Ausnahmen

Die sog. „Exceptions“ dienen zur Behandlung von Laufzeitfehlern, z.B.

- unzlässiger Feldindex
  → java.lang.ArrayIndexOutOfBoundException
- Dereferenzierung von `null`
  → java.lang.NullPointerException
- Datei existiert nicht
  → java.io.FileNotFoundException
- Datei-Ende erreicht
  → java.io.EOFException
- Speicher voll
  → java.lang.OutOfMemoryException

Abfangen von Ausnahmen

Programmiertechnik:

1. kritischen Programmteil in `try {...}` -Block einschließen!
2. ggf. auftretende Ausnahmen in anschließenden `catch(Exception e) {...}` -Klauseln behandeln!

   Wie bei `if(_)_else if (_)_` wird die textuell zuerst passende Klausel angesprungen.
3. die auf alle Fälle auszuführenden Anweisungen in einen anschließenden `finally {...}` -Block schreiben!

`try-catch-finally` kann in einen umfassenden Block eingeschlossen werden.

try-Anweisung (Bsp. 46)

```
class TryAnweisung
{
  public static void main(String args[]) {
    int i = 0;
    try {
      int k = 1 / i;  //Division durch 0
      System.out.println("Test");
    }
    catch (ArithmeticException e) {
      System.out.println(e);
    }
    System.out.println("Ende");
  }
}
```

finally-Block (Bsp. 47)

```
class FinallyBeispiel
{
  public static void main(String args[]) {
    for (int i = 1; i <= 10 ; i++) {
      char[] feld = null;
      char c;
      double d;
      try {
        d = java.lang.Math.random();
        System.out.print(i + ". Zufallszahl: ");
        if (d < 0.5)
          c = feld[0];  //Feldzugriff ueber "null"
        System.out.println(d);
      }
      catch (NullPointerException e) {
        System.out.println("kleiner als 0.5");
      }
      finally {
        System.out.println("Scheifenende");
      }
    }
  }
}
```

Ausnahmen bei Methodendefinitionen

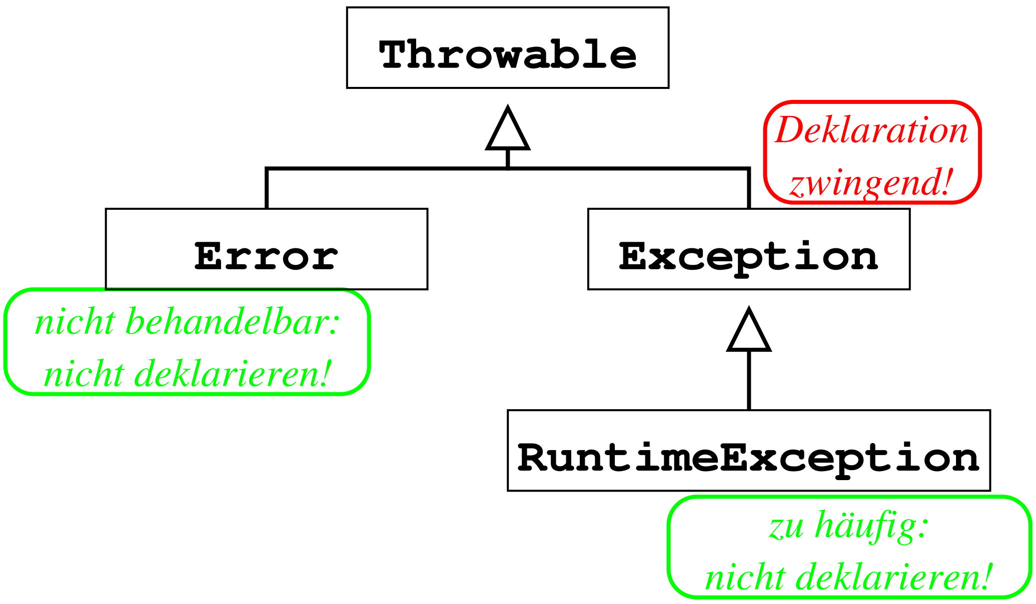
Auch selbstgeschriebene Methoden dürfen Ausnahmen auslösen, um Laufzeitfehler anzuzeigen:

`throw` *exceptionObject*;

Bei einer Methode muss eine nicht abgefangene Ausnahme deklariert werden, wenn

- diese eine Unterklassen von `Exception` ist
- und nicht Unterklasse von `RuntimeException` ist.

