Einführung Threads

Inhalt

Für nebenläufige Programme in Java ist die Klasse `Thread` zuständig. Jeder Thread ist somit eng mit einem Exemplar dieser Klasse verbunden. Die von uns gewünschte, parallel auszuführende Aktivität programmieren wir in der Objektmethode `run()`. Wird das Thread-Objekt gestartet, arbeitet es die `run()`-Programmzeilen parallel ab. Java bietet zur Programmierung von Threads zwei Möglichkeiten an, die wir nachfolgend vorstellen werden.

Threads über die Schnittstelle Runnable

Eine Klasse, deren Exemplare Programmcode parallel ausführen sollen, muss die Schnittstelle `Runnable` implementieren, die eine `run()`-Methode vorschreibt.

Beispiel Wir wollen zwei Threads angeben, wobei einer zwanzigmal das aktuelle Datum und die Uhrzeit ausgibt und der andere einfach eine Zahl.

```
class DateThread implements Runnable {
   public void run() {
         for (int i = 0; i < 20; i++)
               System.out.println(new Date());
   }
}

class CounterThread implements Runnable {
   public void run() {
         for (int i = 0; i < 20; i++) {
               System.out.println(i);
         }
   }
}
```

In der Methode `run()` wird der auszuführende Programmcode eingebettet. Das heißt, bei dem einen in einer Schleife das Date-Objekt auszugeben und in dem anderen die lokale Variable der Schleife.

Starten

Nun reicht es nicht aus, einfach die `run()`-Methode einer Klasse direkt aufzurufen. Würden wir dies tun, dann wäre nichts nebenläufig, sondern wir würden einfach eine Methode sequenziell ausführen. Damit der Programmcode nebenläufig zur Applikation läuft, müssen wir ein `Thread`-Objekt erzeugen und dann den Thread explizit starten. Dazu übergeben wir dem Konstruktor der Klasse `Thread` eine Referenz auf das `Runnable`-Objekt und rufen `start()` auf. Nachdem `start()` für den Thread eine Ablaufumgebung geschaffen hat, ruft es dann selbstständig die Methode `run()` genau einmal auf. Läuft der Thread schon, so wirft die `start()`-Methode eine `IllegalThreadStateException` aus.

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

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

Beim Starten des Programms erfolgt eine Ausgabe auf dem Bildschirm, die etwa so aussehen kann:

```
Thu Jan 18 15:40:20 CET 2001
0
1
2
3
4
5
6
7
8
9
Thu Jan 18 15:40:20 CET 2001
10
...
```

Deutlich ist die Verzahnung der beiden Prozesse zu erkennen. Was allerdings auf den ersten Blick etwas merkwürdig wirkt, ist die erste Zeile des Datum-Threads und viele weitere Zeilen des Zähl-Threads. Dies hat jedoch nichts zu sagen und zeigt deutlich den Nichtdeterminismus bei Threads.

Automatisches Starten während der Erzeugung

Erst der Aufruf der `start()`-Methode lässt den Thread ablaufen. Etwas eleganter ist der Weg, dass das Objekt seinen eigenen Thread verwaltet, der im Konstruktor gestartet wird. Dann muss nur dort ein Thread-Objekt für die `Runnable`-Umgebung angelegt und die `start()`-Methode ausgeführt werden.

```
class DateThreadAutoStart implements Runnable {
   DateThreadAutoStart() {
         new Thread(this).start();
   }

   public void run() {
         for (int i = 0; i < 20; i++)
               System.out.println(new Date());
   }
}
```

Die Klasse Thread erweitern

Da die Klasse `Thread` selbst die Schnittstelle `Runnable` implementiert und die `run()`-Methode mit leerem Programmcode bereitstellt, können wir auch `Thread` erweitern, wenn wir eigene parallele Aktivitäten programmieren wollen. Dann müssen wir kein `Runnable`-Exemplar mehr in den

Konstruktor einfügen, denn wenn unsere Klasse Unterklasse von `Thread` ist, reicht ein Aufruf der geerbten Methode `start()`. Danach arbeitet das Programm direkt weiter, führt also kurze Zeit später die nächste Anweisung hinter `start()` aus. Dann lassen sich auch mehrere Exemplare der `Thread`-Klasse erzeugen.

