Java I – Vorlesung 10
Nebenläufigkeit

28.6.2004

Threads
Synchronisation
Deadlocks
Thread-Kommunikation

Innere Klassen
Anonyme Klassen

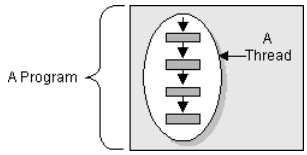
Nebenläufigkeit

`http://java.sun.com/docs/books/tutorial/essential/threads/`

- Manchmal möchte man mehrere Berechnungen gleichzeitig ablaufen lassen:
  - ein Server kommuniziert mit mehreren Clients gleichzeitig
  - Programm führt eine teure Berechnung aus, will aber weiter auf Eingaben reagieren
  - Roboter verarbeitet Sensor-Eingaben
  - Verarbeitung von Klicks auf GUI
- Konzeptuelle Gleichzeitigkeit von Berechnungen heißt Nebenläufigkeit (concurrency).

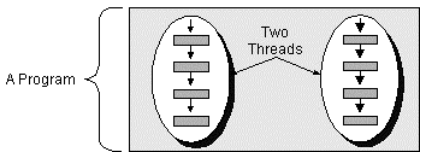
Sequentielle Verarbeitung

- Normalerweise werden Programme sequentiell abgearbeitet: Man fängt vorne an, führt einen einzigen Pfad durch das Programm aus, und hört auf.



Nebenläufige Verarbeitung

- Java unterstützt den Ablauf von nebenläufigen Threads: Konzeptionell laufen verschiedene Berechnungen gleichzeitig ab.



JVM schaltet natürlich in Wirklichkeit schnell zwischen Threads hin und her (scheduling). Keine Kontrolle über Reihenfolge!

Beispiel: Nebenläufigkeit

```
class Primzahlen {
  static int[] faktorisieren(int zahl) {
    // ... berechne Primfaktoren von zahl ...
  }
}

...

public static void main(String[] args) {
  do {
    int zahl = liesZahl();
    int[] prim = faktorisieren(zahl); // TEUER
  } while( zahl > 0 );
}
```


Beispiel: Nebenläufigkeit

```
class Primzahlen extends Thread {
  private int zahl;  // vom Konstruktor gesetzt

  public void run() {
    int[] prim = faktorisieren(zahl);
    System.out.println(prim);
  }
}

do {
  int zahl = liesZahl();
  Thread t = new Primzahlen(zahl);
  t.start();
} while( zahl > 0 );
```

Threads erzeugen

- Benutzer kann neue Threads anstoßen:
  - Objekt `obj` von Klasse Thread (oder abgeleiteter) erzeugen
  - Aufruf von `obj.start()` erzeugt neuen Thread, in dem `obj.run()` ausgeführt wird.
- Thread endet, sobald Methode run() verlassen wird.

Beispiel: Threads laufen "gleichzeitig"

```
class Counter extends Thread {
  private String name; // von Konstr. gesetzt

  public void run() {
    for( int i = 1; i <= 20; i++ )
      System.err.println(name + ": " + i);
  }
}

...

new Counter("counter1").start();
new Counter("counter2").start();
```

Ablauf des Programms

- Jedes Programm hat einen Main-Thread, in dem die Methode main ausgeführt wird. Dieser Thread endet, sobald main() endet.
- Benutzer kann neue Threads erzeugen.
- Programm endet unter einer der beiden folgenden Bedingungen:
  - Aufruf von System.exit()
  - Alle Threads, die keine Dämonen sind (der Default), sind beendet.

Synchronisation

- Manchmal reicht es, wenn man einen Thread einfach anstößt und laufen lässt.
- Häufig muss man Threads aber synchronisieren:
  - Sicherstellen, dass Daten nicht inkonsistent verändert werden
  - Ein Thread wartet auf Ereignisse aus einem anderen Thread
  - Austausch von Nachrichten

Beispiel: Synchronisation nötig

```
class Quadrate {
   public int zahl, quadratDerZahl;

   public void set(int zahl) {
     this.zahl = zahl;
     quadratDerZahl = zahl * zahl;
   }
}

class QuadratSetzer extends Thread {
  public void run() {
    for( int i = 1; i <= 20; i++ )
      q.set(i);
  }
}

Quadrate q = new Quadrate();
new QuadratSetzer(q).start();
int z = q.zahl, qdz = q.quadratDerZahl;
```


