
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 – Ersetzbarkeit

- Methode in Unterklasse soll nur dann eine Exception werfen, wenn Aufrufer der Methode in Oberklasse dies erwartet
- Einschränkungen auf `throws`-Klausel nicht hinreichend
⇒ Zusicherungen beachten
- Information über mögliche Exceptions = Nachbedingung

Einsatz von Ausnahmebehandlungen

- Ursachen unvorhergesehener Programmabbrüche finden
(kaum vermeidbar, außer durch Wiederaufsetzen)
- kontrolliertes Wiederaufsetzen nach Fehlern
(in der Praxis notwendig, sinnvolles Aufsetzen schwierig)
- vorzeitiger Ausstieg aus Sprachkonstrukten
(fehleranfällig, vor allem wenn nicht lokal; vermeidbar)
- Rückgabe alternativer Ergebniswerte
(oft Hinweis auf schlechte Programmstruktur; vermeidbar)

Einsatzbeispiel (ganz schlecht)

Ohne Ausnahmebehandlung:

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

Mit Ausnahmebehandlung:

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

fehleranfällig, da Ausnahme auch in getNext auslösbar

Einsatzbeispiel (eher schlecht)

trickreiche Verwendung von Ausnahmen:

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

keine nicht-lokalen Ausnahmen (daher weniger fehleranfällig)

aber beide Varianten schwer wartbar

Einsatzbeispiel (empfohlen)

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

Nebenläufige Programmierung

- 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 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 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) { 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

- kein einheitliches Schema – 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“)
- solche Probleme derzeit nur durch Testen auffindbar
- nebenläufige Programmierung schwierig und fehleranfällig
(vor allem Ersetzbarkeit sehr schwer zu erreichen)
⇒ auf Einfachheit der Synchronisation achten
⇒ `wait`, `notify` und `notifyAll` möglichst vermeiden
- Erfahrung wichtig (z.B. spezielle Entwurfsmuster)