
PEARSON
Education
MASTER
CLASS
Karsten Samaschke
JAVA 6
Einstieg für Anspruchsvolle
inkl. Lösungen
und Lernkontrolle
CD
ADDISON-WESLEY
[in Kooperation mit]
PEARSON
Studium

6

Threads

Kennen Sie das? In Ihrer Applikation haben Sie eine Aufgabe zu erledigen, die das Programm zeitlich in Anspruch nimmt und somit dessen Ressourcen frisst – aber eigentlich nicht zu den Kernaufgaben des Programms gehört? Oder Ihr Programm muss komplexe Berechnungen ausführen, die es für Minuten blockieren würden?

Dann suchen Sie nach Wegen, derartige Bestandteile des Programms auszulagern, und zwar so, dass sie das Programm nicht mehr stören und dennoch Bestandteil des Programms bleiben. Die Lösung für dieses Problem ist *Threading*.

Threading bedeutet, dass das Programm einen Teil von sich eigenständig laufen lässt und damit Zeit bekommt, sich um andere Dinge zu kümmern. Kommt Ihnen bekannt vor? Kein Wunder: Auf Betriebssystemebene nennt sich das auch Multitasking.

Threads (also die Komponenten, die eigenständig ausgeführt werden sollen) eignen sich ideal für alles, was potenziell viel Rechenzeit in Anspruch nehmen und dabei kontinuierlich ausgeführt werden soll.

6.1 Einen Thread erstellen

Threads sind grundsätzlich normale Klassen, die die Schnittstelle `Runnable` implementieren oder von der Basisklasse `Thread` erben.

Die Schnittstelle `Runnable` erfordert, dass eine Methode `run()` in der Klasse vorhanden ist.

6.1.1 Eine threadingfähige Klasse

Ein einfaches Beispiel für eine threadingfähige Klasse, die die Schnittstelle `Runnable` implementiert, könnte so aussehen:

Listing 6.1
Die Klasse SimpleThread implementiert die Schnittstelle Runnable und kann als eigenständiger Thread laufen.

```
import java.util.Calendar;
import java.text.DateFormat;

/**
 * Einfache Runnable-Implementierung
 */
public class SimpleThread implements Runnable
{
   /**
    * run()-Methode, die vom Interface definiert wird
    */
   public void run()
   {
      // Endlosschleife, der Thread läuft potentiell ewig
      while(true)
      {
         // Anzeigen einer Nachricht
         displayMessage();

         // Pausieren des Threads
         try
         {
            Thread.sleep(5000);
         }
         catch (InterruptedException e) { }
      }
   }

   /**
    * Ausgabe der Nachricht
    */
   private void displayMessage()
   {
      // Aktuelle Zeit ausgeben
      String date = DateFormat.getTimeInstance().format(
         Calendar.getInstance().getTime());
      System.out.println(
         String.format("%s: Still running (%s)!",
         "SimpleThread", date));
   }
}
```

Im Kopf der Klassendefinition geben wir mithilfe des Schlüsselworts `implements` an, dass das Interface `Runnable` implementiert werden soll. Dies erfordert dann in der Konsequenz, dass wir eine Methode `run()` definieren, die über keine Parameter verfügt.

In der `run()`-Methode machen wir nun der Einfachheit halber nichts weiter, als alle fünf Sekunden die Meldung *Still running* samt aktueller Uhrzeit auszugeben.

Zu diesem Zweck beinhaltet sie eine Endlosschleife, die in Form einer `while`-Schleife ausgeführt worden ist. Wie wir wissen, erwartet eine `while`-Schleife immer eine Bedingung, die als Ergebnis `true` haben muss, damit die Schleife durchlaufen werden kann. Wenn die Bedingung selbst `true` ist, haben wir eine Schleife, die nicht abbrechen wird – also den typischen Fall einer Endlosschleife.

Innerhalb dieser Endlosschleife rufen wir zuerst die Methode `displayMessage()` auf, die mithilfe eines `Calendar`-Objekts und einer `DateFormat`-Instanz die aktuelle Uhrzeit anhand der Benutzereinstellungen formatiert

und zusammen mit dem Namen der Klasse ausgibt. Diese `DateFormat`-Instanz erhalten wir, indem wir die statische Methode `getTimeInstance()` der `DateFormat`-Klasse aufrufen, die uns eine Instanz mit den im System hinterlegten Formatierungsinformationen zurückgibt.

Zurück in der Methode `run()`: Hier lassen wir nun den aktuellen Thread für fünf Sekunden pausieren, indem wir die entsprechende Anzahl an Millisekunden an die Methode `Thread.sleep()` übergeben.

Achtung

Wenn Sie einen Thread per `Thread.sleep()` pausieren lassen, müssen Sie dies innerhalb eines `try-catch`-Blocks machen. Dieser `try-catch`-Block muss eine `InterruptedException` abfangen können, da diese von `Thread.sleep()` geworfen wird, wenn der aktuelle Thread abgebrochen worden ist. Die Ausnahme müssen Sie dabei in der Regel nicht weiter behandeln – der `catch`-Block kann also leer bleiben, es sei denn, Sie müssten bestimmte Aufräumarbeiten vornehmen.

6.1.2 Einen Thread starten

Um einen Thread zu starten, müssen wir zunächst eine Referenz auf ein `Thread`-Objekt erzeugen. Falls wir eine Klasse verwenden, die nicht direkt von `Thread` erbt, sondern stattdessen die Schnittstelle `Runnable` implementiert, müssen wir diese bei der Instanzierung der `Thread`-Klasse als Parameter übergeben. Anschließend kann der Thread mithilfe seiner Methode `start()` aktiviert werden.

Umgesetzt in einer Klasse, könnte dies so aussehen:

Listing 6.2
Starten eines Thread

```
import java.text.DateFormat;
import java.util.Calendar;

public class CreateThread
{

   public static void main(String args[])
   {
      // Thread definieren und instanzieren
      Thread thread;
      SimpleThread instance = new SimpleThread();

      // Thread starten
      thread = new Thread(instance);
      thread.start();

      // Meldung ausgeben
      while(true)
      {
         displayMessage();
         try
         {
            thread.sleep(3500);
         }
         catch (InterruptedException e) { }
      }
   }
```

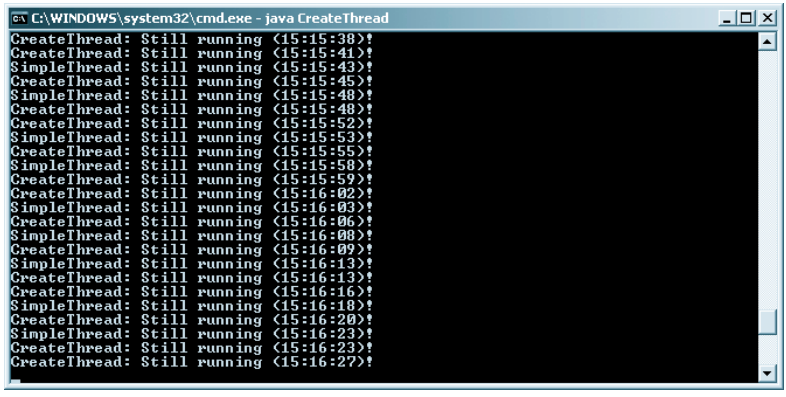
```
    /**
     * Ausgabe der Meldung
     */
    private static void displayMessage()
    {
        // Aktuelle Zeit ausgeben
        String date = DateFormat.getTimeInstance().format(
            Calendar.getInstance().getTime());
        System.out.println(String.format(
            "%s: Still running (%s)!", "CreateThread", date));
    }
}
```

Innerhalb der statischen Methode `main()` deklarieren wir hier zunächst eine Instanz der *Thread*-Klasse, weisen ihr aber noch keine Instanz zu. Dies geschieht erst, nachdem wir die Klasse, die im Thread laufen soll, mithilfe des Schlüsselworts *new* instanziert haben. Das eigentliche Starten des Thread geschieht dann wie bereits erwähnt mit dessen Methode `start()`.

