Universität Osnabrück

VL 20
1

Informatik B - Objektorientierte Programmierung in Java

Vorlesung 20:

Threads

Inhalt

- Threads
  - Ein Beispiel zur Motivation
  - Parallelität, Nebenläufigkeit, Prozess, Thread
  - Thread vs. Prozess
  - Typische Einsatzgebiete von Threads
  - Erzeugung von Threads
  - Beendung von Threads
  - Das Interface **`Runnable`**
  - Synchronisation

Siehe: Krüger: Handbuch der Javaprogrammierung, www.javabuch.de, Kapitel 22.

Motivation

- Führt ein Applet eine lange Berechnung aus, so erfolgt keine Reaktion auf die Betätigung des Stop-Buttons am Browser bzw. Appletviewer.

Ein Beispiel zur Motivation

```
import java.applet.*;
import java.awt.*;

public class DemoOhneThreads extends Applet {
  String s;
  public void start() {
   long n;
    for ( int j=1; j<1000; j++) {
      for ( int i=1; i<1000; i++) {
        for ( int k=1; k<1000; k++) {
          n =  k + i + j;
        } }
      if ((j % 100) == 0) {System.out.print(j+" "); System.out.flush();}
    }
    System.out.println("start: finished");
  }
  public void paint( Graphics g ) {
   String s;
    g.setColor( Color.blue );
    s = new String( "paint: done" );
    g.drawString( s, 10, 20);
  }
}
```

Die laufende Berechnung ist nicht unterbrechbar, weil die Programmkontrolle nicht abgegeben wird.

Nebenläufigkeit

- Java hat das Konzept der *Nebenläufigkeit* innerhalb der Sprache implementiert. Mit Nebenläufigkeit bezeichnet man die Eigenschaft eines Systems, zwei oder mehr Vorgänge gleichzeitig oder quasi-gleichzeitig ausführen zu können.

Prozess

- Ein Prozess bezeichnet ein Programm in Ausführung mit seinen dafür notwendigen betriebssystemseitigen Datenstrukturen und zugeordneten Eigenschaften (z.B. Stack- und Programmzähler, Prozesszustand sowie Eigenschaften der Speicher- und Dateiverwaltung).
- Ein Prozess wird durch das Betriebssystem als eigenständige Instanz - in der Regel unabhängig und geschützt von anderen - ausgeführt. Multitaskingsysteme (wie UNIX und neuere Generationen der Windows-Familie) gestatten die (quasi-)parallele Prozessausführung.

Thread

- Ein Thread (engl. für *Faden*) stellt eine nebenläufige Ausführungseinheit innerhalb genau eines Prozesses dar, die parallel zu anderen Threads laufen kann.
- Ein Thread ähnelt damit einem *Prozess*, arbeitet aber auf einer feineren Ebene.

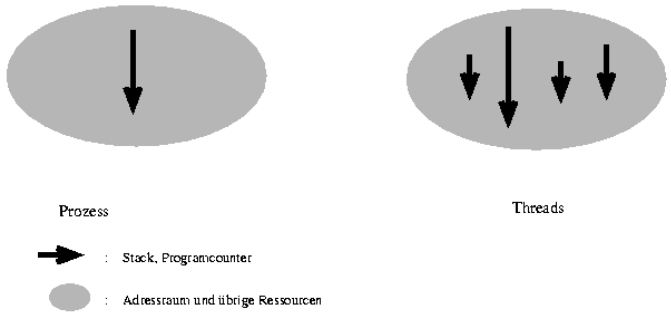
Bemerkungen

- Threads haben viele der prozesstypischen Eigenschaften, wie Zustand, Programmzähler etc. Den Hauptunterschied zu voll entwickelten Prozessen bildet der zwischen allen Threads eines Prozesses geteilte Speicher. Gleichzeitig besitzt jeder Thread seinen eigenen lokalen Speicherbereich in dem u.a. die lokalen Variablen verwaltet werden. Aus diesen Gründen werden Threads oft auch als leichtgewichtige Prozesse bezeichnet.
  Alle Threads bewegen sich im Ausführungskontext des erzeugenden Prozesses.
- In Multithreading-Systemen besitzt jeder ablaufende Prozess mindestens einen Thread, der den Kontrollfluss realisiert.

