Exceptions (Laufzeitfehler)

Ausnahmesituationen meistern statt abstürzen

Softwareentwicklung

2BI 2018/19

Exceptions (Laufzeitfehler)

▸ Einführung

- Exceptions sind:
  1. „Ungeplante“ Ereignisse bei der *Ausführung* eines *syntaktisch fehlerfreien* Programms (Laufzeitfehler)
  2. Instanzen der Klasse **`Exception`** (oder einer ihrer Subklassen) mit Informationen zum Ereignis

Ursachen für Exceptions	
Logische Fehler („Denkfehler“), z.B. • Arrayindex wird zu groß • Methodenaufruf mit Referenzvariable, die **`null`** ist	*Äußere Ereignisse*, auf die man keinen Einfluss hat, z.B. • Falsche Benutzereingabe • Kein Platz mehr auf der Festplatte • Netzwerkfehler
vermeidbar	nicht vermeidbar

Nutzen von Exceptions

- Bisher: Exception → Programmabbruch mit Stacktrace
  - Beispiel: `Double.parseDouble("abc")` führt zu:

```
java.lang.NumberFormatException: For input string: "abc"
  at java.lang.Double.parseDouble(Double.java:510)
  at Rechner.alsDouble(Rechner.java:99)
  at Rechner.access$2(Rechner.java:97)
  at Rechner$1.actionPerformed(Rechner.java:32)
      ⋮
```

- Wenn man Exceptions entsprechend *behandelt*,
  - werden Programmabstürze wegen äußerer Ereignisse vermieden
  - braucht man nur für den „Normalfall“ zu programmieren
  - kümmert man sich um die Ausnahmefälle an passender (anderer) Stelle
  - erhält man ein übersichtlicheres Programm

Exceptions werfen

- Wie verursacht man eine Exception?
  1. Ein neues **`Exception`**-Objekt erzeugen
  2. Dieses Objekt „werfen“ → Schlüsselwort **`throw`**

```
throw new Exception(…);
```

- Exceptions werden häufig nicht selbst geworfen, sondern vom Java-API
- Was geschieht nach dem Werfen einer Exception?
  - Der Programmablauf wird sofort *unterbrochen*, selbst innerhalb einer Java-Anweisung.
  - In den aufrufenden Methoden wird der nächste passende **`catch`**-Block gesucht.
  - Die Programmausführung setzt in diesem **`catch`**-Block fort.
  - Wird kein passender **`catch`**-Block gefunden, wird das Programm (so wie bisher) abgebrochen.

Exceptions behandeln

- Neue Kontrollstruktur → **`try`**/**`catch`**/**`finally`**

```
try {     // try-Block
   // Anweisungen für den „Normalfall“ (könnten Exceptions auslösen)
        ⋮
} catch (Exception1 ex) {     // catch-Block
   // Exceptions der Klasse Exception1 und ihrer Subklassen behandeln
        ⋮
} catch (Exception2 ex) {     // catch-Block
   // Exceptions der Klasse Exception2 und ihrer Subklassen behandeln
        ⋮
} finally {     // finally-Block
   // Abschließende Anweisungen („Aufräumen“), werden immer ausgeführt
        ⋮
}
```

- Überall verwendbar, wo auch andere Kontrollstrukturen (**`if`**, **`for`**, …) erlaubt sind

Exception-Gotchas

- Variablen, die in einem Block definiert wurden, sind nur dort sichtbar
  - Selbe Variable in `try`- und `catch`-/`finally` Block benötigt → außerhalb definieren
- Um Exception-Arten individuell zu behandeln → mehrere `catch`-Blöcke
- Speziellere Exceptions (z.B. `IOException`) muss man *vor* allgemeineren Exceptions (z.B. `Exception`) behandeln
- *Entweder* die `catch`-Blöcke *oder* den `finally`-Block darf man weglassen
  - Aber bei „checked“ Exceptions → „catch-or-declare“-Regel