Nachdem wir den neuen Thread gestartet haben, lassen wir auch in der Methode `main()` regelmäßig eine Statusinformation ausgeben – im Unterschied zur Klasse *SimpleThread* geschieht dies hier im Abstand von dreieinhalb Sekunden (3500 Millisekunden), was wir mithilfe von `Thread.sleep()` erreichen.

Wenn Sie die Klasse ausführen, sollten Sie eine Ausgabe ähnlich dieser erhalten:

Abbildung 6.1
Beide Threads laufen parallel.



Sie stellen sicherlich erfreut fest, dass die beiden Threads parallel laufen – beide warten völlig unabhängig voneinander einen gewissen Zeitraum, bevor sie erneut ihre Ausgaben tätigen.

6.2 Timer statt Thread

Wenn Sie Aufgaben haben, die wiederholt in regelmäßigen Zeitabständen ausgeführt werden sollen, müssen Sie nicht zwingend auf die `Thread`-Ebene herabsteigen. Der `Timer`, der seit Java 1.3 zur Verfügung steht, erledigt genau diese Aufgabe.

Damit eine Klasse vom Timer regelmäßig ausgeführt werden kann, muss sie von der Klasse `TimerTask` erben und deren `run()`-Methode überschreiben. Dies könnte für eine Klasse, die eine Funktionalität analog zu obigem Beispiel implementieren soll, etwa so aussehen:

```
import java.util.TimerTask;
import java.util.Calendar;
import java.text.DateFormat;

public class SimpleTimerTask extends TimerTask
{

   public void run()
   {
      // Aktuelle Uhrzeit bestimmen und ausgeben
      String date = DateFormat.getTimeInstance().format(
         Calendar.getInstance().getTime());
      System.out.println(String.format(
         "%s: Still running (%s)!",
         "SimpleTimerTask", date));
   }
}
```

Die Klasse `SimpleTimerTask` erbt von der Basisklasse `TimerTask`, was wir mithilfe des `extends`-Schlüsselworts angeben. Sie überschreibt die Methode `run()` der `TimerTask`-Klasse und legt dabei den Code fest, der während der `Timer`-Laufzeit ausgeführt werden soll.

Dieser Code ist dabei bewusst simpel gehalten: Bei jedem Aufruf von `run()` durch den `Timer` ermitteln wir die aktuelle Uhrzeit anhand der Benutzereinstellungen im System und geben sie zusammen mit einer kleinen Hinweismeldung aus.

Um den `TimerTask` auszuführen, muss er eingebunden werden. Dies geschieht in diesem Fall in einer externen Klasse, die eine neue `Timer`-Instanz erzeugt und mithilfe von deren `schedule()`-Methode festlegt, wann und in welchen Abständen unsere `SimpleTimerTask`-Klasse ausgeführt wird:

Listing 6.3
In der Klasse CreateTimer wird der Timer gestartet

```
import java.text.DateFormat;
import java.util.Calendar;
import java.util.Timer;
import java.util.TimerTask;

public class CreateTimer
{

   public static void main(String args[])
   {
       // Timer-Instanz anlegen
      Timer timer = new Timer();
      SimpleTimerTask task = new SimpleTimerTask();

      // Ausführungszeiten festlegen
      long delay = 0;
      long period = 5000;
      timer.schedule(task, delay, period);

      // Ausführen der Applikation
      while(true)
      {
```

Listing 6.3 (Forts.)
In der Klasse CreateTimer wird der Timer gestartet

```
            displayMessage();
            try
            {
               Thread.sleep(3500);
            }
            catch (InterruptedException e) { }
         }
      }

      private static void displayMessage()
      {
         String date = DateFormat.getTimeInstance().format(
            Calendar.getInstance().getTime());
         System.out.println(String.format(
            "%s: Still running (%s)!", "CreateTimer", date));
      }
   }
```

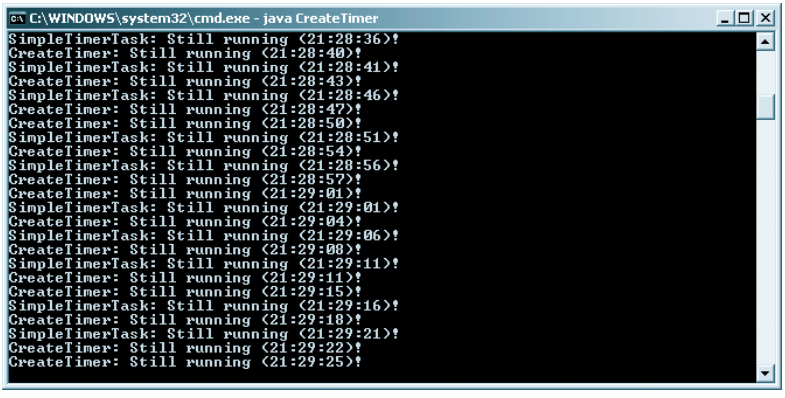
Innerhalb der statischen Methode `main()` erzeugen wir mithilfe des Schlüsselworts `new` eine `Timer`- und eine `TimerTask`-Instanz. Die `TimerTask`-Instanz ist dabei unsere Klasse `SimpleTimerTask` – wir arbeiten hier intern aber mit der Basisklasse, da wir keinerlei weiterführende Funktionalität definiert haben.

Der Methode `schedule()` der `Timer`-Instanz werden nun als Parameter unsere `SimpleTimerTask`-Instanz, die Verzögerung bis zur ersten Ausführung des `Timer` in Millisekunden und die Wiederholverzögerung vor einer erneuten Ausführung in Millisekunden als Parameter übergeben. In unserem Beispiel ist dies eine Ausführungsverzögerung von 0 Millisekunden und eine Wiederholverzögerung von 5000 Millisekunden – also fünf Sekunden.

Der Rest der Methode `main()` ist identisch mit dem vorherigen Beispiel – wir geben hier zur Kontrolle alle dreieinhalb Sekunden einen Text aus, so dass wir sehen, dass die Methode weiterhin ausgeführt wird. Diese Verzögerung erreichen wir aber nicht, indem wir einen weiteren `Timer` einsetzen, sondern den aktuellen Thread mithilfe von `Thread.sleep()` pausieren lassen.

