1	`int`		setzt Priorität
`static sleep`	1	`long`		Thread unterbricht Ausführung (Millisekunden)

15.2 Beispiel „0-1-Printer“ als `Thread`

Zwei Druck-Methoden sollen nebenläufig ausgeführt werden, eine druckt nur Nullen, die andere nur Einsen.
Eine Klasse **`PrinterThread`** ermöglicht den Druck eines Zeichens und zwar 40 mal hintereinander. Um den Druck nebenläufig zu ermöglichen, wird die Klasse von der Klasse **`Thread`** abgeleitet. Dem Konstruktor der Klasse wird das zu druckende Zeichen übergeben. Den Druck übernimmt die Methode **`run`**.

PrinterThread.java

```
// PrinterThread.java                                    MM 2015

/**
 * Nebenläufiger Druck einer Zahl, als Thread.
 */
public class PrinterThread
  extends Thread
{
/**
 * Druckzahl.
 */
  private int n;

/**
 * Konstruktor, setzt Druckzahl.
 * @param n Druckzahl
 */
  public PrinterThread( int n)
  {
    this.n = n;
  }

/**
 * Druckt Zeichen 40x.
 */
  public void run()
  {
    for( int i = 0; i < 40; i++)
      System.out.print( n);
  }

/* ------------------------------------------------------ */
/**
 * Test, sequentiell und parallel.
 */
  public static void main( String[] args)
  {
    Thread nullPrinter = new PrinterThread( 0);
    Thread einsPrinter = new PrinterThread( 1);

// sequentiell
```

```
      nullPrinter.run();
      einsPrinter.run();

      System.out.println();

  // parallel
      nullPrinter.start();
      einsPrinter.start();
    }
  }
```

Im Testprogramm werden zwei Threads der Klasse **PrinterThread** konstruiert, eines zum Drucken von Nullen und eines zum Drucken von Einsen. Zunächst starten wir die Methoden **run** direkt und dann über die Methode **start**. Während im ersten Fall der Eins-Printers erst dann gestartet wird, wenn der Null-Printers den Druck abgeschlossen hat, erfolgt im zweiten Fall ein mehrfacher Wechsel der Ausführung beider.

```
00000000000000000000000000000000000000000011111111111111111111111111111111111111
00000000000000000111111111111111111111111111111111100000000000000000000000111111
```

Das Druckergebnis im *quasiparallelen* Druck ist abhängig von den *Zeitscheiben*, welche dem Thread im Verlauf der Ausführung zur Verfügung stehen. Ein **Scheduler** (Disponent) übernimmt die Verwaltung der Nebenläufigkeit unter Berücksichtigung der Prioritäten der aktiven Threads. Dessen Verhalten ist stark vom jeweiligen Betriebssystem abhängig.

15.3 Das Interface `java.lang.Runnable`

In einer zweiten Variante zur Herstellung eines Threads implementiert man zunächst in einer eigenen Klasse das **Runnable**-Interface, d.h. dessen abstrakte Methode **run**, und übergibt ein Objekt dieser Klasse einem Thread. Der **start**-Aufruf eines **Thread**-Objekts führt dann die **run**-Methode des erhaltenen **Runnable**-Objekts aus.

Konstruktor

- **Thread(Runnable)**,
  erzeugt **Thread**-Objekt, welches ein **Runnable**-Objekt steuert.

run-Methode des Interface Runnable

Name	Parameter-anzahl	Parameter-typ	Ergebnis-typ	Beschreibung
run	0			fasst nebenläufig ausführbare Anweisungen zusammen, kann durch eine **start**-Methode eines **Thread**-Objekt angestoßen werden

15.4 Beispiel „0-1-Printer“ mit `Runnable`

Das letzte Beispiel wird noch einmal aufgegriffen, aber jetzt mit dem **`Runnable`**-Interface realisiert.
Eine kleine Spielerei soll ein etwas „bewegteres“ Bitmuster erzeugen. Dazu lassen wir einen Thread bei seiner Ausführung eine zufällige Zeit schlafen. Da während dieser Zeit u. U. eine Aufforderung zum Abbruch des Threads erfolgen kann, wirft die Methode **`sleep`** ein **`InterruptedException`**, welches abgefangen werden muss.

PrinterRunnable.java

```
// PrinterRunnable.java                                MM 2015

/**
 * Nebenläufiger Druck einer Zahl, als Runnable.
 */
public class PrinterRunnable
  implements Runnable
{
/**
 * Druckzahl.
 */
  private int n;

/**
 * Konstruktor, setzt Druckzahl.
 * @param n Druckzahl
 */
  public PrinterRunnable( int n)
  {
    this.n = n;
  }

/**
 * Druckt Zahl 40x, mit Druckunterbrechung.
 */
  public void run()
  {
    try
    {
      for( int i = 0; i < 40; i++)
      {
        System.out.print( n);
        Thread.sleep((int)( Math.random() * 100));
      }
    }
    catch( InterruptedException e)
    {
      // tue nichts
    }
  }
```

```
/* ------------------------------------------------------ */
/**
 * Test.
 */
  public static void main( String[] args)
  {
    Thread nullPrinter
    = new Thread( new PrinterRunnable( 0));
    Thread einsPrinter
    = new Thread( new PrinterRunnable( 1));

    nullPrinter.start();
    einsPrinter.start();
  }
}
```

Jetzt erhalten wir ein Bitmuster ähnlich dem folgenden Ausdruck:

```
01110111000011010101001001010110101001011010110100101010101011100101011011010000
```

15.5 Zusammenfassung

Prinzipiell gibt es somit mindestens zwei Möglichkeiten, eigene Threads zu erzeugen:

- Man schreibt eine Klasse, die von der Klasse **`Thread`** direkt abgeleitet wurde und deren Objekte damit selbst Threads darstellen.

- Man schreibt eine Klasse, die das Interface **`Runnable`** implementiert und deren Objekte später von einem Thread gesteuert werden.

Die zweite Variante wird insbesondere dann benötigt, wenn die eigene Klasse bereits von einer anderen Klasse abgeleitet wurde.