Exceptions *nicht* behandeln

- Um eine Exception-Art in einer Methode/Konstruktor nicht zu behandeln
  → an aufrufende Methode/Konstruktor *weitergeben*
- Dort Exception behandeln oder wieder weitergeben → „catch-or-declare“-Regel
- Um eine Art von Exceptions weiterzugeben → **`throws`** im Methodenkopf

```
// Diese Methode kann IOExceptions werfen
public String lesen(String dateiName) throws IOException {
    ⋮
}
```

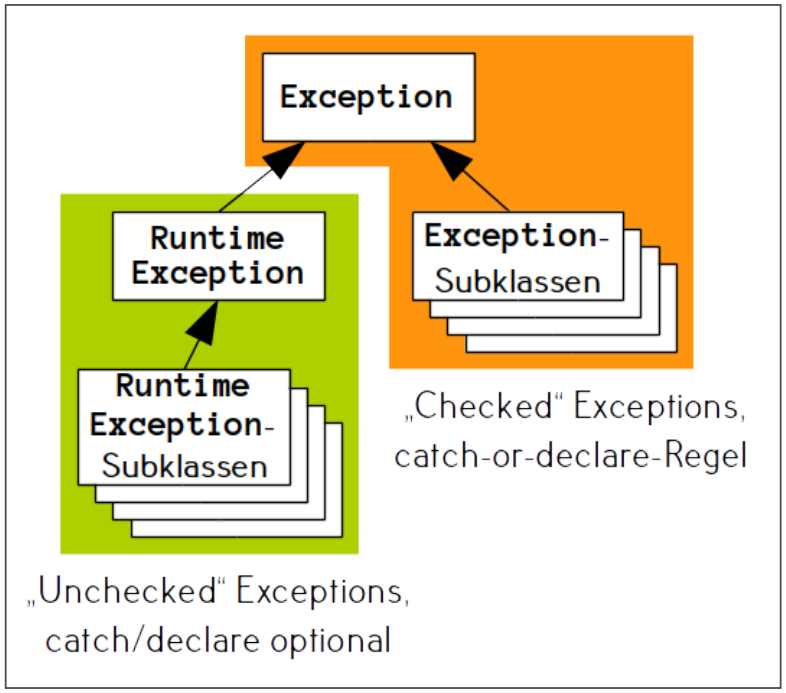
- Mehrere Arten von Exceptions weitergeben → nach **`throws`** aufzählen

```
// Diese Methode kann IOExceptions und TimeoutExceptions werfen
public String lesen(String dateiName) throws IOException, TimeoutException {
    ⋮
}
```

- Unterscheide zwischen **`throw`** und **`throws`**!

Exception-Hierarchie

- Beim *Programmieren* unterscheidet man zwischen:
  - „Checked“ Exceptions → *müssen* behandelt werden → „catch-or-declare“-Regel
  - „Unchecked“ Exceptions → *können* behandelt werden, müssen aber nicht



- Zur *Laufzeit* verhalten sich beide Arten von Exceptions gleich

`try`-with-resources

- Ressourcen → alles, was **`AutoCloseable`** ist, z.B. **`InputStream`**, **`Reader`**, **`Writer`**, …
  - **`AutoCloseable`** → besitzt Methode **`close()`**
- Vergleich mit konventionellem **`try`**/**`catch`**/**`finally`**
  - Benötigte Ressourcen werden mit **`try (…)`** erzeugt
  - Ressourcen werden automatisch geschlossen → oft kein **`finally`** nötig
  - Übersichtlichere und kürzere Schreibweise

```
try (
    InputStream istr = Files.newInputStream(…);
    OutputStream ostr = Files.newOutputStream(…);
) {
    // istr und ostr verwenden
        ⋮
} catch (IOException ex) {
        ⋮
}

// ostr.close() und istr.close() wurden automatisch aufgerufen
```