Wenn Sie die Klasse *CreateTimer* ausführen lassen, werden Sie folgende Ausgabe erhalten:

Abbildung 6.2
Auch per Timer können Klassen in anderen Threads laufen.



Tipp

Der `Timer` empfiehlt sich also immer dann für einen Einsatz, wenn Sie wiederholt und vor allem regelmäßig Code-Fragmente ausführen wollen. Für einen Einsatz in Szenarien, in denen Sie Code einmalig auslagern und unabhängig extern laufen lassen wollen, empfiehlt sich stattdessen nach wie vor der Einsatz von Threads.

6.3 Priorität von Threads setzen

Threads können nicht nur parallel laufen. Sie können zusätzlich auch mit einer Priorität versehen werden. Die Priorität von Threads kommt dann zum Tragen, wenn verschiedene Threads zeitgleich ausgeführt werden sollen – Java entscheidet dann anhand der Priorität des Thread, welchen es tatsächlich ausführen soll und in welcher Reihenfolge weitere Threads ablaufen sollen.

Wenn mehrere Threads mit der gleichen Priorität auf Ausführung warten, entscheidet Java, welcher Thread tatsächlich als Erstes zur Ausführung kommt. Die weiteren Threads werden anschließend einer nach dem anderen abgearbeitet – man spricht in diesem Fall von einem *Round-Robin*-Verfahren, bei dem die Last gleichmäßig verteilt wird.

Die Priorität von Threads setzen Sie mithilfe der Methode `setPriority()` einer `Thread`-Instanz. Die Methode nimmt einen `Integer`-Wert entgegen, der zwischen `MIN_PRIORITY` und `MAX_PRIORITY` liegen muss. `MIN_PRIORITY` kennzeichnet die geringste Priorität, `MAX_PRIORITY` steht für die höchste mögliche Priorität eines Thread. Beide konstante Variablen sind in der Klasse `Thread` definiert.

Info

Wenn Sie einen neuen Thread ohne explizite Priorität starten, erbt dieser die Priorität von dem erzeugenden Thread. Dies bedeutet, dass ein Thread, der von einem Thread mit `MAX_PRIORITY` erzeugt worden ist, selber auch diese Priorität besitzt.

Sehen wir uns ein Beispiel für die Priorität von Threads an:

Listing 6.4
Priorisierung von Threads

```
public class PriorityThread implements Runnable
{

   private String name;

   // Hält die Priorität
   private int priority;

   public String getName()
   {
      return name;
   }
```

Listing 6.4 (Forts.)
Priorisierung von Threads

```
    public void setName(String name)
    {
       this.name = name;
    }

    /**
     * Gibt die Priorität zurück
     */
    public int getPriority()
    {
       return priority;
    }

    /**
     * Setzt die Priorität
     */
    public void setPriority(int priority)
    {
       this.priority = priority;
    }

    public void run()
    {
       System.out.println(
          String.format(
             "Thread %s with priority %d has executed!",
             this.getName(), this.getPriority()));
    }

    public static void main(String args[])
    {
       // Instanz erzeugen und Prioritäten setzen
       PriorityThread minThread = new PriorityThread();
       minThread.setName("minPrio");
       minThread.setPriority(Thread.MIN_PRIORITY);

       // Instanz erzeugen und Prioritäten setzen
       PriorityThread maxThread = new PriorityThread();
       maxThread.setName("maxPrio");
       maxThread.setPriority(Thread.MAX_PRIORITY);

      // Threads erzeugen
        Thread minPrio = new Thread(minThread);
       Thread maxPrio = new Thread(maxThread);

       // Thread-Informationen setzen
       minPrio.setName(minThread.getName());
       minPrio.setPriority(minThread.getPriority());
       maxPrio.setName(maxThread.getName());
       maxPrio.setPriority(maxThread.getPriority());

       // Threads ausführen
       minPrio.start();
       maxPrio.start();
    }
}
```

Die Klasse `PriorityThread` implementiert das Interface `Runnable` mit dessen Methode `run()` – wobei diese Methode sehr simpel gehalten ist, denn sie gibt einfach nur aus, dass der Thread beendet worden ist.

Innerhalb der statischen Methode `main()` werden mithilfe des Schlüsselworts `new` zwei neue `PriorityThread`-Instanzen erzeugt: `minThread` und `maxThread`. Beiden wird mithilfe ihrer Methode `setName()` ein Name zuge-

wiesen – die Instanz, die die geringere Priorität haben soll, erhält den Namen `minPrio`, die andere heißt `maxPrio`.

Entsprechend der Namen werden auch die Werte für die Prioritäten verteilt – `minThread` erhält die Priorität `MIN_PRIORITY`, `maxThread` wird der Wert von `MAX_PRIORITY` zugewiesen. Die eigentliche Zuweisung erfolgt mithilfe der Methode `setPriority()` der jeweiligen `Thread`-Instanz. Beachten Sie, dass wir in diesem Beispiel noch keine wirklichen Thread-Prioritäten verteilt haben – wir haben die Werte zunächst nur unseren `PriorityThread`-Instanzen zugewiesen, damit wir sie später ausgeben können.

Nun erzeugen wir die beiden Thread-Instanzen `minPrio` und `maxPrio`, denen wir die Prioritätswerte aus den beiden `PriorityThread`-Instanzen zuweisen. Dies bedeutet, dass `minPrio` mit einer geringeren Priorität als `maxPrio` laufen wird.

Um dies zu beweisen, werden die beiden Threads am Ende mithilfe ihrer `start()`-Methoden gestartet.

Beachten Sie, dass wir zunächst `minPrio` und erst anschließend `maxPrio` starten – denn die Ausgabe ist interessant:

Thread maxPrio with priority 10 has executed!

Thread minPrio with priority 1 has executed!

Offensichtlich führt die höhere Priorisierung dazu, dass `maxPrio` zuerst ausgeführt worden ist, obwohl `minPrio` vorher gestartet worden ist. Sie können also die Ausführungsreihenfolge von Threads mithilfe von deren Priorität ändern.

6.4 Threads beenden

Können laufende Threads beendet werden? Laut Dokumentation ist dies nicht (mehr) möglich, denn die Methode `stop()` einer `Thread`-Instanz ist als *deprecated* gekennzeichnet – von der Verwendung wird also im Sinne einer Zukunftskompatibilität dringend abgeraten.

6.4.1 Verwendung von interrupt() zum Abbrechen eines Thread

In verschiedenen Foren wird gerne empfohlen, die Methode `interrupt()` der jeweiligen `Thread`-Instanz zu verwenden, denn die sei im Gegensatz zur Methode `stop()` nicht `deprecated`.

Um es kurz zu sagen: Wenn es Ihnen nichts ausmacht, dass dabei jedes Mal eine `InteruptedException` geworfen wird und dem Thread keine Möglichkeit gelassen wird, sich sauber zu beenden, dann nutzen Sie diese Methode.

Wir werden davon hier Abstand nehmen, da diese Lösung genau genommen keine Lösung ist, sondern mehr eine letzte Möglichkeit darstellt, um die Beendigung eines Thread zu erzwingen.