Threads

- Ein Thread hat gewisse Ähnlichkeit mit einem Prozess:
  - er kann vom Scheduler angehalten und fortgesetzt werden
  - er kann ohne CPU-Belastung auf Eingabe warten:
    - der Thread wird gestoppt
  - wenn die Eingabe kommt, wird er vom Scheduler fortgesetzt
- Mit Hilfe des Threads wird ermöglicht, dass andere Anwendungsteile reagieren können:
  - in regelmäßigen Abständen stoppt der Scheduler einen Thread und setzt einen anderen Prozess (Thread) fort
  - nach einer gewissen Zeit wird der gestoppte Thread wieder aktiviert

Zusammenhang zwischen Prozess und Thread



- Der Prozess wird in der Regel erst mit dem Ende des letzten Threads beendet.

Threads als Teil von Prozessen

- Threads sind in Prozesse eingebettet, d.h. Teile von Prozessen.
- Jeder Prozess besteht aus mindestens einem Thread.
- Threads eines Prozesses besitzen einen gemeinsamen Speicherbereich.
- Dies führt zu
  - leichter Kommunikation zwischen den Threads *eines* Prozesses
  - Koordinierungsprobleme

Typische Einsatzgebiete von Threads

- Bevor auf Details eingegangen wird, hier zunächst typische Einsatzgebiete von Threads:
- zur Vermeidung von Blockaden der Benutzerschnittstelle; dann kann eine Anwendung weiterhin auf Benutzereingaben reagieren
- um nach Ablauf einer bestimmten Zeit ein Ereignis auszulösen. Beispiel: eine Bilderfolge in einem Applet, die im Sekundentakt verändert wird
- zur Parallelisierung einer Anwendung auf mehrere Prozessoren; dann kann in Mehrprozessorsystemen für jeden Prozessor ein Thread gestartet werden; der parallel verarbeitbare Programmteil wird in einen Thread verpackt
- zur Bedienung von blockierenden Schnittstellen. Beispiel: Warten auf Netzwerkverbindungen (siehe kommende Vorlesung)

Zu klärende Probleme

- Wie werden Threads vereinbart?
- Wie werden Threads gestartet?
- Wie werden Threads gestoppt?
- Wie erfolgt die Koordinierung?

Erzeugung von Threads

- Threads können auf zwei Weisen erzeugt werden:
  - durch Implementierung des Interfaces **`java.lang.Runnable`**
  - durch Vererbung von der Klasse **`Thread`** (die das Interface **`java.lang.Runnable`** implementiert)
- Die Methode **`run()`** (aus dem Interface **`Runnable`**) muss implementiert werden.
- **`run()`** enthält die im Thread auszuführenden Anweisungen.

Erzeugung von Threads

- **`Thread`** stellt die Basismethoden zur Erzeugung, Kontrolle und zum Beenden von Threads zur Verfügung. Um einen konkreten Thread zu erzeugen, muss eine eigene Klasse aus **`Thread`** abgeleitet und die Methode **`run`** überlagert werden.
- Mit Hilfe eines Aufrufs der Methode **`start`** wird der Thread gestartet und die weitere Ausführung an die Methode **`run`** übertragen. **`start`** wird nach dem Starten des Threads beendet, und der Aufrufer kann parallel zum neu erzeugten Thread fortfahren.

Beispiel: Erzeugung von Threads

- Listing2201.java
- Das Beispiel zeigt einen einfachen Thread, der in einer Endlosschleife einen Zahlenwert hochzählt.
- Die Methode **`run`** sollte vom Programm niemals direkt aufgerufen werden. Um einen Thread zu starten, ist immer **`start`** aufzurufen.

