

Exceptions

Fehler abfangen

zur Laufzeit auftretene Fehler sollen Programm nicht terminieren

- ungültige Eingabe kann wiederholt werden lassen
- Fehler in einem Teil muss andere Teile nicht berühren
- plötzliche Termination evtl. fatal
- ...

gebraucht: Mechanismus, um Fehler zur Laufzeit abzufangen und zu behandeln

muss damit natürlich Teil der Sprache werden

Was ist eigentlich *Fehler*? Allgemeiner: *Ausnahme* / *Exception*

Mechanismus

Ausnahmen werden

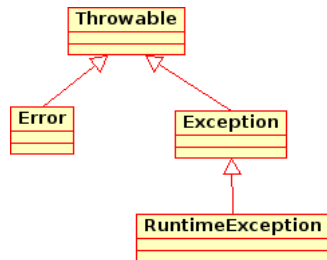
- **geworfen**: signalisiert das Auftreten, Ausführung des Programms wird unterbrochen
- **gefangen**: anderer Teil des Programms übernimmt Ausführung, normalerweise Fehlerbehandlung
- **deklariert**: Compiler soll Nicht-Abbruch garantieren können

verschiedene Ausnahmen erfordern i.A. verschiedene Behandlungen

Mechanismus für Verschiedenheit in Java vorhanden:
Klassenhierarchie mit Vererbung

Fehler- und Ausnahmenmodellierung

- **Throwable**: formalisiert allgemein Ausnahmen
- **Error**: Abnormalitäten, die normalerweise zu Programmabbruch führen
sollten nicht gefangen werden
- **Exception**: unerwartete / unerwünschte Situationen
sollten gefangen werden
- **RuntimeException**: Fehler in der Programmlogik
sollten erst gar nicht auftreten, also auch nicht gefangen werden



RuntimeExceptions

typische RuntimeExceptions: `IndexOutOfBoundsException`, `NullPointerException`, etc.

Auftreten kann eigentlich vorher ausgeschlossen werden

sollten nicht gefangen werden, denn was würde Fehlerbehandlung schon besser machen?

RuntimeExceptions sind *unchecked*: müssen im Gegensatz zu Unterklassen von `Exception` nicht deklariert werden

Idee dahinter: Laufzeitfehler sind unvorhergesehen, können praktisch in jeder Methode auftreten

Ausnahmen in Java

Ausnahmen werfen mit `throw e`, wobei `e` vom Typ `Throwable` (besser: `Exception`)

Ausnahmen fangen:

$$\text{try } c \text{ catch } (E_1 \ e_1) \ c_1 \dots \text{catch } (E_i \ e_i) \ c_i \dots \text{catch } (E_n \ e_n) \ c_n$$

Block `c` wird ausgeführt

beim Auftreten einer Ausnahme `e` wird kleinstes i gesucht, sodass `e` Typ E_i hat

Block c_i wird ausgeführt

Finally

Programmfluss kann durch Ausnahmen unterbrochen werden, *bevor* essentieller Code ausgeführt wurde; Bsp.: EOFException beim Lesen aus einer Datei

essentieller Code (z.B. Datei schließen) kann dennoch automatisch ausgeführt werden (auch ohne Exception!)

```
FileReader reader;
try {
    reader = new FileReader("meineLieblingsDatei.txt");
    while (true) {
        reader.read();
    }
} catch (FileNotFoundException e) {
    System.out.println("Gibt's nicht. Schade.");
} catch (IOException e) {
    System.out.println("Heute genug getan.");
} finally {
    reader.close();
}
```

Ausnahmen deklarieren

kann eine Exception, die nicht RuntimeException ist, in einer Methode auftreten, so muss dies deklariert werden

```
class NichtsFunktioniert extends Exception { ... }

class A {
    ...
    public void probierMal() throws NichtsFunktioniert {
        ...
        throw new NichtsFunktioniert();
        ...
    }
    ...
}
```

nicht nötig, falls Ausnahme innerhalb der Methode gefangen wird

Zu beachten

- Ausnahmen selbst definieren falls nötig
- try-Blöcke verlangsamen Code nicht, auftretende Ausnahmen aber schon
- Ausnahmen nicht zur Steuerung des Kontrollflusses (wie z.B. `break`) einsetzen
- Ausnahmen sind wirklich Teil der Sprache: können z.B. auch in `catch`-Teilen auftreten
- Ausnahmen so speziell wie möglich abfangen und wirklich behandeln, nicht

```
catch (Exception e) { e.printStackTrace(System.err); }
```
- Ausnahmen in JavaDoc-Dokumentation nicht vergessen