Achtung

Die Verwendung von `stop()` oder `interrupt()` führt dazu, dass ein Thread unter Umständen keine Möglichkeit mehr hat, seine Ressourcen aufzuräumen. In der Konsequenz kann dies bedeuten, dass Dateien geöffnet bleiben oder Änderungen nicht übernommen werden.

6.4.2 Verwendung einer eigenen Thread-Ableitung

Viel eleganter, als die Verwendung der Methode `interrupt()` ist der Einsatz einer eigenen `Thread`-Ableitung. Diese Ableitung gibt uns Auskunft darüber, ob eine Beendigung des Thread gewollt ist oder nicht:

Listing 6.5 Die Klasse StoppableThread teilt interessierten Klassen mit, dass der Thread stoppen soll

```
public class StoppableThread extends Thread
{

   private boolean stop = false;

   StoppableThread()
   {
      super();
   }

   StoppableThread(String name)
   {
      super(name);
   }

   StoppableThread(ThreadGroup g, String name)
   {
      super(g, name);
   }

   public boolean getStop()
   {
      return this.stop;
   }

   public void setStop(boolean value)
   {
      this.stop = value;
   }
}
```

Die Klasse `StoppableThread` erbt von `Thread`, was wir mithilfe des Schlüsselworts `extends` definieren. Sie implementiert darüber hinaus Getter- und Setter-Methoden für die private Variable `stop`. Diese boolesche Variable gibt Auskunft darüber, ob der aktuelle Thread angehalten werden soll (dann ist der Wert `true`) oder nicht.

Um die Arbeit mit unserer neuen Klasse zu erleichtern, verfügt diese über drei Konstruktoren. Der Standardkonstruktor nimmt keine Argumente entgegen und ruft mithilfe des Schlüsselworts `super` den Standardkonstruktor

der Super-Klasse auf. Die beiden anderen Konstruktoren nehmen den Namen des Threads und eine Thread-Gruppe, zu der der Thread gehören soll, entgegen. Alle Werte werden via `super` an die übergeordnete Thread-Klasse übergeben – diese kümmert sich dann auch um deren Verarbeitung.

6.4.3 StoppableThread statt Thread als Super-Klasse

Leider ist dies nur die Hälfte der Arbeit – Klassen, die als Thread laufen sollen, müssen nun regelmäßig prüfen, ob der Thread noch weiterlaufen soll oder nicht. Diese Prüfung ist aber zum Glück nicht allzu komplex, wie das folgende Beispiel beweist:

Listing 6.6
Dieses Thread-Element kann sich kontrolliert beenden.

```
public class SimpleStoppableThread extends StoppableThread
{

   public void run() {
      System.out.println("Thread started!");

      // Thread läuft, bis getStop() wahr ist
      while(!this.getStop())
      {
         try
         {
            Thread.sleep(100);
            System.out.print(".");
         }
         catch (InterruptedException e) { }
      }

      System.out.println("\nThread stopped!");
   }

   public static void main(String args[])
   {
      SimpleStoppableThread stoppable = new SimpleStoppableThread();
      stoppable.start();

      try
      {
         Thread.sleep(2000);
      }
      catch (InterruptedException e) { }

      stoppable.setStop(true);
   }
}
```

Das Prinzip, das uns jede Menge Arbeit erspart, ist: Alle neuen Threads erben nicht mehr von der Basisklasse `Thread`, sondern von der eben definierten Klasse `StoppableThread`. Diese stellt mithilfe ihrer Methoden `getStop()` und `setStop()` alles bereit, was wir benötigen, um eine Stopanforderung zu erkennen oder zu setzen.

Nebenbei ersparen wir uns mit dem Erben von `Thread` oder `StoppableThread`, dass wir das Interface `Runnable` explizit implementieren müssen – `Thread` und damit auch alle abgeleiteten Klassen implementieren dies bereits und wir müssen die Methode `run()` nur noch mit unserer Logik

überschreiben. Außerdem können sich unsere Threads nunmehr selbst starten – wir müssen also nicht mehr auf eine externe `Thread`-Instanz zurückgreifen.

Innerhalb der überschriebenen Methode `run()` definiert unsere `Thread`-Implementierung eine `while`-Schleife, deren Abbruchbedingung darin besteht, dass der Getter `getStop()` `true` zurückliefert.

Sollte nun also dem aktuellen Thread der Befehl zum Stoppen gegeben werden, kann dies recht schnell und einfach ermittelt werden – `getStop()` würde in diesem Fall `true` zurückgeben. Spätestens nach 100 Millisekunden (solange warten wir innerhalb der Schleife mithilfe der statischen Methode `sleep` der `Thread`-Klasse) würde sich unser Thread beenden.

Diese Lösung ist viel sauberer und eleganter als ein Abbruch per `stop()`- oder `interrupt()`-Methode, denn sie erlaubt es uns, in Ruhe alle benötigen Ressourcen freizugeben oder andere Aufräumarbeiten vorzunehmen.

Die Ausführung von der Kommandozeile findet mithilfe der statischen Methode `main()` statt, die wir der Einfachheit halber ebenfalls innerhalb der Klasse definieren. Die Methode erstellt und startet eine neue Instanz der `SimpleStoppableThread`-Klasse. Anschließend pausiert sie zwei Sekunden mithilfe von `Thread.sleep()`.

Am Ende wird der aktuelle Thread beendet, indem `setStop()` mit dem Wert `true` aufgerufen wird. Da der Thread regelmäßig den Wert der privaten Variablen `stop` mithilfe des Getter `getStop()` überprüft, kann er sich nun kontrolliert beenden.

Wenn Sie die Lösung starten, sollten Sie folgende Ausgabe erhalten:

Thread started!

.....

Thread stopped!

Würden wir stattdessen die als `deprecated` (also als veraltet und potenziell nicht mehr unterstützt) gekennzeichnete Methode `stop()` verwenden, erhielten wir folgende Ausgabe, die verdeutlicht, warum wir Threads lieber kontrolliert abbrechen sollten:

Thread started!

.....

Hier hatte der Thread offensichtlich keine Möglichkeit mehr, sich kontrolliert zu beenden.

6.5 Zugriff auf den aktuellen Thread

Aus jeder Klasse heraus kann auf den Thread zugegriffen werden, in dem sie läuft. Dies geschieht mithilfe der statischen Methode `currentThread()` der `Thread`-Klasse:

Listing 6.7
Ausgabe von Informationen zum aktuellen Thread

```
public class CurrentThread {

   public static void main(String args[])
   {
      // Auf den aktuellen Thread zugreifen
      Thread current = Thread.currentThread();
      System.out.println(
         String.format(
            "ID: %d\nName: %s\nPriority: %d\nGroup-Name: %s",
            current.getId(),
            current.getName(),
            current.getPriority(),
            current.getThreadGroup().getName()));
   }
}
```

Die Funktion der statischen Methode `main()` ist schnell erläutert: Am Anfang holen wir uns eine Referenz auf den aktuellen Thread mittels `Thread.currentThread()` und weisen diese der lokalen Variablen `current` zu. Dann geben wir einige Informationen zum Thread aus – im Einzelnen sind dies:

- ID des Thread
- Name des Thread
- Priorität des Thread
- Name der Thread-Gruppe, zu der der Thread gehört