Abbrechen eines Threads

- Ein Thread wird dadurch beendet, dass das Ende seiner **`run`**-Methode erreicht ist.
- In manchen Fällen ist es jedoch erforderlich, den Thread von außen abzubrechen.
- Bis JDK 1.1 erfolgte dies durch die Methode **`stop`**, die aber inzwischen deprecated ist.

Beispiel: Abbrechen mit stop

- Listing2202.java
- Gleichzeitig zum Thread fährt das Hauptprogramm mit dem Aufruf der **`sleep`**-Methode und dem Aufruf von **`stop`** fort. Beide Programmteile laufen also parallel ab.
- **`stop`** sollte aber nicht benutzt werden, denn es ist nicht voraussagbar und auch nicht definiert, an welcher Stelle ein Thread unterbrochen wird, wenn ein Aufruf von **`stop`** erfolgt.
- Besser: **`interrupt`**

Beispiel: Abbrechen mit interrupt

- Listing2203.java
- Der Thread gibt ununterbrochen Textzeilen auf dem Bildschirm aus. Das Hauptprogramm soll den Thread erzeugen und nach 2 Sekunden durch einen Aufruf von **`interrupt`** eine Unterbrechungsanforderung erzeugen. Der Thread soll dann die aktuelle Zeile fertig ausgeben und anschließend terminieren.
- Durch Aufruf von **`interrupt`** wird ein Flag gesetzt, das eine Unterbrechungsanforderung signalisiert. Durch Aufruf von **`isInterrupted`** kann der Thread feststellen, ob das Abbruchflag gesetzt wurde und der Thread beendet werden soll.

Sleep interrupted

- Erfolgt der Aufruf von **`interrupt`** während des Aufrufs von **`sleep`**, wird **`sleep`** mit einer **`InterruptedException`** abgebrochen (auch wenn die geforderte Zeitspanne noch nicht vollständig verstrichen ist). Wichtig ist hier, dass das Abbruchflag zurückgesetzt wird und der Aufruf von **`interrupt`** somit eigentlich verlorenginge, wenn er nicht direkt in der **`catch`**-Klausel behandelt würde. Wir rufen daher innerhalb der **`catch`**-Klausel **`interrupt`** erneut auf, um das Flag wieder zu setzen und **`run`** die Abbruchanforderung zu signalisieren.

Weitere Methoden von Thread: sleep

- **`public static void sleep(long millis) throws InterruptedException`**
- **`public static void sleep(long millis, int nanos) throws InterruptedException`**
- sorgt dafür, dass der aktuelle Thread für die (in Millisekunden und Nanosekunden angegebene) Zeit unterbrochen wird
- **`Thread.sleep`** kann während der Wartezeit eine Ausnahme vom Typ **`InterruptedException`** auslösen.

Weitere Methoden von Thread: join

- Mit der Methode

```
public void join()
throws InterruptedException

public void join(long millis)
throws InterruptedException
```

kann auf das Ende des entsprechenden Threads gewartet werden (ggf. maximale Wartezeit).

- „join“ = „vereinen“

Beispiel für join

```
public class ThreadJoin extends Thread {

  public static void main( String args[] ) {
    ThreadJoin thJ = new ThreadJoin();
    System.out.println( "Start des Threads" );
    thJ.start();
    try {
      thJ.join();
    } catch (InterruptedException e) { }
    System.out.println( "Main: Fertig" );
  }
  public void run() {
    int i = 0;
    while ( i < 10 ) {
      System.out.println( "Thread:   " + i++ );
    }
  }
}
```


Das Interface `Runnable`

- Nicht immer ist es möglich oder sinnvoll, eine Klasse, deren Code in einem Thread laufen soll, von **`Thread`** abzuleiten.
- Ausweg: Implementierung des Interfaces **`Runnable`**. **`Runnable`** enthält nur eine einzige Deklaration, nämlich die der Methode **`run`**:

- **`public abstract void run()`**

Implementieren von Runnable

- Um Code in einer nicht von **`Thread`** abgeleitete Klasse in dieser Weise als Thread laufen zu lassen, ist in folgenden Schritten vorzugehen:
- Zunächst wird ein neues **`Thread`**-Objekt erzeugt.
- An den Konstruktor wird das Objekt übergeben, deren **`run`**-Methode parallel ausgeführt werden soll.
- Die Methode **`start`** des neuen **`Thread`**-Objekts wird aufgerufen.
- Nun startet das **`Thread`**-Objekt die **`run`**-Methode des übergebenen Objekts, das sie ja durch die Übergabe im Konstruktor kennt.

