Raumeinteilung für Schein-/Bachelorklausur Teil I

185434 - 259425: AHV (Hörn)
259502 - 272413: AM
272420 - 273565: Fo I
273570 - 274323: Fo 2
274347 - 275180: Gr
275209 - 277120: Ro

Bitte rechtzeitig um 18:00 Uhr da sein.

Studentenausweis und Personalausweis mitbringen.
Taschen, Jacken und Handys werden vorne abgegeben.
Schreiben Sie mit blauem oder schwarzem Kugelschreiber.

Kein Toilettengang.
Wer schummelt, fällt sofort durch!

Schreiben Sie nur das, was in der Aufgabe verlangt wird.
Wenn Sie bei einer Aufgabe nicht weiterkommen, dann lösen Sie eine andere Aufgabe, sonst verlieren Sie Zeit.

Wer sich verspätet, kann nicht mitschreiben!
Es sind **keine** Hilfsmittel zugelassen.

Rückblick

Was ist der Stack-Speicher?

Was ist der Heap-Speicher?

Unterschied zwischen Instanz- und lokalen Variablen?
Was ist ein Konstruktor?

Wann wird der Konstruktor ausgeführt?

Konstruktoren der Superklasse aufrufen?

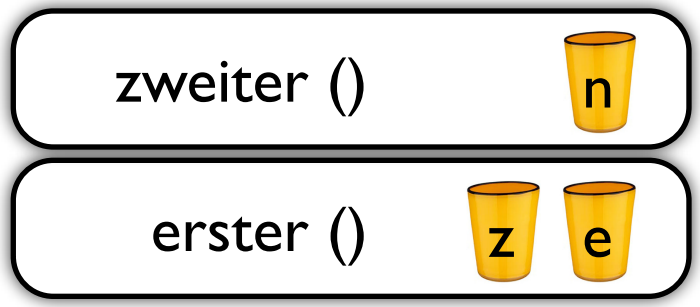
Konstruktoren innerhalb der Klasse aufrufen?

Geltungsbereich von lokalen Variablen und Referenzvariablen

```
class Test {
   // lokale Variablen z und e sind...
   public void erster () {
      int z = 13;              // ... nur in dieser
      int e = zweiter (z);  // Methode sichtbar
   }

   // n ist nur in dieser Methode sichtbar
   public int zweiter (int n) {  // lokaler Par.
      z = z * 2;     // z ist hier nicht sichtbar !!!
      return n * 2;
   }
}
```

erster() kann **n** nicht sehen !

n ist nur in der Methode zweiter() gültig, d.h., in ihrem Geltungsbereich.



z und **e** sind nur in der Methode erster() gültig.

zweiter() kann **z** und **e** nicht sehen !

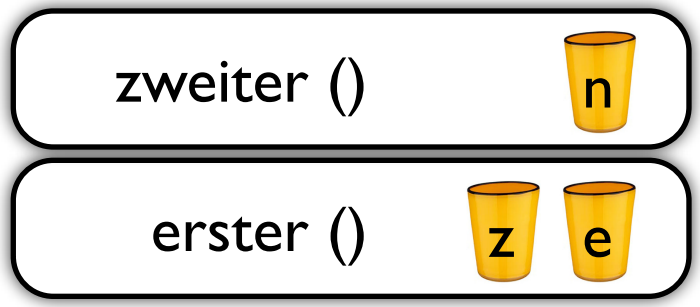
Stack-Speicher

Leben von lokalen Variablen und Referenzvariablen

```
class Test {
   // lokale Variablen z und e sind...
   public void erster () {
      int z = 13;              // ... nur in dieser
      int e = zweiter (z);  // Methode sichtbar
   }

   // n ist nur in dieser Methode sichtbar
   public int zweiter (int n) {  // lokaler Par.
      z = z * 2;     // z ist hier nicht sichtbar !!!
      return n * 2;
   }
}
```

erster() kann **n** nicht sehen !

n ist nur in der Methode zweiter() gültig. Sie lebt, solange zweiter() auf dem Stack ist.



z und **e** sind nur in der Methode erster() gültig. Sie leben, solange erster() auf dem Stack ist.

zweiter() kann **z** und **e** nicht sehen !

