

# Java Threads

-  
Proseminar Parallelität  
Wintersemester 1999/2000

Manuel Klimek<sup>1</sup>

07. Februar 2000

<sup>1</sup>email: [klimek@fmi.uni-passau.de](mailto:klimek@fmi.uni-passau.de)

---

<b>1</b>	<b>Einführung</b>	<b>3</b>
<b>2</b>	<b>Konzeptuelles</b>	<b>3</b>
2.1	Erstellen eines Threads . . . . .	3
2.1.1	Erben der Klasse <i>Thread</i> . . . . .	3
2.1.2	Implementieren des <i>Runnable</i> -Interfaces . . . . .	3
2.2	Synchronisation . . . . .	4
2.2.1	<i>synchronized</i> - Blöcke . . . . .	4
2.2.2	<i>wait()</i> - und <i>notify()</i> -Konstrukte . . . . .	5
2.2.3	Unterschiede zum traditionellen Monitor . . . . .	6
2.3	Scheduling . . . . .	7
2.3.1	Green Threads . . . . .	7
2.3.2	Native Threads . . . . .	8
<b>3</b>	<b>Probleme und Lösungen</b>	<b>8</b>
3.1	Lokale Variablen in Threads . . . . .	8
3.2	Deprecated Methods . . . . .	8
3.3	Priority Mapping . . . . .	9
3.4	Starvation . . . . .	9
<b>4</b>	<b>Zusammenfassung</b>	<b>10</b>
<b>A</b>	<b>Kurzreferenz - Die wichtigsten Klassen</b>	<b>11</b>
A.1	Die Klasse <i>Thread</i> . . . . .	11
A.1.1	Konstruktoren . . . . .	11
A.1.2	Variablen . . . . .	11
A.1.3	Methoden . . . . .	11
A.2	Die Klasse <i>Object</i> . . . . .	12

---

# 1 Einführung

Java wird heutzutage als die anwendungsorientierte Programmiersprache der nächsten Jahre gehandelt. Unter anderem verdankt Java diesen Ruf der Tatsache, dass es mit Java zum ersten Mal gelungen ist, eine high-level Entwicklungsumgebung mit relative guter Performance zur Verfügung zu stellen. Zur anwendungsorientierten Programmierung benötigt man natürlich unter anderem eine Schnittstelle, mit der sowohl einfache Thread-basierte Anwendungen intuitiv und schnell zu gestalten sind, als auch kompliziertere Probleme gelöst werden können. Diese Schnittstelle wird im Folgenden vorgestellt.

## 2 Konzeptuelles

### 2.1 Erstellen eines Threads

Zunächst ein kurzer Überblick über die verschiedenen Möglichkeiten, in Java einen Thread zu erzeugen.

#### 2.1.1 Erben der Klasse *Thread*

Dies stellt die gradlinigste Implementation dar. Man hat nur die *run()*-Methode der Klasse *Thread* zu überschreiben, und diese mit dem im Thread abzuarbeitenden Code zu füllen. Danach startet man den Thread, indem man die geerbte *start()*-Methode ausführt. Diese erstellt einen neuen konkurrierenden Thread, in dem die (überschriebene) *run()*-Methode ausgeführt wird. Der Thread wird automatisch beendet, wenn ein *return* aus der *run()*-Methode erfolgt.

```
class MyThread extends Thread {
    public void run() {
        /* Hier den auszufuehrenden Code-Teil
           einfuegen... */
    }
}

// ... und zum Starten des Threads:
(new MyThread()).start();
```

#### 2.1.2 Implementieren des *Runnable*-Interfaces

In Java wurde auf Mehrfachvererbung verzichtet, um den Programmierer zum durchdachten Aufbau seiner Vererbungsbäume zu zwingen. Durch Interfaces wird jedoch ein zusätzliches Sprachkonstrukt geboten, welches ausreichende Mächtigkeit für fast alle Gestaltungswünsche bietet. Deshalb existiert eine weitere Möglichkeit, einen Thread zu erzeugen: Das Interface *Runnable*. Dieses enthält nur die abstrakte Spezifikation der *run()*-Methode, die, wie auch im vorigen Abschnitt, den auszuführenden Thread-Code enthalten muss.

