

# Tagesprogramm

Ausnahmebehandlung

Nebenläufige Programmierung

## 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 { ... }
```

## Ausnahmebehandlung und Ersetzbarkeit

Methode in Unterklasse soll nur dann eine Exception werfen,  
wenn Aufrufer der Methode in Oberklasse dies erwartet

Einschränkungen durch `throws`-Klausel nicht hinreichend

→ Zusicherungen beachten (**Nachbedingung**)

## Einsatz von Ausnahmebehandlungen

Ursachen unvorhergesehener Programmabbrüche finden  
(kaum vermeidbar)

kontrolliertes Wiederaufsetzen nach Fehlern  
(notwendig, aber schwierig)

vorzeitiger Ausstieg aus Sprachkonstrukten  
(fehleranfällig, vermeidbar)

Rückgabe alternativer Ergebniswerte  
(schlechte Programmstruktur, vermeidbar)

## Ganz schlechtes Einsatzbeispiel

Ohne Ausnahmebehandlung:

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

Mit Ausnahmebehandlung:

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

nicht-lokal und fehleranfällig, Ausnahme auch in getNext auslösbar

## Schlechtes Einsatzbeispiel

trickreiche Verwendung von Ausnahmen:

<pre>if (x instanceof T1) {...} else if (x instanceof T2 {...} ... else if (x instanceof Tn {...} else {...}</pre>	<pre>try { throw x } catch (T1 x) {...} catch (T2 x) {...} ... catch (Tn x) {...} catch (Exception x) {...}</pre>
--	---

nur lokale Ausnahmen (daher weniger fehleranfällig)

aber beide Varianten schwer wartbar

## Empfohlenes Einsatzbeispiel

Ohne Ausnahmebehandlung:

```
public static String addA (String x, String y) {  
    if (onlyDigits(x) && onlyDigits(y)) { ... }  
    else return "Error";  
}
```

Mit Ausnahmebehandlung:

```
public static String addB (String x, String y)  
    throws NoNumberString {  
    if (onlyDigits(x) && onlyDigits(y)) { ... }  
    else throw new NoNumberString();  
}
```

sinnvoller Einsatz, da Fehlerabfragen vermieden werden

## Aufgabe: Welche Art von Ausnahmen?

Ausnahmen der Typen `RuntimeException` und `Error` brauchen in `throws`-Klauseln nicht angegeben zu werden.

Der Umgang mit allen anderen Ausnahmen wird überprüft.

Soll man bevorzugt **überprüfte Ausnahmen** verwenden?

- A: Ja, weil Überprüfungen sicherstellen, dass Ausnahmen abgefunden werden.
- B: Ja, weil sie nicht mit Ausnahmen vom System verwechselbar sind.
- C: Nein, weil sie für die wichtigsten Einsatzbereiche nicht anwendbar sind.
- D: Nein, weil sie die Wiederverwendung behindern können.



# Threads und Synchronisation

mehrere gleichzeitig ablaufende (= nebenläufige) Threads

gleichzeitige und überlappte Zugriffe auf Variablen

→ Fehler wenn inkonsistente Zustände existieren (häufig!)

**Synchronisation** soll gleichzeitige und überlappte Zugriffe vermeiden während Zustände inkonsistent sein können

Programmierer(in) muss sich um Synchronisation kümmern

Synchronisation ist häufige Fehlerquelle

## Beispiel für fehlende Synchronisation

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

überlappte Aufrufe von `schnipp` in mehreren Threads

→ möglicherweise  $i \neq j$

auch bei nur einer Variablen werden Aufrufe „vergessen“

Problem bei einfachem Testen kaum feststellbar

## Einfache Synchronisation in Java

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

während der Ausführung einer `synchronized` Methode kann kein anderer Thread eine `synchronized` Methode desselben Objekts ausführen:

### Mutual-Exclusion

weitere (fast) gleichzeitige Aufrufe von `schnipp` blockiert  
→ andere Threads warten bis ausgeführte Methode fertig

`synchronized` Methoden sollen nur sehr 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  $i \neq j$  möglich, aber kein Aufruf „vergessen“

**Lock** für Thread auf Argument von `synchronized` (= `this`)

nur dieser Thread darf auf `synchronized` Blöcke und Methoden desselben Objekts zugreifen, andere müssen warten

`synchronized` Methoden setzen Lock immer auf `this`

## 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();
    }
}
```

## Erzeugen neuer Threads

```
public class Produzent implements Runnable {
    private Druckertreiber t;
    public Produzent(Druckertreiber t) { this.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();
}
```

## Synchronisation und Bibliotheksklassen

manchmal Client für Synchronisation verantwortlich, manchmal Server

→ Dokumentation (Zusicherungen) sehr wichtig

z.B. Synchronisation in `Vector` nicht beeinflussbar

z.B. Client muss für Synchronisation in `List` sorgen:

`List` von `synchronized Methoden/Blöcken` aus verwenden  
oder `Listen` folgendermaßen erzeugen:

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

## Probleme bei Synchronisation

Synchronisation kann Threads blockieren und dabei die Ausführung von Programmen (sehr stark) verzögern

**Deadlock** und **Livelock** als Extremfälle  
(= unendliche Verzögerung, **Liveness-Properties**)

meist nur durch Testen auffindbar

nebenläufige Programmierung schwierig und fehleranfällig  
(vor allem Ersetzbarkeit schwer zu erreichen)

- auf Einfachheit der Synchronisation achten
- `wait`, `notify` und `notifyAll` möglichst vermeiden

Erfahrung wichtig (z.B. spezielle Entwurfsmuster)



## Erreichbarer Parallelisierungsgrad

Synchronisation **verringert** erreichbaren Grad an Parallelität

hoher Parallelisierungsgrad erreichbar, wenn Daten **nicht** voneinander abhängen

daher **gesamte Programmstruktur** so aufbauen, dass Daten möglichst wenig voneinander abhängen (lokal nicht viel erreichbar)

**Datenabstraktion** und **Datenunabhängigkeit** schwer vereinbar