Beispiel

- Listing2204.java
- Die Klasse B2204 ist von A2204 abgeleitet und kann daher nicht von **`Thread`** abgeleitet sein. Statt dessen implementiert sie das Interface **`Runnable`**. Um nun die **`run`**-Methode der Klasse B2204 als Thread auszuführen, wird in **`main`** von Listing2204 eine Instanz dieser Klasse erzeugt und an den Konstruktor der Klasse **`Thread`** übergeben. Nach dem Aufruf von **`start`** wird die **`run`**-Methode von B2204 ausgeführt.

Synchronisation

- Wenn man sich mit Nebenläufigkeit beschäftigt, muss man sich in aller Regel auch mit Fragen der *Synchronisation* nebenläufiger Prozesse beschäftigen. In Java erfolgt die Kommunikation zweier Threads auf der Basis gemeinsamer Variablen, die von beiden Threads erreicht werden können. Führen beide Threads Änderungen auf den gemeinsamen Daten durch, so müssen sie synchronisiert werden, denn andernfalls können undefinierte Ergebnisse entstehen.

Synchronisation

- Beispiel: Listing2209.java
- Zwei Threads zählen einen gemeinsamen Zähler hoch.
- Mögliche Ausgabe:

```
0
1
…
31
33  <-- Nanu? Wo ist die 32?
34
…
56
32  <-- Ach so, hier!
…
```

Synchronisation

- Erläuterung Beispiel: Beide Prozesse greifen unsynchronisiert auf die gemeinsame Klassenvariable **`cnt`** zu. Da die Operation **`System.out.println(cnt++);`** unterbrechbar (also nicht *atomar)* ist, kommt es zu dem Fall, dass die Operation mitten in der Ausführung unterbrochen wird und der Scheduler mit dem anderen Thread fortfährt. Erst später, wenn der unterbrochene Thread wieder Rechenzeit erhält, kann er seinen vor der Unterbrechung errechneten Zählerwert von 32 ausgeben. Sein Pendant war in der Zwischenzeit allerdings bereits bis 56 fortgefahren. Um diese Art von Inkonsistenzen zu beseitigen, bedarf es der *Synchronisation* der beteiligten Prozesse.

Monitore

- Zur Synchronisation nebenläufiger Prozesse hat Java das Konzept des *Monitors* implementiert. Ein Monitor ist die Kapselung eines *kritischen Bereichs* (also eines Programmteils, der nur von jeweils einem Prozess zur Zeit durchlaufen werden darf) mit Hilfe einer automatisch verwalteten Sperre. Diese Sperre wird beim Betreten des Monitors gesetzt und beim Verlassen wieder zurückgenommen. Ist sie beim Eintritt in den Monitor bereits von einem anderen Prozess gesetzt, so muss der aktuelle Prozess warten, bis der Konkurrent die Sperre freigegeben und den Monitor verlassen hat.

Monitore

- Durch **`synchronized`** kann entweder eine komplette Methode oder ein Block innerhalb einer Methode geschützt werden. Der Eintritt in den so deklarierten Block wird durch das Setzen einer Sperre auf einer Objektvariablen erreicht. Bezieht sich **`synchronized`** auf eine komplette Methode, wird als Sperre die **`this`**-Referenz verwendet, andernfalls ist eine Objektreferenz explizit anzugeben.
- Wird eine statische Methode synchronisiert, erfolgt die Sperre mit Hilfe des Klassenobjekts.