`throws` *ExceptionSubclass* [, *ExceptionSublass*]



Wird in einer Klasse eine Methode überschrieben, darf sie nur die Ausnahmen auslösen, die schon die (überschriebenen) Methode der Oberklasse deklariert hat.

throw-Anweisung (Bsp. 48)

```
class ThrowBeispiel
{
  public static void main(String args[]) {
    for (int i = 1; i <= 10 ; i++) {
      double d;
      try {
        d = java.lang.Math.random();
        System.out.print(i + ". Zufallszahl: ");
        if (d < 0.5)
          throw new NullPointerException();
        System.out.println(d);
      }
      catch (NullPointerException e) {
        System.out.println("kleiner als 0.5");
      }
      finally {
        System.out.println("Ende der Schleife");
      }
    }
  }
}
```

Ausnahme durch Aufruf (Bsp. 49)

```
class AusnahmeBeiAufruf
{
  static void erzeugeAusnahme() {
    throw new NullPointerException();
  }

  public static void main(String args[]) {
    try {
      erzeugeAusnahme();
      System.out.println("Alles in Ordnung.");
    }
    catch (Throwable t) {
      System.out.println(t);
    }
    System.out.println("Ende");
  }
}
```

throws-Klausel (Bsp. 50)

```
class AusnahmeBeiAufruf
{
  static void erzeugeAusnahme()
    throws java.io.IOException {
      //eine nicht-behandelte Ausnahme:
      throw new java.io.IOException();
  }

  public static void main(String args[]) {
    try {
      erzeugeAusnahme();
      System.out.println("Alles in Ordnung.");
    }
    catch (Throwable t) {
      System.out.println(t);
    }
    System.out.println("Ende");
  }
}
```

Benutzerdefinierte Ausnahmen

Der Name einer Ausnahme sollten den aufgetretenen Fehler ausreichend beschreiben, auch wenn er länglich wird, z.B.

`java.lang.ArrayIndexOutOfBoundException`

`MultipleElementInIntegerSetException`

Unbestimmte Schleifen

Ist die Anzahl der Schleifendurchläufe nicht im Voraus bestimmt, kann die Schleife ggf. auch durch Auslösen einer Ausnahme verlassen werden.

Schleifen und Ausnahmen (Bsp. 51)

```
class NeueAusnahme extends Exception {
}

class ThrowInSchleife
{
  public static void main(String args[]) {
    try {
      while (true) {
        System.out.println("Schleife...");
        throw new NeueAusnahme();
      }
    }
    catch (NeueAusnahme e) {
      System.out.println("Schleife abgebrochen");
    }
  }
}
```

Alles zusammen... (Bsp. 52)

```
class LeeresFeld extends Exception { }
class UngueltigeZiffer extends Exception {
  char zeichen;   //das unzulaessige Zeichen
  int position;   //der Feldindex des Zeichens
  UngueltigeZiffer(char z, int pos) {
    zeichen = z;
    position = pos;
  }
}
class FeldMitDezimalziffern {
  static long wert(char[] feld)
    throws LeeresFeld, UngueltigeZiffer {
      if (feld.length == 0) throw new LeeresFeld();
      int ergebnis = 0;
      for (int i = 0; i < feld.length; i++) {
        if (feld[i] < '0' || feld[i] > '9')
          throw new UngueltigeZiffer( feld[i], i );
        ergebnis = ergebnis * 10 + feld[i] - '0';
        if (ergebnis < 0) throw new ArithmeticException();
      }
      return ergebnis;
  }
  public static void main(String args[]) {
    try {
      char feld1 = {'8'}; wert(feld1);
      char feld2 = {'4','3','6','5','9','0'}; wert(feld2);
      char feld3 = {'2','7','?','1'}; wert(feld3);
    }
    catch (UngueltigeZiffer e) {
      System.out.println("Ungueltige Dezimalziffer: '"
        + e.zeichen + "' an Position " + e.position);
    }
    catch (Throwable t) { System.out.println(t); }
  }
}
```

Threads

= nebenläufige oder parallele Programme (Kontrollflüsse) auf gemeinsamem Speicherbereich

Motivation:

- Benutzeroberfläche ist nicht blockiert, wenn ein anderer Programmteil rechnet
- Ausnutzen von neuen Mehrprozessor-Rechnern
- ggf. übersichtlichere Programme (z. B. Verkehrssimulation)

Methoden aus der Klasse `java.lang.Thread`

- `run()` beinhaltet das „Hauptprogramm“ des Threads.
- `start()` erzeugt einen neuen (nebenläufigen) Kontrollfluss, der `run()` ausführt.
  Wurde der Thread mit `new Thread (obj)` erzeugt, wird statt dessen die Methode `run()` des `Runnable`-Objekts `obj` aufgerufen.
- `stop()` unterbricht `run()`
- `Thread.sleep()` legt den Thread „echt schlafen“ (kein busy waiting)

Nicht verwechseln mit den Methoden aus der Klasse `java.applet.Applet`:

- `init()` enthält notwendige einmalige Initialisierungsanweisungen (an Stelle einer `main()`-Methode)
- `start()` wird aufgerufen, wenn das Applet im Browser (bzw. auf dem Bildschirm) sichtbar wird.
- `stop()` wird aufgerufen, wenn das Applet verdeckt oder weggescrollt wird.

