Ludwig Maximilians Universität München

Java-Threads

Nebenläufige Ausführung

von Prozessen

in einer Java - VM

Ludwig Maximilians Universität München

Was ist ein Thread?

- nebenläufiger Ausführungsstrang (*thread* (engl.) = Faden)
- "leichtgewichtiger" Prozeß (kein Kontext)
- in herkömmlichen Sprachen: Nebenläufigkeit durch Starten weiterer Prozesse mit Systemraufrufen (`fork`, `spawn`)
- in Java: Threads eingebaut (built-in)

Ludwig Maximilians Universität München

Warum Threads?

- Auslagern zeitintensiver Jobs (Laden von Bildern, etc.)
- Animationen
- Architektur
  - Verteiltes objekt-orientiertes Modell
  - Objekte können nebenläufig Methoden ausführen

Ludwig Maximilians Universität München

Threads in Java

- Klasse `java.lang.Thread` im Java-API
- `Thread` implementiert das Interface `java.lang.Runnable`
- Jeder Thread muß eine Methode `run()` besitzen, welche die Aufgabe des Threads definiert.

Ludwig Maximilians Universität München

Was kann man mit Threads anstellen?

- Erzeugen
- Starten
- Pausieren
- Synchronisieren
- (Name, Gruppe, Priorität bestimmen & veränden)

Ludwig Maximilians Universität München

Zwei Arten, Threads zu erzeugen

1. "*Sub-classing*" (beschränkt durch "single inheritance")
   - Bilden einer Unterklasse C von `Thread`
   - Überschreiben der Methode `run()`
   - `new` C`()`
2. "*Delegation*" (universell)
   - Klasse C als `implements Runnable` deklarieren
   - Methode `run()` darin definieren
   - `new Thread(`o`)` mit o Objekt der Klasse C

Ludwig Maximilians Universität München

Beispiel für "*Delegation*":

```
class Animation extends Applet implements Runnable {
  Image myImage;
  public Animation {
     myImage = getImage("myimage.gif"); }
  public void run() {
      ---hier passiert's---    }
  public void init() {
     Thread myThread = new Thread(this); }
}
```

Ludwig Maximilians Universität München

Beispiel für "*Sub-classing*":

```
class Animation extends Thread {
  Image myImage;
  Applet a;
  public Animation {
     myImage = a.getImage("myimage.gif"); }
  public void run() {
     ---hier passiert's--- }
  public static void main(String[] argv) {
     Thread myThread = new Animation(); }
}
```

Ludwig Maximilians Universität München

LMU

Starten, Unterbrechen & Pausieren (Unabhängige Prozesse)

- Starten: `start()`
- Unterbrechen: `interrupt()`
- Pausieren: `sleep(long` *`millisecs`*`)`
- Kontrollflußabgabe: `yield()`

Ludwig Maximilians Universität München

Synchronisation

- Problem bei nebenläufigen Prozessen:
  - Zugriff auf *gemeinsame* Variable
  - gleichzeitiges Schreiben und Lesen führt evtl. zu inkonsistenten Zuständen
- deshalb *Synchronisation* durch Objekte:
  - `synchronized (`*o*`) { ... }`
  - `synchronized ...` *m*`(...) { ... }`

Ludwig Maximilians Universität München

Semantik von `synchronized (`*o*`) { ... }`

- *o* bezeichnet ein Objekt
- Nur *ein* Thread kann einen *Lock* auf dieses Objekt anlegen.
- Alle, die ein `synchronized (`*o*`) { ... }` auswerten wollen, müssen warten, bis der Block (*kritischer Bereich*) vom ersten Thread wieder verlassen und der *Lock* damit freigegeben wurde.
- Bei synchronisierten Methoden `synchronized ...` *m*`(...)` wird der *Lock* beim Methodenaufruf *o*.*m* für das Objekt *o* angelegt.

Ludwig Maximilians Universität München

`wait` und `notify`

- `wait()` und `notify()` nur in synchronisierten Abschnitten
- *o*.`wait()` legt den ausführenden Thread schlafen und löst alle seine *Locks* auf *o*
- *o*.`notify()` weckt irgendeinen Thread auf, der bis dahin wegen eines `wait()` auf *o* blockiert war. Dieser kehrt wieder in den Wettbewerb um den entsprechenden Lock zurück.
- *o*.`notifyAll()` wie *o*.`notify()`, aber *alle* auf *o* wartenden Threads werden aufgeweckt
- `notify()` garantiert keine Fairneß beim Auswählen des geweckten Threads. Man sollte keine Annahmen über die Auswahl machen.

Ludwig Maximilians Universität München

Klasse `ThreadGroup`

- Threads können "gruppenweise" angesprochen werden
- Konstruktoren: `ThreadGroup(String name)` und `ThreadGroup(ThreadGroup group, String name)`
- Anzahl der Threads bzw. Gruppen: `int activeCount()` und `int activeGroupCount()`
- In der Hierarchie nach oben: `ThreadGroup getParent()`