Hat man eine Klasse, die das *Runnable*-Interface implementiert, so kann man diese einem neuen *Thread*-Objekt im Konstruktor übergeben. Die *start()*-Methode des neuen *Thread*s führt dann die *run()*-Methode im übergebenen Objekt aus.

```
class MyThread implements Runnable {
    public void run() {
        /* Hier den auszufuehrenden Code-Teil
           einfuegen... */
    }
}

// gestartet wird jetzt wie folgt:
(new Thread(new MyThread())).start();
```

## 2.2 Synchronisation

Ebenso wie in vielen anderen anwendungsorientierten Sprachen, muss der Programmierer in Java die Synchronisation der kritischen Funktionen selbst in die Hand nehmen. Da dies eine mitunter äußerst schwierige Aufgabe darstellen kann, bietet Java ein monitorähnliches Konzept, dessen Anwendung sich relativ intuitiv gestaltet.

### 2.2.1 *synchronized* - Blöcke

In Java besitzt jedes Objekt (auch Klassen-Objekte) einen Monitor. Diesen betritt man durch das Schlüsselwort *synchronized(Object o)*.

```
Object object = new Object();

/* Falls sich beim Aufruf dieser
   Funktion schon ein Thread im Monitor
   zu object befindet, wird der aktuelle
   Thread so lange angehalten, bis ein
   anderer Thread den Monitor wieder
   verlaesst. */

synchronized(object) {

    /* Wir befinden uns im Monitor zu object.
       Nun koennen wir auf saemtliche kritischen
       Variablen zugreifen, die durch den
       Monitor geschuetzt werden. */

    doSomethingCritical();
}
```

Man kann als Vereinfachung auch Funktionen direkt als *synchronized* deklarieren:

```
class MyMonitor {
    synchronized public void doSomethingCritical() {
        ...
    }
}
```

Entspricht semantisch dem Konstrukt

```
class MyMonitor {
    public void doSomethingCritical() {
        synchronized(this) {
            /* hier befinden wir uns im Monitor zur
               aktuellen Instanz von MyMonitor */
            ...
        }
    }
}
```

**Wichtig:** Welche Variablen durch den Monitor geschützt werden, kann nicht durch Sprachkonstrukte festgelegt werden. Solche Fehler werden deshalb **nicht zur Compilezeit** festgestellt.

### 2.2.2 *wait()*- und *notify()*-Konstrukte

Ein Thread, welcher sich in einem oder mehreren Monitoren befindet, benötigt auch eine Methode, um den Monitor wieder freizugeben, falls er im aktuellen Zustand des Monitors nicht weiterarbeiten kann. Ausserdem muss ein Thread den wartenden Threads mitteilen können, dass er den Zustand eines Monitors verändert hat, damit diese eventuell weiterarbeiten können. Zu diesem Zweck gibt es in Java die Methoden *wait()* und *notify()* bzw. *notifyAll()*.

Ist ein Thread nicht in der Lage, im aktuellen Zustand des Monitors seine Arbeit zu Ende zu bringen, so kann er den Monitor durch den Aufruf von *wait()* freigeben. Ändert ein Thread nun etwas am Zustand des Monitors, so dass eventuell andere Threads ihre Arbeit fortsetzen können, so benachrichtigt er die wartenden Threads per *notify()* oder *notifyAll()*.

```
public class Computer {

    /* Kritische Daten ... */
    private String os = "Windows 95";

    synchronized public void setOS(String os) {

        /* setze das neue OS */
        this.os = os;

        /* und benachrichtige wartende Threads
           ueber die Zustandsaenderung */
    }
}
```

```

notifyAll();

    /* Hier koennen noch beliebig viele
       kritische Funktionen ausgefuehrt
       werden. Die durch notifyAll()
       geweckten Threads duerfen den
       Monitor erst betreten, wenn der
       aktuelle Thread den Monitor
       verlassen hat. */
}

synchronized public void work() {

    /* solange kein echtes Betriebssystem laeuft... */
    while(os.compareTo("Windows 95") == 0) {
        try {

            /* Warte und gebe dabei den Monitor frei.
               Fuehrt jetzt ein anderer Thread setOS
               aus, so bricht wait ab. */
            wait();
        } catch(InterruptedException e) {}
    }
}
}
}