Thread-Erzeugung (Bsp. 53)

```
class NeuerThread extends Thread
{
  public void run() {
    System.out.println("Ein neuer Thread...");
  }
}

class ThreadBeispiel
{
  public static void main(String args[]) {
    Thread t = new NeuerThread() ;
    t.start() ;
  }
}
```

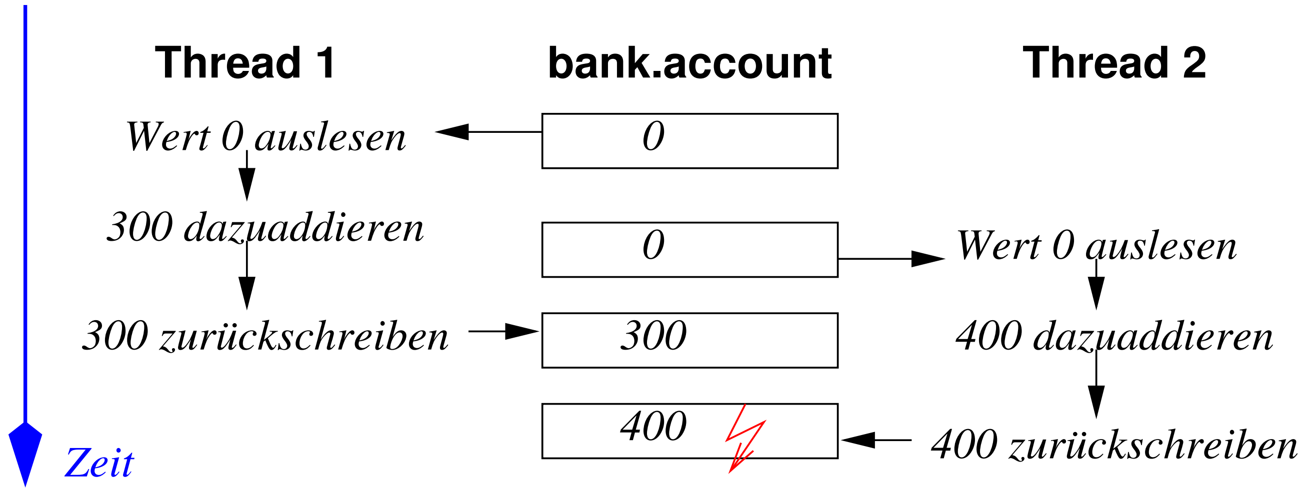
Mehrere Threads (Bsp. 54)

```
class PrintThread extends Thread
{
  private String text;
  PrintThread(String text) { this.text = text; }

  public void run() {
    while (true) {
      System.out.println(text);
      try {
        //ganzzahlige Zufallszahl zwischen 0 und 1000:
        int millisek = (int)(1000 * Math.random());
        sleep(millisek) ;
      }
      catch (Exception e) break;
    }
  }
  public static void main(String args[]) {
    new PrintThread("morgens").start();
    new PrintThread("mittags").start();
    new PrintThread("abends").start();
  }
}
```

Zugriffskonflikte

Da Threads auf gemeinsame Variablen zugreifen dürfen, kann es zu Konflikten beim gleichzeitigen Schreiben kommen.



Synchronisation zwischen Threads

Synchronisation von Code-Blöcken mit kritischen Bereichen:

```
synchronized (obj) {
    obj.x = obj.x + 300;
}
```

Synchronisation ganzer Methoden:

```
void erhoehe(int delta) {
  synchronized(this) {
    obj.x = obj.x + delta;
  }
}
```

... ist gleichbedeutend mit:

```
synchronized void erhoehe(int delta) {
  obj.x = obj.x + delta;
}
```

Synchronisation zwischen Threads (2)

Abhilfe: Kritische Bereiche durch Sperren schützen! Java stellt für jedes Objekt und jede Klasse eine (implizite) Sperre zur Verfügung.

`synchronized` (*obj*) {*kritischer Bereich*}