Nebenläufige Programmierung

Threads, Callbacks, Synchronisation, Deadlocks

Nebenläufigkeit

Aktionen sind nebenläufig, bzw. parallelisierbar, wenn keine Aktion das Resultat einer anderen Aktion benötigt.

(Definition nach Wikipedia)

Nebenläufigkeit

Problem: Prozessor kann lediglich eine Operation gleichzeitig auführen

Lösung: durch organisiertes Aufteilen der Prozesse in sogenannte Threads können Prozesse ähnlich dem Pipelineprinzip abwechselnd bearbeitet werden

Nebenläufigkeit

Nutzen:

- Scheinbar gleichzeitige Ausführung von verschiedenen Operationen (ob Nutzung verschiedener Programme oder Ausführung mehrerer Prozesse innerhalb eines Programmes)
- Geringere Laufzeit

Threads

- Ein Prozess enthält immer mindestens einen Thread

  → der „rote Faden“ im Programmfluss

  Threads werden
- bei den meisten Prozessoren heutzutage eingesetzt (sog. native Threads)
- von einem Scheduler verwaltet (Zuteilung der Ressourcen)

Threads in Java

In der JVM werden Threads auch bei Betriebssystemen ohne eigene Nebenläufigkeit simuliert

Allgemeines Problem: bei möglichen Ressourcenkonflikten muss der Programmierer vorsorgen!

Threads in Java
Implementierung

1. Methode: Runnable-Interface

- Thread-Objekte implementieren *Runnable*
- 1 Initialisierungs-Objekt

```
TollesObjekt Objekt1 = new Thread();    // erstellt neues Thread-Objekt
TollesObjekt Objekt2 = new Thread();
Objekt1.start();                        //startet das Thread-Objekt
Objekt2.start();
```

Threads in Java
Implementierung

2. Methode — Thread-Klasse erweitern

Vorteile:

- super.start() direkt im Konstruktor
- sehr viele verschiedene Funktionen, z.b. Setting der Priorität, des Namens etc. (static Thread currentThread())

→ nützlich bei Prioritätsabfrage

Threads in Java
Implementierung

2. Methode

Nachteile:

- Andere Klassenerweiterungen sind nicht mehr möglich

→ Entscheidung je nach Situation

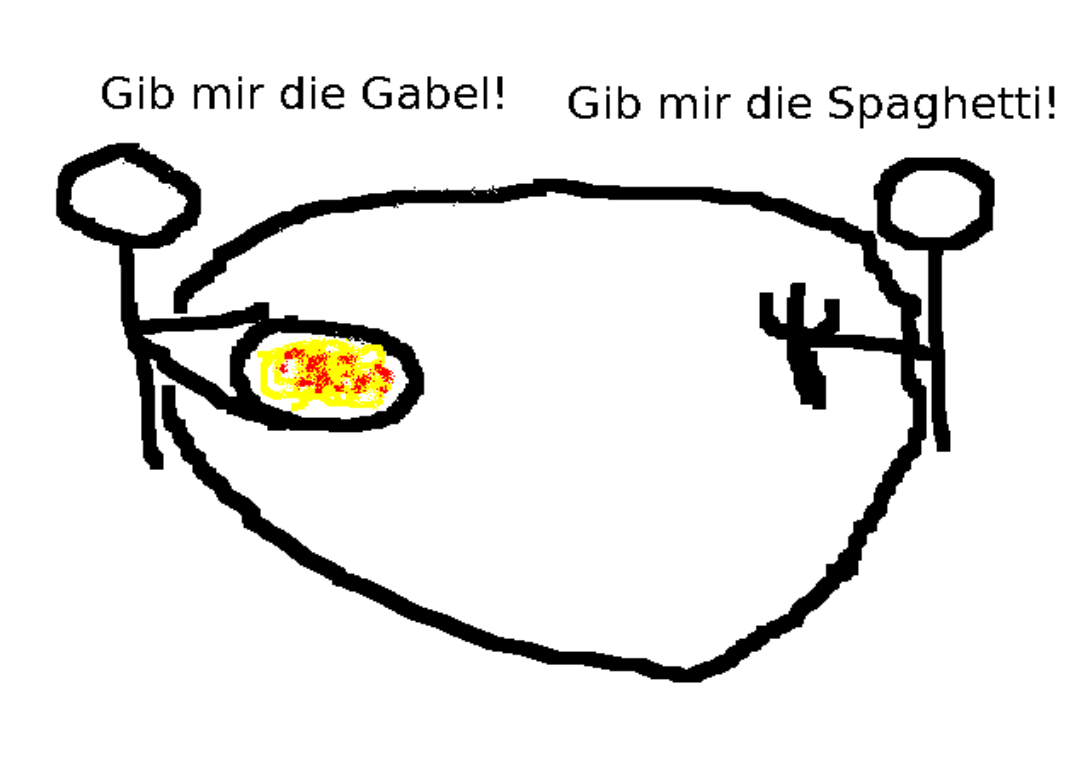
Threads in Java Implementierung

Thread.currentThread().getPriority();
→ liefert Priorität des gewählten Threads
→ wichtig bei Lösung/Vorbeugung von Ressourcenkonflikten

Thread.sleep(millisec);
→ versetzt den ausführenden Thread in „Ruhemodus“
Wichtig: statische Methode

Deadlocks

Situation:



Deadlocks

Erkennen von Deadlocks:

- Übersichtliche Programmstruktur
- Schnittstellen/Zugriffe kennzeichnen

Tipp in Java:

JVM besitzt eingebaute Deadlock-Erkennung

→ einfach in Konsole

Windows: Strg + Break

Linux: Strg + /

Synchronisation

Problem: Aufgrund statischer Methoden, welche von mehreren Objekten genutzt werden, besteht die Gefahr eines Ressourcenkonflikts!

- werden auch als „kritische Abschnitte“, bzw. „nicht thread-sicher“ bezeichnet

Lösung: Nutzung diverser Schließ- und Prüfmethoden zur Synchronisation

Synchronisation

Zwei Verfahren zum Sichern kritischer Abschnitte:

1. synchronized-Abschnitte
- sehr einfache Implementierung
- jedoch auch wenig Möglichkeiten

2. Erstellung eines Lock-Objektes
- etwas komplizierter als „synchronized“, jedoch immer noch einfach
- viele verschiedene Methoden zu unterschiedlichster Anwendung

Synchronisation
Lock-Objekte

- Lock-Objekte werden in der genutzten Klasse angelegt

Interessante Methoden:

- void lock()
- boolean tryLock()
- void unlock()

Callbacks

- Aufruf eines Objekts über die Initialisierungsklasse

```
MainClass.Objekt1.interessanteMethode();
```

- „Interaktiver Code“, zum Beispiel in Keylistenern verwendet

  → kann sehr komplex werden, mit Vorsicht genießen

Quellen

- Wikipedia „Nebenläufigkeit“
- Galileo Open Book „Java ist auch eine Insel“
- „Einführung in die Programmierung mit Java“ - Robert Sedgewick, Kevin Wayne
- „Parallele Programmierung spielend gelernt mit dem Java-Hamster-Modell“ - Dietrich Boles