Wenn Sie die Klasse kompilieren und ausführen, sollten Sie eine Ausgabe ähnlich wie diese erhalten:

ID: 1

Name: main

Priority: 5

Group-Name: main

6.6 Ermitteln aller laufenden Threads

Mithilfe der `Thread`-Klasse können wir alle laufenden Threads ermitteln. Dies geschieht, indem wir ein Array erzeugen, das Platz für die Gesamtzahl der laufenden Threads bietet und dieses Array der statischen Methode `enumerate()` der `Thread`-Klasse übergeben:

Listing 6.8
Auflisten aller laufenden Threads einer Applikation

```
public class AllThreads extends StoppableThread
{

   public void run()
   {
```

Listing 6.8 (Forts.)
Auflisten aller laufenden Threads einer Applikation

```
        while(!getStop())
        {
            try {
                Thread.sleep(100);
            } catch (InterruptedException e) { }
        }
    }

    public static void main(String[] args)
    {
        for(int i=0; i<18; i++)
        {
            Thread t = new AllThreads();
            t.setName(String.format("Thread %d", i));
            t.start();
        }

        Thread[] threads = new Thread[Thread.activeCount()];
        Thread.enumerate(threads);

        System.out.println("ALL THREADS");
        System.out.println("===========");
        for (Thread ct : threads) {
            System.out.println(ct.getName());
            if(ct instanceof StoppableThread) {
                ((StoppableThread)ct).setStop(true);
            }
        }
    }
}
```

Die Klasse `AllThreads`, die das Auflisten aller laufenden Threads erledigen wird, erbt selbst von der Basisklasse `StoppableThread`, was wir mithilfe des Schlüsselworts `extends` angeben. Die Klasse `StoppableThread` erlaubt es uns, einen Thread kontrolliert zu beenden. Wir haben sie bereits weiter oben in diesem Kapitel ausführlich behandelt.

Die Verwendung von `StoppableThread` als Basisklasse versetzt uns in die Lage, `AllThreads` selbst als Thread zu verwenden. Wir nehmen diese Möglichkeit innerhalb der statischen Methode `main()` wahr und erzeugen gleich einmal achtzehn neue Threads vom Typ `AllThreads`.

Jedem dieser achtzehn Threads wird mithilfe der Setter-Methode `setName()` ein eigener Name zugewiesen. Dieser Name beinhaltet die Thread-Nummer. Anschließend wird der jeweilige Thread mithilfe seiner Methode `start()` gestartet. Er wird so lange laufen, bis er per `setStop()` das Stopsignal erhalten hat. Alternativ könnte der Thread natürlich auch per `stop()` oder `interrupt()` unterbrochen werden, was aber nicht zu empfehlen ist.

Nachdem wir alle Threads gestartet haben, erzeugen wir ein neues Array mit der Anzahl der laufenden Threads als Obergrenze. Diese Anzahl kann mithilfe der statischen Methode `activeCount()` der `Thread`-Klasse ermittelt werden. Dem Array werden nun mithilfe der statischen Methode `enumerate()` der `Thread`-Klasse alle laufenden Thread-Instanzen zugewiesen.

Jetzt verfügen wir über eine Liste aller Threads, die wir nun per `for-each`-Schleife durchlaufen können. Dabei wird bei jedem Durchlauf der lokalen Variablen `ct` eine Referenz auf den jeweiligen Thread zugewiesen und dessen Name kann mithilfe von `getName()` ermittelt und ausgeben werden.

Zuletzt überprüfen wir jeden Thread mithilfe des Schlüsselworts `instanceof` darauf, ob er eine `StoppableThread`-Instanz ist. Sollte dies der Fall sein, wird er in den Typ `StoppableThread` gecastet und ihm kann mithilfe seiner Methode `setStop()` signalisiert werden, dass er sich beenden soll – schließlich benötigen wir ihn nicht mehr in unserer Applikation und möchten die entsprechenden Ressourcen wieder freigeben.

Wenn Sie die Klasse kompilieren und ausführen, sollten Sie eine Ausgabe analog zu dieser erhalten:

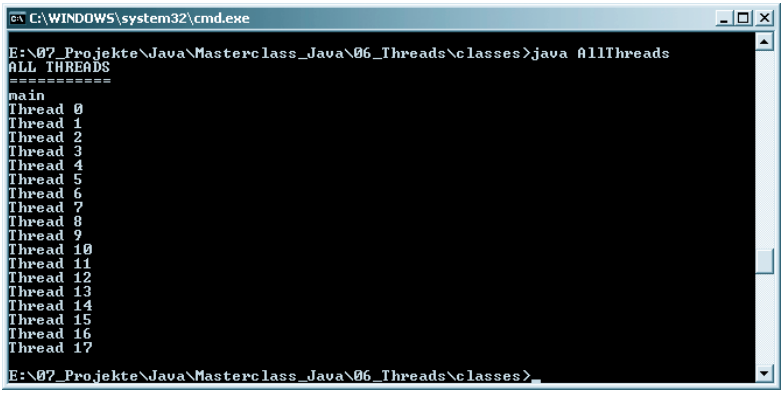


Abbildung 6.3
Ausgabe aller laufenden Threads

Der oberste Thread `main` entspricht übrigens dem Thread, den wir gestartet haben, indem wir die Klasse von der Kommandozeile aus aufgerufen haben. Er wird als letzter Thread beendet werden.

6.7 Organisation von Threads in Thread-Gruppen

Thread-Gruppen stellen eine Organisationsform von Threads dar, mit deren Hilfe diese leichter verwaltet und gemanaged werden können. Jeder Thread kann genau einer Thread-Gruppe angehören und von dieser beeinflusst werden. Sehen wir uns dies in einem Beispiel an:

Listing 6.9
ThreadGroups erlauben die Organisation der enthaltenen Threads

```
public class ThreadGroupExample extends StoppableThread
{

   ThreadGroupExample(ThreadGroup g, String name)
   {
      super(g, name);
   }

   public void run()
   {
      while(!this.getStop())
      {
         try
         {
            Thread.sleep(100);
```

Listing 6.9 (Forts.)
ThreadGroups erlauben die Organisation der enthaltenen Threads

```
            } catch (InterruptedException e) { }
        }
    }

    private static void enumerateGroup(ThreadGroup tg)
    {
        System.out.println("ENUMERATING GROUP");
        System.out.println("================");
        System.out.println(String.format("Group: %s", tg.getName()));

        Thread[] tgItems = new Thread[tg.activeCount()];
        tg.enumerate(tgItems);

        for(Thread t : tgItems)
        {
            System.out.println(
                String.format("     Thread %s (%d)",
                t.getName(), t.getPriority()));
            if(t instanceof StoppableThread)
            {
                ((StoppableThread)t).setStop(true);
            }
        }

        System.out.println();
    }

    public static void main(String args[])
    {
        ThreadGroup group = new ThreadGroup("sampleGroup");
        group.setMaxPriority(3);
        for(int i=0; i<6; i++)
        {
            Thread t = new ThreadGroupExample(
                group, String.format("%d", i));
            t.start();
        }

        ThreadGroup secondGroup =
            new ThreadGroup("secondGroup");
        for(int i=5; i<12; i++)
        {
            Thread t = new ThreadGroupExample(
                secondGroup, String.format("%d", i));
            t.start();
        }

        enumerateGroup(group);
        enumerateGroup(secondGroup);
    }
}
```

Die Klasse `ThreadGroupExample` ist selbst eine Ableitung der bereits weiter vorne in diesem Kapitel beschriebenen Klasse `StoppableThread`. Dadurch verfügt sie über die Getter und Setter `getStop()` und `setStop()`, die es uns erlauben, einen Thread kontrolliert zu beenden. Voraussetzung dafür ist natürlich, dass die `Thread`-Instanz dies in ihrer `run()`-Methode überprüft – was hier der Fall ist.