```

**Wichtig:** *wait()* und *notify()* sind in jedem Objekt vorhanden. Allerdings müssen beide Methoden in dem Objekt ausgeführt werden, in dessen Monitor sich der aktuelle Thread befindet. Wird eine dieser Funktionen außerhalb eines Monitors aufgerufen, wirft die Funktion eine **IllegalMonitorStateException** (muß nicht abgefangen werden).

### 2.2.3 Unterschiede zum traditionellen Monitor

Der Hauptunterschied zwischen dem Monitorkonzept in Java und der Standard-Definition des Monitors besteht darin, dass im Java-Monitor die logische Struktur des Programmes eine andere Form besitzt.

Im **Standard-Monitorkonzept** muss sich der Programmierer vor dem absetzen eines *notify()*-Signals davon überzeugen, dass der Monitor sich in einem gültigen Zustand befindet. Das bedeutet, dass alle auf der selben Condition-Variablen wartenden Programmteile **ohne Überprüfung des Zustandes** weiter arbeiten können. Beim Aufruf des *notify()* wird dann **direkt** der nächste wartende Thread im Monitor ausgeführt.

Im **Java-Monitorkonzept** muss jeder wartende Programmteil sich selbst davon überzeugen, dass die Voraussetzung für den folgenden Programmverlauf korrekt erfüllt sind. So können leicht andere Funktionen mit völlig neuen preconditions den Monitor betreten, ohne dass der Monitor geändert werden muss, was der objektorientierten Struktur des Programmes zugute kommt. Außerdem kann man auf diese Weise einfach auf das gleichzeitige Eintreten mehrerer Conditions warten, was im Standard-Monitorkonzept

nur durch das Einführen neuer Conition-Variablen mölgich ist. Das *notify()* führt in Java **nicht** zum direkten Verlassen des Monitors.

Java Monitor	Klassischer Monitor (Pseudo Code)
<b>Initialisierung</b>	
<pre>Object monitor = new Object(); boolean cond;  synchronized(monitor) {</pre>	<pre>Monitor monitor; Condition cond(monitor); Condition notCond(monitor); monitor.enter();</pre>
<b>Beim Verlassen des Monitors</b>	
<pre>    monitor.notifyAll();     /* weiter im Monitor */ } /* Ausserhalb des Monitors */</pre>	<pre>assert(cond.Precondition); cond.notify(); /* Falls noch kritische Operationen ausgeführt werden sollen, darf zuerst ein anderer Thread den Monitor betreten. */</pre>
<b>Warten auf den Monitor</b>	
<pre><b>Thread A</b> while(!cond) {     try {         monitor.wait();     } catch(InterruptedException e) {} } /* cond erfüllt */</pre>	<pre>cond.wait(); /* Bedingung von cond erfüllt */</pre>
<pre><b>Thread B</b> while(cond) {     try {         monitor.wait();     } catch(InterruptedException e) {} } /* !cond erfüllt */</pre>	<pre>notCond.wait(); /* Bedingung von notCond erfüllt */</pre>

## 2.3 Scheduling

### 2.3.1 Green Threads

Hier übernimmt die JVM das Thread-Handling selbst. Da oft mehrere Threads gleichzeitig rechenwillig sind, muss die Virtual Machine einen dieser Threads auswählen, um ihn auszuführen. Dieser wird durch das sogenannte **Fixed-Priority-Scheduling** bestimmt. Dabei erhält jeder Thread eine Priorität zwischen *MIN\_PRIORITY* und

*MAX\_PRIORITY*. Aus allen rechenwilligen Threads wird der mit der höchsten Priorität ausgewählt. Sind zwei Threads mit gleicher Priorität rechenwillig, so kann die VM einen beliebigen dieser Threads ausführen.