Beispiel: Synchronisation in Wirklichkeit

```
class InputThread extends Thread {
  private List<String> queue;

  public void run() {
    String line = someReader.readLine();
    queue.add(line);
  }
}

...

List<String> queue = new ArrayList<String>();
new InputThread(queue, someReader1).start();
new InputThread(queue, someReader2).start();

while(queue.size() > 0 ) {
  String next = queue.iterator().next();
  ...
}
```

Beispiel: Synchronisation nötig

```
class Quadrate {
   public int zahl, quadratDerZahl;

   public synchronized void set(int zahl) {
     this.zahl = zahl;
     quadratDerZahl = zahl * zahl;
   }
}
class QuadratSetzer extends Thread {
  public void run() {
    for( int i = 1; i <= 20; i++ )
      q.set(i);
  }
}

Quadrate q = new Quadrate();
new QuadratSetzer(q).start();

synchronized(q) {
  int z = q.zahl, qdz = q.quadratDerZahl;
}
```


Monitore

- Jedes Objekt besitzt einen eigenen Monitor (oder "mutex"), der zu jedem Zeitpunkt von höchstens einem Thread gehalten werden kann.
- Mit dem Schlüsselwort synchronized kann ein Thread versuchen, den Monitor eines Objekts zu nehmen.
- Wenn schon ein anderer Thread den Monitor hält, wird der Thread so lange angehalten, bis der Monitor wieder frei ist.

Das Schlüsselwort "synchronized"

- Ein Block kann mit synchronized umfasst werden:

```
synchronized(obj) {
  ... wird erst ausgeführt, wenn
  Thread den Monitor von obj hat ...
}
```

- Eine Methode kann mit synchronized markiert werden. Dies entspricht Synchronisation auf Objekt bzw. Klassen-Objekt, zu dem Methodenaufruf gehört.

Synchronisierte Collections

- Collections sind aus Effizienzgründen nicht thread-safe, d.h. Methoden sind nicht synchronisiert.
- Es gibt Methoden in java.util.Collections, die synchronisierte Collection-Objekte erzeugen.
- Iteration muss trotzdem extra synchronisiert werden: Nicht nebenläufig während Iteration neue Einträge einfügen!

Beispiel: Synchronisation in Wirklichkeit

```
class InputThread extends Thread {
  private List<String> queue;

  public void run() {
    String line = someReader.readLine();
    queue.add(line);
  }
}

List<String> queue =
  Collections.synchronizedList(new ArrayList<String>());
new InputThread(queue, someReader1).start();
new InputThread(queue, someReader2).start();

do
  synchronized(queue) {
    for( String str : queue )  ..... // oder wie oben
  }
while( queue.size() > 0 );
```


Deadlocks

- Jedes nebenläufige Programm ist in ständiger Gefahr, Deadlocks zu enthalten.
- Deadlock entsteht, wenn zwei Threads zwei Monitore A und B nehmen wollen:
  - Thread 1 nimmt Monitor A
  - Thread 2 nimmt Monitor B
  - Thread 1 kann jetzt B nicht nehmen
  - Thread 2 kann jetzt A nicht nehmen
  - Programm hängt

Beispiel: Deadlocks

```
class DeadlockThread extends Thread {
  private String mon1, mon2;

  public void run() {
    synchronized(mon1) {
      System.err.println("Habe Monitor " + mon1);

      synchronized(mon2) {
        System.err.println("Habe Monitor " + mon2);
      }
    }
  }
}

String m1 = "foo", m2 = "bar";
new DeadlockThread(m1, m2).start();
new DeadlockThread(m2, m1).start();
```


Deadlocks auflösen

- Beim nebenläufigen Programmieren muss man sorgfältig ausschließen, dass Deadlocks auftreten können.
- Deadlocks treten nur auf, wenn Threads die Monitore in verschiedener Reihenfolge nehmen wollen.
- Eine (unter vielen) mögliche Strategie zur Vermeidung: Sicherstellen, dass alle die Monitore in gleicher Reihenfolge nehmen.

Wait und Notify

- Klasse Object definiert die Methoden wait() und notifyAll():
  - Muss Monitor von obj haben, um wait() oder notifyAll() aufrufen zu können.
  - Aufruf von obj.wait() gibt Monitor ab und suspendiert den aktuellen Thread
  - Aufruf von obj.notifyAll() weckt alle Threads auf, die auf obj suspendiert sind. Threads versuchen als erstes, Monitor wieder zu nehmen.
- Zweck: Andere Threads über Ereignisse informieren.