Die Klasse `ThreadGroupExample` verfügt neben der Methode `run()` über einen Konstruktor, der die Thread-Gruppe und den Namen des Threads entgegennimmt.

Die Methode main()

Die eigentlich interessanten Dinge finden jedoch in den Methoden `main()` und `enumerateGroup()` statt. Sehen wir uns zunächst die `main()`-Methode etwas näher an: Diese stellt die Startmethode der Klasse dar, die eingebunden wird, wenn ein Aufruf über die Kommandozeile erfolgt.

Zuerst erzeugen wir hier die `ThreadGroup` »sampleGroup«, für die wir festlegen, dass alle enthaltenen Threads eine maximale Thread-Priorität von 3 erhalten sollen. Eine derartige Festlegung muss vor dem Zuweisen von Threads erfolgen, denn sie zeigt keinerlei Auswirkungen auf bereits vorhandene Elemente.

Danach werden sechs neue Threads erzeugt und durch `<ThreadInstanz>.start()` aktiviert. Bei ihrer Erzeugung wird der neuen Instanz die Thread-Gruppe, zu der sie gehört und ihr jeweiliger Name als Parameter übergeben (wobei der Name hier der Einfachheit halber nur aus einer laufenden Nummer besteht).

Jetzt erzeugen wir eine weitere Thread-Gruppe namens »secondGroup«. Dieser Thread-Gruppe werden ebenfalls sechs Threads zugeordnet, indem diesen bei der Erzeugung die `secondGroup`-Instanz als Parameter übergeben wird.

enumerateGroup()

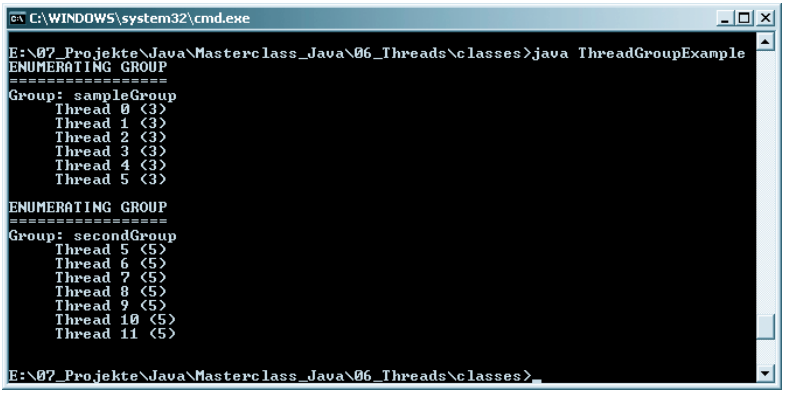
Nachdem wir nunmehr über zwei Thread-Gruppen mit jeweils sechs Elementen verfügen, lassen wir uns deren Detail-Informationen mithilfe der ebenfalls statischen Methode `enumerateGroup()` ausgeben. Zuerst erfolgt die Ausgabe des Gruppennamens, den wir mithilfe von `getName()` ermitteln können.

Anschließend erzeugen wir ein leeres Thread-Array mit der von `<ThreadGroup>.activeCount()` zurückgegebenen Anzahl an Elementen. Das Array lassen wir per `<Threadgroup>.enumerate()` befüllen. Wir erhalten so alle der Thread-Gruppe zugeordneten Threads in einem Array, dass wir anschließend per `for-each`-Schleife durchlaufen.

Für jeden enthaltenen Thread geben wir seinen Namen und seine Priorität aus. Der guten Ordnung halber überprüfen wir zuletzt mithilfe des Schlüsselworts `instanceof`, ob der Thread eine Instanz von `StoppableThread` ist, und beenden ihn, falls dies zutrifft, indem wir ihn zunächst nach `StoppableThread` casten und dann seiner `setStop()`-Methode den Wert `true` übergeben.

Nach dieser Menge an Informationen sollten wir unsere Klasse einfach von der Kommandozeile ausführen. Die Ausgabe sollte dann so aussehen:

Abbildung 6.4
Gruppierung von Threads



Thread-Gruppen eignen sich also sehr gut, um Threads zu organisieren. Gerade dann, wenn Sie viele Threads in einer Applikation verwenden (müssen), bietet sich ihr Einsatz an.

6.8 Synchronisierung von Threads

Wenn Threads miteinander kommunizieren oder Dateisystemzugriffe durchführen wollen, ist eine Synchronisierung dieser Prozesse unerlässlich – beispielsweise könnte der Zugriff auf eine Datei, in die von einem anderen Thread gerade geschrieben wird, fehlschlagen oder Variablen könnten undefinierte Werte halten.

Nehmen wir als Beispiel eine gemeinsame Zählvariable, die von zwei Threads parallel genutzt und hochgezählt werden soll:

```
public class SynchronizedThread extends Thread
{

   static int counter = 0;

   public void run()
   {
      while(true)
      {
         int current = counter++;
         try
         {
            Thread.sleep(10);
         }
         catch (InterruptedException ex) {}
         System.out.println(current);
      }
   }

   public static void main(String[] args)
   {
      Thread t1 = new SynchronizedThread();
      t1.start();
```

```
        Thread t2 = new SynchronizedThread();
        t2.start();
    }
}
```

Wenn Sie dieses Beispiel eine Weile laufen lassen, werden Sie früher oder später feststellen, dass der Zugriff auf die Zählvariable offensichtlich nicht synchronisiert ist:

```
650
653
652
655
654
657
656
659
658
661
660
663
662
665
664
667
666
669
668
671
670
```

Wie anders ist es zu erklären, dass die Zahlen nicht in korrekter Reihenfolge ausgegeben werden?

Der Hintergrund dieses Verhaltens liegt darin, wie Multitasking und die Verwaltung von Prozessen implementiert ist: Der Scheduler, der für das Zuweisen von Rechenzeit zu den einzelnen Threads zuständig ist, hat den einen Thread nach dem Erhöhen der Zählvariablen, aber vor dem Ausgeben des neuen Werts unterbrochen und den anderen Thread aktiviert. Dieser hat nun seinerseits die Zählvariable hochgezählt und deren Wert ausgegeben.

Um dieses Verhalten zu verhindern, benötigen wir einen Prozess, mit dem wir die Aktionen der einzelnen Threads synchronisieren können.

6.8.1 Monitor

Die Synchronisation von Threads erfolgt mithilfe von *Monitoren*: Kritische Code-Fragmente werden automatisch gesperrt und erst nach dem Durchlauf des Fragments wieder freigegeben. Ist der Code-Bereich bereits gesperrt, muss ein Thread warten, bis dieser Bereich wieder freigegeben worden ist.