Fassen wir die drei Schritte noch einmal zusammen:

- Wir erweitern die Klasse `Thread` mit oder in einer neuen Unterklasse.
- Für ein Programmstück, das nebenläufig ablaufen soll, überschreiben wir die `run()`-Methode. `run()` darf keinen Parameter enthalten, wenn doch, stimmt die Signatur nicht mit der in der Oberklasse `Thread` überein und statt des Überschreibens gäbe es ein Überladen. Dann aber geschieht beim Starten des Threads nichts, weil die geerbte parameterlose (und leere) `run()`-Methode aufgerufen wird.
- Damit der Thread und damit das Programm startet, rufen wir die Objektmethode `start()` auf. Sie veranlasst die Laufzeitumgebung, Ressourcen zur Verfügung zu stellen. `start()` ruft dann automatisch `run()` auf. `start()` kann von uns auch überschrieben werden, was aber nur selten sinnvoll beziehungsweise nötig ist. Wir müssen dann darauf achten, `super.start()` aufzurufen, damit der Thread wirklich losrennt.

```
class DateThreadExtends extends Thread {
   public void run() {
         for (int i = 0; i < 20; i++)
               System.out.println(new Date());
   }
}
```

Der Startschuss fällt für den Thread wieder beim Aufruf von `start()`.

```
Thread t = new DateThreadExtends();
t.start();
```

Oder auch ohne Zwischenspeicherung der Objektreferenz:

```
new DateThreadExtends().start();
```

Damit wir als Thread-Benutzer nicht erst die `start()`-Methode aufrufen müssen, kann ein Thread sich auch wieder selbst starten. Der Konstruktor enthält einfach als letzte Anweisung den Methodenaufruf `start()`.

```
class DateThreadExtends extends Thread {
   DateThreadExtends() {
         start();
   }
   // ...der Rest bleibt...
}
```

```
class java.lang.Thread implements Runnable
```

- `void run()`
  Diese Methode enthält den parallel auszuführenden Programmcode.
- `void start()`
  Ein neuer Thread – neben dem die Methode aufrufenden Thread – wird gestartet. Der neue Thread führt die `run()`-Methode nebenläufig aus. Läuft der Thread schon, wird eine `IllegalThreadStateException` ausgelöst.

Was passiert, wenn wir statt start() nun run() aufrufen?

Ein Programmierfehler, der hin und wieder passiert, ist folgender: Anstatt `start()` rufen wir aus Versehen `run()` auf. Was geschieht? Fast genau das Gleiche wie bei `start()`, nur mit dem Unterschied, dass die Objektmethode `run()` nicht parallel zum übrigen Programm abgearbeitet wird. Der aktuelle Thread bearbeitet die `run()`-Methode sequenziell bis sie zu Ende ist und die Anweisungen nach dem Aufruf an die Reihe kommen. Der Fehler fällt nicht immer direkt auf, denn die Aktionen in `run()` finden ja statt – nur nicht nebenläufig.

Runnable ist kein Thread. Wie können wir die Thread-Methoden nutzen?

Eine Erweiterung der Klasse `Thread` hat den Vorteil, dass alle geerbten Methoden sofort genutzt werden können. Wenn wir `Runnable` implementieren, dann genießen wir den Vorteil nicht. Wie lassen sich dann trotzdem die Operationen nutzen? Dazu bietet `Thread` die Klassenmethode `currentThread()` an, um die Objektreferenz für das Thread-Exemplar zu holen, das gerade diese Methode ausführt.

Beispiel Die aktuelle Priorität des laufenden Threads soll ausgegeben werden.

```
System.out.println(Thread.currentThread().getPriority());
```

Erweitern von Thread oder implementieren von Runnable?

	Vorteil	**Nachteil**
Ableiten von Thread (extends Thread)	Programmcode in run() kann die Methoden der Klasse Thread nutzen.	Da es in Java keine Mehrfachvererbung gibt, kann die Klasse nur Thread erweitern.
Implementieren von Runnable (implements Runnable)	Die Klasse kann von einer anderen, problemspezifischen Klasse erben.	Kann sich nur mit Umwegen selbst starten; allgemein: Thread-Methoden können nur über Umwege genutzt werden.

Referenzen

[JavaInsel]	Java ist auch eine Insel (2. Aufl.) von Christian Ullenboom Programmieren für die Java 2-Plattform in der Version 1.4 Kapitel 9 Threads und nebenläufige Programmierung http://www.galileocomputing.de/openbook/javainsel3/javainsel_090001.htm