Anwendung von `synchronized` auf einen Block von Anweisungen

- Beispiel: Listing2210.java:

```
public class Listing2210 extends Thread
{
  static int cnt = 0;
  public static void main(String[] args)
  {
    Thread t1 = new Listing2210();
    Thread t2 = new Listing2210();
    t1.start();
    t2.start();
  }
  public void run()
  {
    while (true) {
      synchronized (getClass()) {
        System.out.println(cnt++);
}}}}
```

Warum sollte hier nicht **`this`** stehen?

Anwendung von synchronized auf einen Block von Anweisungen

- Beispiel: Listing2210.java
- Synchronisation durch **`synchronized`**-Block auf der Variablen **`this`** funktioniert leider nicht. Da die Referenz **`this`** für jeden der beiden Threads, die ja unterschiedliche Instanzen repräsentieren, neu vergeben wird, wäre für jeden Thread der Eintritt in den Monitor grundsätzlich erlaubt.
- Statt dessen verwenden wir die Methode **`getClass`**, die uns ein Klassenobjekt beschafft (ein und dasselbe für alle Instanzen derselben Klasse), mit dem wir die Klassenvariable **`cnt`** schützen können.
- Nun werden alle Zählerwerte in aufsteigender Reihenfolge ausgegeben.

Anwendung von synchronized auf eine Methode

- Synchronisation des Zugriffs auf ein Objekt selbst, weil damit zu rechnen ist, dass mehr als ein Thread zur gleichen Zeit das Objekt verwenden will.
- Das Beispiel enthält ein Zählerobjekt, dessen Aufgabe es ist, einen internen Zähler zu kapseln, auf Anforderung den aktuellen Zählerstand zu liefern und den internen Zähler zu inkrementieren.
- Im Beispiel wird der Zähler von fünf Threads verwendet. Die Langsamkeit und damit die Wahrscheinlichkeit, dass der Scheduler die Zugriffsoperation unterbricht, wird in unserem Beispiel durch eine Sequenz eingestreuter Fließkommaoperationen erhöht.

Anwendung von synchronized auf eine Methode

- Beispiel: Listing2211.java
- Problem: Es werden doppelte Schlüssel produziert, z.B.

```
10 for Thread-2
11 for Thread-4
10 for Thread-0
10 for Thread-1
11 for Thread-2
11 for Thread-3
12 for Thread-4
13 for Thread-0
```


Anwendung von synchronized auf eine Methode

- Lösung:
  Eine einfache Markierung der Methode **`nextNumber`** als **`synchronized`** macht diese zu einem synchronisierten Block und sorgt dafür, dass der komplette Code innerhalb der Methode wie ein synchronisierter Block behandelt wird. Ein Betreten des kritischen Abschnitts durch einen zweiten Thread ist dann nicht mehr möglich:

```
public synchronized int nextNumber()
{...
```

Anwendung von `synchronized` auf eine Methode: Erläuterung

- Durch das **`synchronized`**-Attribut wird beim Aufruf der Methode die Instanzvariable **`this`** des **`Counter2211`**-Objekts gesperrt und damit der Zutritt für andere Threads unmöglich gemacht. Erst nach Verlassen der Methode und Entsperren von **`this`** kann **`nextNumber`** wieder von anderen Threads aufgerufen werden.
- Diese Art des Zugriffschutzes wird in Java von vielen Klassen verwendet, um ihre Methoden *thread-sicher* zu machen. Im Einzelfall ist die Java-Dokumentation zu Rate zu ziehen.

Ausblick

- Threads besitzen auch eine Reihe administrativer Eigenschaften, die besonders dann nützlich sind, wenn das Thread-Konzept stark genutzt wird.
  - Zum einen gibt es Eigenschaften, die bei den Threads selbst zu finden sind, beispielsweise die *Priorität* oder der *Name* eines Threads. Mit Hilfe der Methode **`getName`** kann der Name eines Threads abgefragt werden: **`public String getName()`**
  - *Thread-Gruppen* dienen dazu, Informationen zu verwalten, die nicht nur für einen einzelnen Thread von Bedeutung sind, sondern für eine ganze Gruppe.