Objektorientierte Programmierung

Ausnahmebehandlung und Nebenläufigkeit

9. Vorlesung am 15. Dezember 2010

Ausnahmebehandlung in Java

```
class A {
    void foo() throws Help, SyntaxError { ... }
}
class B extends A {
    void foo() throws Help {
        if (helpNeeded())
            throw new Help();
    }
}
...
try { ... }
catch (Help e) { ... }
catch (Exception e) { ... }
finally { ... }
```

Liste erlaubter Ausnahmen

eingeschränkte Liste erlaubter Ausnahmen

Werfen einer Ausnahme

bis zum Auftreten einer Ausnahme ausgeführt

try-catch-finally-Block innerhalb einer Methode

Handler der Reihe nach (von oben nach unten) probiert bis Typ passt

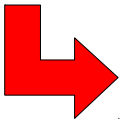
am Ende jedenfalls ausgeführt egal ob Ausnahme aufgetreten

Ausnahmebehandlung/Ersetzbarkeit

- Methode in Unterklasse soll nur dann Exception werfen, wenn Aufrufer der Methode in Oberklasse das erwartet

  entsprechend den Zusicherungen

- Einschränkungen durch **`throws`**-Klausel nicht hinreichend, da dynamisches Verhalten wichtig

   Zusicherungen beachten 

- Information über Exceptions = Nachbedingung

Einsatz von Ausnamebehandlungen

- Ursachen von Programmabbrüchen finden
  - ↳ unvorhergesehene Abbrüche nicht vermeidbar
- Kontrolliertes Wiederaufsetzen nach Fehlern
  - ↳ in Praxis notwendig, sinnvolles Aufsetzen schwierig
- Vorzeitiger Ausstieg aus Programmkonstrukten
  - ↳ fehleranfällig (speziell wenn nicht lokal), vermeidbar
- Rückgabe alternativer Ergebniswerte
  - ↳ Hinweis auf schlechte Struktur, vermeidbar

Beispiel für Ausnahmen 1

- Ohne Ausnahmebehandlung:

```
while (x != null)
    x = x.getNext();
```

  - doppelte Überprüfung ob ungleich **`null`**

- Mit Ausnahmebehandlung:

```
try { while (true)
          x = x.getNext(); }
catch (NullPointerException e) {}
```

  - fehleranfällig, da Exception in **`getNext`** auslösbar

Beispiel für Ausnahmen 2

- Trickreiche Verwendung von Ausnahmen:

```
if(x instanceof T1){…}
else if(x instanceof T2)
     {…}
…
else if(x instanceof Tn)
     {…}
else {…}
```

```
try {throw(x);}
catch(T1 x){…}
catch(T2 x){…}
…
catch(Tn x){…}
catch(Exception x)
     {…}
```

  - Ausnahmen lokal, daher wenig fehleranfällig
  - aber beide Varianten schwer wartbar

Beispiel für Ausnahmen 3

- Ohne Ausnahmebehandlung:

```
String addA (String x, String y) {
    if (isNum(x) && isNum(y)) {…}
    else { return "Error"; }
}
```

- Mit Ausnahmebehandlung:

```
String addB (String x, String y)
        throws NoNumString {
    if (isNum(x) && isNum(y)) {…}
    else {throw new NoNumString();}
}
```

Nebenläufige Programmierung

- *Mehrere* gleichzeitig laufende *Threads*
- Gleichzeitige bzw. *überlappte Variablenzugriffe*
  - ↳ Fehler wenn Zustände inkonsistent (häufig!)
- *Synchronisation* vermeidet Variablenzugriffe während inkonsistenter Zustände
- *Programmierer* muss sich um Synchronisation kümmern, System macht das nicht automatisch
- Synchronisation ist häufige *Fehlerquelle*

Beispiel: fehlende Synchronisation

```
public class Zaehler {
    private int i = 0, j = 0;
    public void schnipp() { i++; j++; }
}
```

- Mehrere Threads rufen **`schnipp`** auf
  - ↳ möglicherweise `i` ≠ `j` durch Überlappung
- Aufrufe „vergessen“ (auch bei einer Variablen)
- Problem durch Testen schwer feststellbar

Einfache Synchronisation in Java

```
public class Zaehler {
    private int i = 0, j = 0;
    public synchronized void schnipp()
        { i++; j++; }
}
```

- Maximal *eine* Ausführung einer **`synchronized`** Methode eines Objekts zu einem Zeitpunkt — „*mutual exclusion*“
- Weitere fast gleichzeitige Aufrufe von **`schnipp`** blockiert

  ↳ andere Threads warten bis ausgeführte Methode fertig
- **`synchronized`** Methoden sollen nur kurz laufen

Synchronisierte Blöcke in Java

```
public class Zaehler {
   private int i = 0, j = 0;
   public void schnipp() {
      synchronized(this) { i++; }
      synchronized(this) { j++; }
} }
```

- Kurzfristig **`i`** ≠ **`j`** möglich, aber kein „vergessener“ Aufruf
- *Lock* für Thread auf **`this`** (= Argument von **`synchronized`**)
  - **`synchronized`** Methoden setzen Lock automatisch immer auf **`this`**
  - Zugriff auf **`synchronized`** Blöcke/Methoden nur für diesen Thread

Threads warten auf Objektzustände

```
public class Druckertreiber {
    private boolean online = false;
    public synchronized void drucke (String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // schicke s zum Drucker
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
}
```

nur in synchronized Methode oder Block

negierte Eingangs-bedingung wiederholt überprüft

Thread kommt in Warteschlange des Objekts bis ein anderer Thread ihn wieder aufweckt

Exception muss abgefangen werden. Exception ausgelöst bei Abbruch des wartenden Threads

weckt alle Threads in Warteschlange des aktuellen Objekts (= **this**) auf

Erzeugen neuer Threads

```
public class Produzent implements Runnable {
    private Druckertreiber t;
    public Produzent(Druckertreiber _t) { t = _t; }
    public void run() {
        String s = ...
        for (;;) {
            ...              // produziere neuen Wert in s
            t.drucke(s); // schicke s an Druckertreiber
}   }   }
........
        Druckertreiber t = new Druckertreiber(...);
        for (int i = 0; i < 10; i++) {
            Produzent p = new Produzent(t);
            new Thread(p).start();
```

Interface für Thread-Erzeugung

Methode die im Thread ausgeführt wird

fast immer eine Endlosschleife

neuen Thread erzeugen

Synchronisation und Bibliotheken

- Client oder Server sorgt für Synchronisation
  - kein einheitliches Konzept, von Fall zu Fall anders
  - Dokumentation (Zusicherungen) besonders wichtig
- Beispiele:
  - **`Vector`** sorgt selbst dafür, nicht beeinflussbar
  - Client muss für Synchronisation in **`List`** sorgen oder Listen z.B. folgendermaßen erzeugen:

```
List x = Collections.synchronizedList(
                              new LinkedList(…))
```

Probleme bei Synchronisation

- Synchronisation kann Threads blockieren und dabei die Ausführung (sehr stark) verzögern
  - „deadlock“ und „livelock“ als Extremfälle (unendliche Verzögerung, „liveness properties“)
- solche Probleme nur durch Testen auffindbar
- nebenläufige Programmierung schwierig und fehleranfällig, Ersetzbarkeit schwer erreichbar

  ↳ auf Einfachheit der Synchronisation achten

  ↳ **`wait`**, **`notify`** und **`notifyAll`** möglichst vermeiden
- Erfahrung wichtig, spezielle Entwurfsmuster