In Java erfolgt die Synchronisierung mithilfe des Schlüsselworts `synchronized`. Dieses Schlüsselwort ist in der Lage, entweder einen Bereich oder eine ganze Methode zu schützen. Der Einsatz von `synchronized` erfordert, dass entweder eine Objektvariable oder der Verweis auf die aktuelle Klas-

seninstanz per `this`-Schlüsselwort als Parameter angegeben werden. Wichtig dabei ist, dass die Objektvariablen unterschiedlicher Threads, die miteinander synchronisiert werden sollen, stets auf die gleiche Objektinstanz zeigen. Gibt es keine Membervariable in unterschiedlichen Threads, die auf die gleiche Objektinstanz zeigt, kann man sich damit behelfen, dass man mithilfe der Methode `getClass()` eine derartige Referenz selbst erzeugt.

Um die Threads des obigen Beispiels zu synchronisieren, kapseln wir den kritischen Bereich mithilfe eines `synchronized`-Blocks:

Listing 6.10 Synchronisation zweier Threads per Monitor

```
public class SynchronizedThread extends Thread {

   static int counter = 0;


   public void run()
   {
      while(true)
      {
         // Synchronisierter Block, Synchronisation erfolgt
         // Anhand des Klassentyps
         synchronized(this.getClass())
         {
            // Zählerstand erhöhen und in lokaler
            // Membervariablen einlesen
            int current = counter++;
            try
            {
               // Thread pausieren lassen
               Thread.sleep(10);
            }
            catch (InterruptedException ex) {}

            // Aktuellen Zählerstand ausgeben
            System.out.println(current);
         }
      }
   }

   public static void main(String[] args)
   {
      // Zwei Threads erzeugen und starten
      Thread t1 = new SynchronizedThread();
      t1.start();

      Thread t2 = new SynchronizedThread();
      t2.start();
   }
}
```

Nunmehr wird die Ausgabe der beiden Threads in der korrekten Reihenfolge geschehen:

```
0
1
2
3
4
5
6
7
8
```

Um eine Methode eines Objekts, das etwa von mehreren Instanzen parallel verwendet wird, zu synchronisieren, reicht es aus, das Schlüsselwort `synchronized` in die Methodensignatur aufzunehmen.

Nehmen wir wieder ein Beispiel: Hier haben wir das Verwalten und Hochzählen eines Zählers ausgelagert – die Klasse `Counter` erledigt genau dies. Mithilfe ihrer Methode `increment()` wird der Zähler erhöht und das Ergebnis dieser Operation zurückgegeben. Um dabei eine zeitaufwändigere Operation zu simulieren, lassen wir den jeweils aktuellen Thread bei der Ausführung ein wenig pausieren:

Listing 6.11
Die Instanz der Klasse Counter ist nicht synchronisiert

```
class Counter {

   private int id = 0;

   public int increment()
   {
      // Anzahl erhöhen
      int result = id++;
      try
      {
         // Thread pausieren lassen
         Thread.sleep(100);
      }
      catch (InterruptedException ie) {}
      return result;
   }
}

public class SyncMethod extends Thread
{
   // Membervariablen
   private Counter counter;
   private String name;

   /**
    * Konstruktor
    */
   SyncMethod(Counter counter, String name)
   {
      this.counter = counter;
      this.name = name;
   }

   /**
    * Ausführung des Codes bei Thread-Ausführung
    */
   public void run()
   {
      while(true)
      {
         System.out.println("Name: " + name +
```

Listing 6.11 (Forts.)
Die Instanz der Klasse Counter ist nicht synchronisiert

```
                ", Counter: " + counter.increment());
        }
    }

    public static void main(String[] args)
    {
        // Instanzen erzeugen und ausführen lassen
        Counter counter = new Counter();
        Thread[] t = new Thread[4];
        for(int i=0; i<4; i++)
        {
            t[i] = new SyncMethod(counter, "Thread_" + i);
            t[i].start();
        }
    }
}
```

Beim Start der Applikation werden fünf Instanzen der `SyncMethod`-Klasse erzeugt. Diese erben jeweils von `Thread` und geben in ihrer `run()`-Methode ihren Namen und den Zählerstand aus. Im Konstruktor wird ihnen eben dieser Name und eine Instanz der `Counter`-Klasse als Parameter übergeben.

Noch einmal zur Verdeutlichung: Alle fünf Thread-Instanzen greifen auf die gleiche Counter-Instanz zurück. Deren Methode `increment()` ist (noch) nicht synchronisiert. Und das kommt dann dabei heraus:

```
Name: Thread_0, Counter: 0
Name: Thread_1, Counter: 1
Name: Thread_2, Counter: 2
Name: Thread_3, Counter: 3
Name: Thread_3, Counter: 7
Name: Thread_2, Counter: 6
Name: Thread_1, Counter: 5
Name: Thread_0, Counter: 4
Name: Thread_0, Counter: 11
Name: Thread_1, Counter: 10
Name: Thread_2, Counter: 9
Name: Thread_3, Counter: 8
```

Um dieses Verhalten zu vermeiden, kennzeichnen wir die Methode `increment()` der `Counter`-Klasse als `synchronized`:

Listing 6.12
Nun ist die Methode increment() als synchronized gekennzeichnet

```
class Counter
{

    private int id = 0;

    /**
     * Synchronisierte Verarbeitung
     */
    public synchronized int increment()
    {
        int result = id++;
        try
        {
            Thread.sleep(100);
        }
        catch (InterruptedException ie) {}

        return result;
    }
}
```

Wird die Methode `increment()` als `synchronized` gekennzeichnet, sieht die Ausgabe komplett anders aus:

```
Name: Thread_0, Counter: 0
Name: Thread_1, Counter: 1
Name: Thread_2, Counter: 2
Name: Thread_3, Counter: 3
Name: Thread_0, Counter: 4
Name: Thread_1, Counter: 5
Name: Thread_2, Counter: 6
Name: Thread_3, Counter: 7
Name: Thread_0, Counter: 8
Name: Thread_1, Counter: 9
Name: Thread_2, Counter: 10
Name: Thread_3, Counter: 11
Name: Thread_0, Counter: 12
Name: Thread_1, Counter: 13
Name: Thread_2, Counter: 14
Name: Thread_3, Counter: 15
```

Beachten Sie auch die Reihenfolge der Threads – diese ist nunmehr wesentlich geordneter, denn das Schlüsselwort `synchronized` bewirkt, das die komplette Methode so lange gesperrt bleibt, bis sie komplett durchlaufen worden ist. Salopp gesagt, bedeutet das für die Threads: hinten anstellen und warten, bis man an der Reihe ist.

6.8.2 wait() und notify()

Ein synchronisiertes Objekt oder ein synchronisierter Block verfügt über eine sogenannte Warteliste. In dieser Liste werden vom Scheduler alle Threads eingetragen, die auf die Beendigung der Sperre des Objekts oder Blocks warten.