Die Referenz-Implementierung der JVM realisiert noch interessante zusätzliche Eigenschaften. Auf diese sollte man sich jedoch beim Programmieren **auf keinen Fall** verlassen, da die maschinenunabhängigkeit sonst nicht mehr gewährleistet ist. Zwei „Features“ der Referenz-Implementierung werden im Folgenden kurz vorgestellt:

- **Round-Robin Verfahren**

Sobald ein Thread den Zustans “laufend” verlässt, wird er an das Ende der Thread-Liste dieser Priorität angehängt. Implementiert eine JVM dieses Verfahren, so kann man durch `yield()` Aufrufe `time-sliceing` simulieren.

**Vorsicht!** Der Code ist dann nicht mehr maschinenunabhängig.

- **Prioritäts-Vererbung**

Wenn ein Thread A mit Priorität 8 auf einen Thread B mit Priorität 3 wartet, dann hat A faktisch eine Priorität von 3 (Prioritäts- Inversion). Die JVM kann B nun kurzfristig die selbe Priorität wie A geben. Dieses Verfahren heißt Prioritäts-Vererbung.

### 2.3.2 Native Threads

Die JVM überlässt das Scheduling dem Betriebssystem. Dadurch ergeben sich eine Menge von Problemen (siehe auch 3.3).

## 3 Probleme und Lösungen

Auch wenn das Thread-Design von Java in vielerlei Hinsicht sehr schön gestaltet ist, gibt es doch einige Probleme, über die sich jeder Programmierer im Klaren sein sollte, bevor er sich an Threads heranwagt. Mißachtet man ein paar grundlegene Eigenschaften der Java-Threads, so hat man schnell völlig maschinenabhängigen Code in Java “verbrochen”.

### 3.1 Lokale Variablen in Threads

Der VM ist es gestattet, in jedem Thread lokale Kopien der Klassen-Variablen zu halten. Soll aus einem anderen Thread auf diese Klassen-Variablen zugegriffen werden (z.B. **busy-waiting**), so muss diese Variable als *volatile* deklariert werden.

### 3.2 Deprecated Methods

Es gibt in der Klasse Thread einige Methoden, die mit der Version 1.2 der JDK “mißbilligt” wurden. Das heißt, dass diese Methoden zwar aus Kompatibilitätsgründen noch vorhanden sind, jedoch nicht mehr verwendet werden sollten. Hier eine kurze Auflistung:

- *suspend()*

Threads, die durch den Aufruf dieser Funktion in den Schlaf versetzt werden, behalten alle ihre Locks. Dadurch wird zwar eine Konsistenz der Daten gewährleistet, jedoch verursacht ein *suspend()*-Aufruf leicht Deadlocks.

- *stop()*  
Ein gestoppter Thread verliert sofort alle seine Locks. Dadurch kann es sein, dass Daten inkonsistent bleiben. Eine Alternativlösung ist, dass ein Thread nur so lange läuft, bis eine Terminationsbedingung eingetreten ist.

```
class StopableThread extends Thread {
    private volatile boolean terminated = false;

    public void terminate() {
        terminated = true;

        /* eventuell hier warten, bis der Thread
           wirklich tot ist; es kann auch
           vorkommen, dass man noch in einem
           wait() steckende Teile des Threads
           ein notify() senden muss */
    }

    public void run() {
        while(!terminated) {

            // tu deine Arbeit!

        }
    }
}
```

- *notify()*  
Diese Methode ist zwar nicht deprecated, aber man sollte sie dennoch nicht verwenden, da sie unvorhersehbare Ergebnisse liefert. In allen mir zugänglichen JVM's wurden alle wartenden Threads geweckt. Bei native Threads auf Solaris waren es ohne erkennbaren Zusammenhang mal mehr und mal weniger.

### 3.3 Priority Mapping

Wenn das Betriebssystem weniger Prioritäten unterstützt als Java gerne hätte, so werden mehrere Java-Prioritäten auf die selbe Betriebssystem-Priorität abgebildet. Man sollte also versuchen, seine Programme nicht zu sehr von Prioritätsunterschieden abhängig zu gestalten. Allerdings findet man so ein Betriebssystem wohl äußerst selten.

