Ausnahmebehandlung

Laufzeitfehler

- weitere Ausführung unmöglich $\Rightarrow$ Ausnahme *geworfen*
- Beispiele: `ArrayIndexOutOfBoundsException`
  `NullPointerException`
  `ArithmeticException`
  `AssertionError`
  `OutOfMemoryError`
  `StackOverflowError`
- Ausnahmen sind Instanzen von `Throwable`
- eigene Ausnahme werfen: `throw new MyException();`
- Ausnahme nicht abgefangen $\Rightarrow$ Abbruch mit Stack Trace

Abfangen von Ausnahmen

```
public static void main (String[] args) {
    try {
        for (int i = 0; i < 3; i++)
            System.out.println(args[i]);
    }
    catch (ArrayIndexOutOfBoundsException ex) {
        System.out.println("ERROR: Argumente fehlen!");
    }
    catch (Exception ex) {
        System.out.println("ERROR (abgefangen):");
        ex.printStackTrace();
    }
    System.out.println("keine Ausnahme zu sehen!");
}
```

Weiterleiten eigener Ausnahmen

```
class MyExc extends Exception {
    public MyExc (String msg) { super(msg); }
}
public class ExceptionPropagationTest {
    public static void main (String[] args) {
        try { for (int i=0; i < args.length; i++)
                System.out.println(test(args[i])); }
        catch (MyExc ex)
            { System.out.println (ex.getMessage()); }
    }
    private int test (String s) throws MyExc {
        if (s.equals("end")) throw new MyExc("Fertig!");
        else return s.length();
    }
}
```

Arten von Ausnahmen

- vordefinierte Ausnahmen:
  - Untertypen von `RuntimeException`: abfangbar
  - Untertypen von `Error`: nicht abfangbar
- selbstdefinierte Ausnahmen:
  - meist Untertypen von `Exception`
  - müssen abgefangen oder weitergeleitet werden
  - Weiterleitung nur wenn in Methodenkopf spezifiziert

```
void foo() throws ExcA, ExcB {...}
```

Ausnahmen und Untertypen

- Untertypen werfen nicht mehr Ausnahmen als Obertypen:

```
class A { void foo() throws ExcA, ExcB {...} }
class B extends A { void foo() throws ExcB {...} }
```

- Programmierer muss darauf achten:

  Methode in Untertyp darf nur in solchen Fällen Ausnahmen werfen wo man das auch für Methode in Obertyp erwartet

  Falsch:

```
class A { int get(){ return 1; } }
class B extends A {
    int get(){ throw new RuntimeException("not OK"); }
}
```

Einsatz von Ausnahmen

- Notwendigkeit von `throws`-Klauseln umstritten
- oft unklar, wo Ausnahme geworfen wurde
- Ausnahmen nicht zum vorzeitigen Ausstieg verwenden!
- Ausnahmen nicht für alternative Ergebnistypen verwenden!
- Ausnahmen für echte Ausnahmefälle gerechtfertigt, sonst nicht
- Ausnahmebehandlung kann nicht alle Auswirkungen von Ausnahmen restlos beseitigen

Vorbeugen statt Heilen

```
import java.io.*;
public class Numbered {
    private static final String errmsg =
                          "Usage: java Numbered <in> <out>";
    public static void main(String[] args) {
        if (args.length != 2) {
            System.err.println(errmsg);
            return;
        }
        ...  // eigentlicher Programmcode
    }
}
```

```
try {  // in main in Numbered (continued) ...
    BufferedReader in = null; BufferedWriter out = null;
    try {
        String line;
        in = new BufferedReader(new FileReader(args[0]));
        out = new BufferedWriter(new FileWriter(args[1]));
        for (int i=1; (line=in.readLine()) != null; i++)
            out.write(String.format("%6d: %s\n", i, line));
    }
    finally {
        if (in != null) in.close();
        if (out != null) out.close();
    }
}
catch (IOException ex) {
    System.err.println("I/O Error: " + ex.getMessage());
}
```

Aufräumen

- `try-catch-finally` als syntaktische Einheit
- `finally` immer ausgeführt (auch in Ausnahmefällen)
- in `finally`-Blöcken wird aufgeräumt
- Aufräumen schwierig, weil unklar wie weit `try` ausgeführt
- nur Variablen aus äußeren Blöcken zugreifbar
- falsche Programmaufrufe Normalfall, keine Ausnahme

Umgang mit Dateien

- Dateien über verschiedene Arten von *Streams* zugreifbar
- allgemein: 1. öffnen, 2. zugreifen, 3. schließen
  dabei immer `IOException` möglich
- Arten von Streams:

Byte Streams:	rohe Daten
Character Streams:	Zeichen eines Textes

- Arten von Dateizugriffen:

ungepuffert:	übergibt Daten rasch an Betriebssystem
gepuffert:	sammelt Daten zuerst (effizienter)

- `System.in`, `System.out`, `System.err`:
  ungepufferte Byte Streams (`InputStream`, `PrintStream`)

Character Streams auf Dateien

- gepuffert: `FileReader`, `FileWriter`
- ungepuffert: `BufferedReader`, `BufferedWriter`
- Lesen mit `readLine` (ähnlich wie in Iterator)
- Schreiben mit `write`
- `String.format` mit variabler Argumentanzahl
  - Formatstring `"%6d: %s\n"` beschreibt Ergebnis
  - je ein weiteres Argument für jedes '%' in Formatstring
  - z.B.: `"%6d"` = 6-stellige ganze Zahl
  - z.B.: `"%s"` = beliebig langer String