Der Aufruf der Methode `wait()` innerhalb eines synchronisierten Blocks oder einer synchronisierten Methode führt dazu, dass diese Methode oder der Block als wartend markiert und in die Warteliste des Objekts, anhand dessen synchronisiert wird, aufgenommen werden kann. Ein Objekt kann sich somit aktiv selbst aus dem Verkehr ziehen und anderen Objekten die Abarbeitung des Blocks oder der Methode ermöglichen.

Der Aufruf von `notify()` sorgt dafür, dass die als wartend markierten Threads die entsprechenden Methoden oder Blöcke ausführen können. Der Aufrufer wird dagegen aus der Warteliste entfernt und stellt sich somit quasi an deren Ende wieder an. Die Warteliste wird danach wie gewohnt abgearbeitet.

Wichtig ist, das stets die `wait()`- und `notify()`-Methoden der als Parameter des `synchronized`-Statements verwendeten Objektinstanz aufgerufen werden.

Beide Methoden eignen sich somit für Szenarien, in denen die zeitliche Abfolge der Prozesse gesteuert werden soll.

Nehmen wir ein Beispiel: Dieses demonstriert anhand eines *Producer/Consumer*-Ansatzes die Synchronisation und zeitliche Steuerung von Prozessen. Der Producer (Produzent) stellt etwas her – in unserem Fall einige

zufällig erzeugte Zahlen, die von einem der Consumer (Verbraucher) dann ausgegeben werden sollen. Dabei soll stets nur ein Consumer die Ausgabe übernehmen und in der Zeit dürfen auch keine anderen Daten in die zum Datenaustausch verwendete ArrayList eingestellt werden. Um den ganzen Prozess noch etwas komplexer zu gestalten, pausieren sowohl Producer als auch Consumer jeweils für einen zufälligen Zeitraum.

Sehen wir uns an, wie dies umgesetzt werden könnte:

Listing 6.13 wait() und notify() im Einsatz

```
import java.util.ArrayList;

class Producer extends Thread
{

   // ArrayList mit den Daten
   private ArrayList<Integer> data;

   /**
    * Konstruktor
    */
   Producer(ArrayList<Integer> data)
   {
      this.data = data;
   }

   /**
    * Hier werden die Daten erzeugt
    */
   public void run()
   {
      int current;

      while(true)
      {
         // Synchronisierter Block
         synchronized(data)
         {
            // Drei neue Zufallszahlen erzeugen
            for(int i=0;i<3;i++)
            {
               current = (int)(Math.random() * 100);
               data.add(current);
               System.out.println("Producer produced: " + current);
            }

            // Eventuelle Konsumenten benachrichtigen
            data.notify();
         }

         // Zufälligen Zeitraum warten
         try
         {
            Thread.sleep((int)Math.random() * 100);
         }
         catch (InterruptedException e) {}
      }
   }
}

/**
 * Konsumiert die vom Producer erzeugten Daten
 */
class Consumer extends Thread {
```

Listing 6.13 (Forts.)
wait() und notify()
im Einsatz

```
   private String name;
   // Daten
   private ArrayList<Integer> data;

   /**
    * Konstruktor
    */
   Consumer(String name, ArrayList<Integer> data)
   {
      this.name = name;
      this.data = data;
   }

   /**
    * Ausführen der Verarbeitung
    */
   public void run()
   {
      while(true)
      {
         // Synchronisierter Block
         synchronized(data) {
            // Anzahl der produzierten Daten überprüfen
            if(data.size() < 1)
            {
               // Wenn keine Daten verfügbar, dann warten
               try
               {
                  data.wait();
               }
               catch (InterruptedException e) {}
            }
            else
            {
               // Ausgabe der Daten
               for(int current : data)
               {
                  System.out.println(
                     "Consumer " + name +
                        " found value: " + current);
               }

               // Datenliste leeren
               data.clear();
            }

            try
            {
               Thread.sleep((int) Math.random() * 100);
            }
            catch (InterruptedException e) {}
         }
      }
   }
}

public class ProducerConsumer {

   public static void main(String[] args)
   {
      // Gemeinsame Liste für die Daten erzeugen
      ArrayList<Integer> data = new ArrayList<Integer>();

      // Producer erzeugen
      Producer producer = new Producer(data);
      producer.start();
```

Listing 6.13 (Forts.) wait() und notify() im Einsatz

```
        // Consumer erzeugen
        Consumer c1 = new Consumer("#1", data);
        c1.start();

        Consumer c2 = new Consumer("#2", data);
        c2.start();
    }
}
```

Sowohl Consumer- als Producer-Instanzen verfügen über eine Instanz einer `ArrayList`. Es handelt sich dabei immer um die gleiche Objektinstanz. Anhand dieser Instanz können Synchronisierung, Warten und Aktivieren der verschiedenen Threads erfolgen.

Die Consumer-Threads prüfen zu diesem Zweck innerhalb ihrer `run()`-Methode, ob in der ArrayList Elemente enthalten sind. Wenn das nicht der Fall sein sollte, wird der aktuelle Block mithilfe von `wait()` in die Warteschleife eingereiht und wartet somit, bis er wieder aktiviert wird. Sobald das geschieht, gibt der Block die in der `ArrayList` enthaltenen Daten aus und leert diese danach.

Der Produzent erzeugt innerhalb seiner `run()`-Methode stets drei neue Elemente für die `ArrayList`. Da dies auch in einem `synchronized`-Block geschieht, findet die Erzeugung nur statt, wenn dieser Block nicht gesperrt ist. Nach der Erzeugung der Daten benachrichtigt der Produzent alle wartenden Blöcke mithilfe der Methode `notify()`.

Dieser Vorgang wiederholt sich nun kontinuierlich – und die Ausgabe zeigt, dass `wait()` und `notify()` gute Dienste leisten:

```
Producer produced: 82
Producer produced: 2
Producer produced: 79
Consumer #1 found value: 82
Consumer #1 found value: 2
Consumer #1 found value: 79
Producer produced: 38
Producer produced: 46
Producer produced: 53
Consumer #2 found value: 38
Consumer #2 found value: 46
Consumer #2 found value: 53
Producer produced: 9
Producer produced: 98
Producer produced: 64
Consumer #2 found value: 9
Consumer #2 found value: 98
Consumer #2 found value: 64
Producer produced: 37
Producer produced: 48
Producer produced: 47
Consumer #1 found value: 37
Consumer #1 found value: 48
Consumer #1 found value: 47
Producer produced: 7
Producer produced: 60
Producer produced: 49
Consumer #1 found value: 7
Consumer #1 found value: 60
Consumer #1 found value: 49
```

6.9 Fazit

Die Arbeit mit Threads ist komplex und spannend zugleich. Threads bieten uns viele Möglichkeiten, sich wiederholende und ressourcenintensive Vorgänge auszulagern und somit die Ausführung der Hauptapplikation nicht zu behindern.

Die Zuordnung verschiedener Threads zu Thread-Gruppen bietet interessante Möglichkeiten der Verwaltung. Dass die Arbeit mit Threads auch komplex werden kann, belegen die gezeigten Synchronisationsszenarien. Doch mithilfe des Schlüsselworts `synchronized` und der beiden Methoden `wait()` und `notify()` sind wir in der Lage, auch komplexere Synchronisationsaufgaben erfolgreich zu bewältigen.