### 3.4 Starvation

Die JVM Referenz-Implementation benutzt kein time-slicing. Das bedeutet, dass ein Thread nicht unterbrochen wird, so lange er nicht wartet (auf IO oder einen Monitor). Dies ist besonders bei Anfängern der Java-Thread programmierung ein häufiger Fehler, der leicht durch `sleep()`-Anweisungen behoben werden kann.

---

## 4 Zusammenfassung

Angesichts der komplexen Materie ist in Java eine recht programmierfreundliche low-level Lösung gelungen. Dank der objektorientierten Sprachkonstrukte werden in Zukunft wohl auch entsprechende high-level Lösungen für Java entwickelt werden. Ohne diese Erweiterungen gestaltet sich die multithreaded-Programmierung in Java noch fehleranfällig und umständlich.

---

## A Kurzreferenz - Die wichtigsten Klassen

### A.1 Die Klasse *Thread*

Dieser Abschnitt stellt eine Kurzreferenz mit den wichtigsten Informationen zur Thread-Programmierung dar.

#### A.1.1 Konstruktoren

- **Thread(ThreadGroup *group*, Runnable *run*, String *name*)**  
Erstellt ein Thread-Objekt mit Namen *name*, welches das *run*-Objekt als ausführbare Komponente enthält und zur Gruppe *group* gehört. Alle Argumente sind optional.

#### A.1.2 Variablen

- **MAX\_PRIORITY**  
Die maximale Priorität, die einem Thread zugewiesen werden kann.
- **MIN\_PRIORITY**  
Die minimale Priorität, die einem Thread zugewiesen werden kann.
- **NORM\_PRIORITY**  
Die Standard-Priorität.

#### A.1.3 Methoden

- **String getName()/ setName(String)**  
Liest/ Setzt den Namen des Threads.
- **int getPriority()/ setPriority(int)**  
Zum auslesen/ setzten der Priorität.
- **boolean isDaemon()/ setDaemon(boolean)**  
Liest/ Setzt ob der Thread ein Daemon-Thread ist.
- **void join() throws InterruptedException**  
Wartet, bis der Thread terminiert.
- **void join(long *milli*) throws InterruptedException**  
**void join(long *milli*, int *nano*) throws InterruptedException**  
Wartet maximal *milli* Millisekunden und *nano* Nanosekunden bis der Thread terminiert.
- **static void sleep(long *milli*)**  
**sleep(long *milli*, int *nano*)**  
Hält diesen Thread die angegebene Zeit an. Allerdings ist die zeitliche Auflösung je nach Betriebssystem und JVM-Implementierung unterschiedlich.

- **static void yield()**  
Der aktuelle Thread wird angehalten, um anderen Threads die Möglichkeit zur Ausführung zu geben.
- **static Thread currentThread()**  
Liefert den gerade laufenden Thread.

## A.2 Die Klasse *Object*

### Thread-spezifische Methoden

- **wait() throws InterruptedException**  
Wartet im aktuellen Monitor. Somit können ihn andere, wartende Threds betreten. Der thread schläft nun, bis ein anderer Thread *notify()* oder *notifyAll()* ausführt. Danach reiht sich der aktuelle Thread in die Schlange der wartenden Threads ein.
- **notify()**  
Weckt einen Thread, der durch *wait()* im aktuellen Monitor schläft. Der geweckte Thread wird erst gestartet, wenn der aktuelle Thread den Monitor verlässt.
- **notifyAll()**  
Weckt alle Threads, die durch *wait()* im aktuellen Monitor schlafen. Diese konkurrieren alle gleichberechtigt um den Monitor.

## Literatur

- [1] Doug Lea. *Concurrent Programming in Java*. Addison-Wesley, 1999.
- [2] Tim Lindholm and Frank Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, 1997. Spezifiziert Java 1.0.2.
- [3] Scott Oaks and Henry Wang. *Java Threads*. O'Reilly, 1999.
- [4] Sun. Java 2 platform api specification. <http://www.db.fmi.uni-passau.de/local/docs/-jdk1.2/api/index.html>.
- [5] Sun. The java tutorial - threads. <http://www.db.fmi.uni-passau.de/local/docs/-tutorial/essential/threads/index.html>.