Beispiel: Wait und Notify

- Threads 1, 2:

```
try {
  synchronized(obj) {
    obj.wait();
  }
} catch(InterruptedException e) { }
```

- Thread 3:

```
synchronized(obj) {
  obj.notifyAll();
}
```

Abbrechen von Threads

- Um einem Thread zu signalisieren, dass er sich beenden soll, ruft man seine interrupt()-Methode auf.
- Im Thread wird dann
  - eine Exception geworfen: InterruptedException in wait() und sleep(), InterruptedIOException in IO-Methoden.
  - Zustand geändert, so dass Methode isInterrupted() true zurückgibt

Pipes

- Man kann größere Mengen von Daten zwischen Threads über Pipes austauschen.
- Klassen PipedInput/OutputStream, PipedReader/Writer in java.io.
- Man erzeugt einen PipedWriter und einen PipedReader als Paar; was man in den einen reinschreibt, kommt aus dem anderen heraus.

Verschachtelte Klassen (und Interfaces)

- Genau wie Felder und Methoden können Klassen und Interfaces Members anderer Klassen sein (statisch oder nichtstatisch).
- Kann Klassen sogar lokal innerhalb einer Methode deklarieren.
- Solche Klassen und Interfaces heißen verschachtelt.
- Verschachtelte Klassen dürfen auf private Members der umschließenden Klasse zugreifen.

Innere Klassen

- Eine Klasse heißt innere Klasse einer anderen, wenn sie
  - ein nichtstatisches Member ist
  - lokal in einer Methode deklariert
  - anonyme Klassen (siehe später)
- Objekte der inneren Klasse gehören konzeptuell zu Objekten der umfassenden dazu (machen ohne keinen Sinn).
- Zugriff auf alle Members der äußeren Klasse.

Beispiel: Innere Klassen

```
class ManyReaders {
  private List<String> queue;

  private class InputThread extends Thread {
    public void run() {
      String line = someReader.readLine();
      queue.add(line);
    }
  }

  public void foo() {
    queue = new ArrayList<String>();
    new InputThread(someReader1).start();
    new InputThread(someReader2).start();
    ...
  }
}
```


Verschachtelte Klassen: Einschränkungen

- Innere Klassen dürfen keine statischen Members haben.
- Statische Member-Klassen dürfen nicht auf nichtstatische Members der umschließenden Klasse zugreifen.
- Lokale Klassen (in Methoden definiert) dürfen nur auf lokale Variablen zugreifen, die als `final` definiert sind.

Anonyme Klassen

- Manchmal braucht man nur ein einziges kleines Objekt von einer Klasse; es lohnt sich fast nicht, dafür extra eine Klasse zu definieren.
- Dafür gibt es anonyme Klassen.
- In einem einzigen `new`-Ausdruck wird Klasse abgeleitet oder Interface implementiert und dann ein Objekt erzeugt.

Beispiel: Anonyme Klassen

```
class ManyReaders {
  private List<String> queue;

  public void startThread(BufferedReader reader) {
    final BufferedReader someReader = reader;
    Thread rThread = new Thread() {
      public void run() {
        String line = someReader.readLine();
        queue.add(line);
      }
    }.start();
  }

  public void foo() {
    queue = new ArrayList<String>();
    startThread(someReader1);
    startThread(someReader2);
    ...
  }
}
```

Anonyme Klassen: Syntax

- Hinter new-Ausdruck `new C(...)` kann in geschweiften Klammern Körper einer Klassendeklaration kommen.
- Damit wird eine Klasse von C abgeleitet (oder Interface C implementiert), als ob sie diesen Körper hätte.
- Dann wird Objekt dieser Klasse erzeugt und zurückgegeben.
- Anonyme Klassen können nur Default-Konstruktoren haben.

Anonyme und innere Klassen: Wann?

- Die meisten Klassen, die man verwendet, sind weder anonym noch innere.
- Innere Klassen: Klassen, die nur im Kontext einer anderen Klassen Sinn machen. Dann typischerweise private deklarieren.
- Anonyme Klassen: Nur ein einziger new-Ausdruck für die Klasse im ganzen Programm; wenig Code (max. 10-20 Zeilen).
- Anonyme Klassen können leicht unübersichtlich werden.

Zusammenfassung

- Nebenläufigkeit (concurrency): Mehrere Threads laufen konzeptuell gleichzeitig ab.
- Synchronisation: Sicherstellen, dass nur ein Thread gleichzeitig ein Stück Code ausführen kann.
- Kommunikation zwischen Threads: Wait/Notify, Pipes
- Verschachtelte, anonyme Klassen.