Stack-Speicher

Geltungsbereich von Instanzvariablen und Instanz-Referenzvariablen

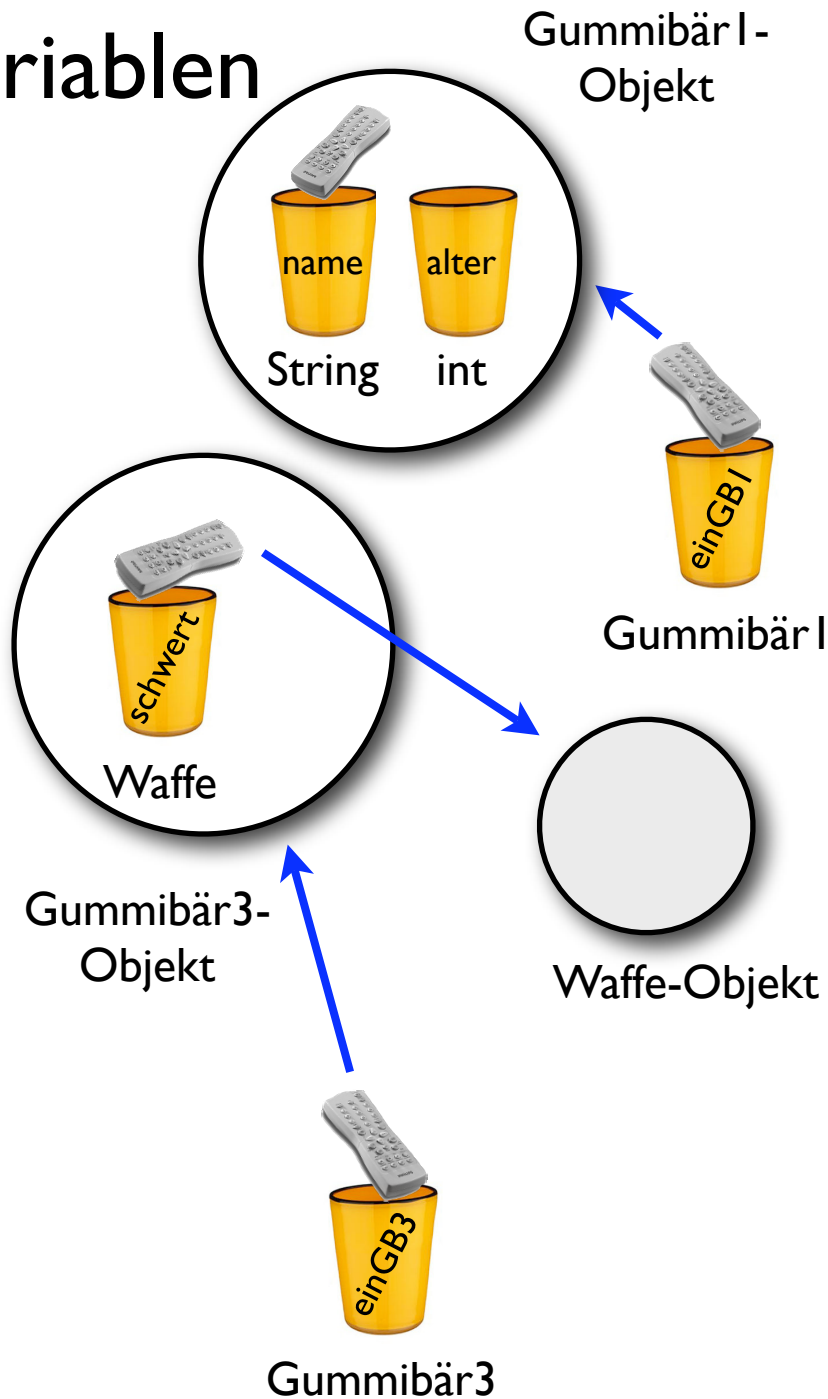
```
class Test {
   int ergebnis;  // Instanzvariablen und Instanz-Referenzvariablen sind
   Foo a;         // überall in der Klasse sichtbar

   public void erster () {
      int z = 13;    // lokale Variable z ist nur in dieser
      zweiter (z);  // Methode sichtbar
   }

   // zweiter () kann auf die Instanzvariable ergebnis zugreifen
   public void zweiter (int n) {  // lokaler Parameter
      ergebnis = n * 2;
   }
}
```

Leben von Instanz-Referenzvariablen

```
class Gummibär1 {
    String name;        // Instanzvariablen
    int alter;
}
```

```
class Gummibär3 {
    // Instanz-Referenzvariable und neues Objekt
    Waffe schwert = new Waffe ();
}
```

```
// irgendwo im Programm
Gummibär1 einGB1 = new Gummibär1 ();
Gummibär3 einGB3 = new Gummibär3 ();
```



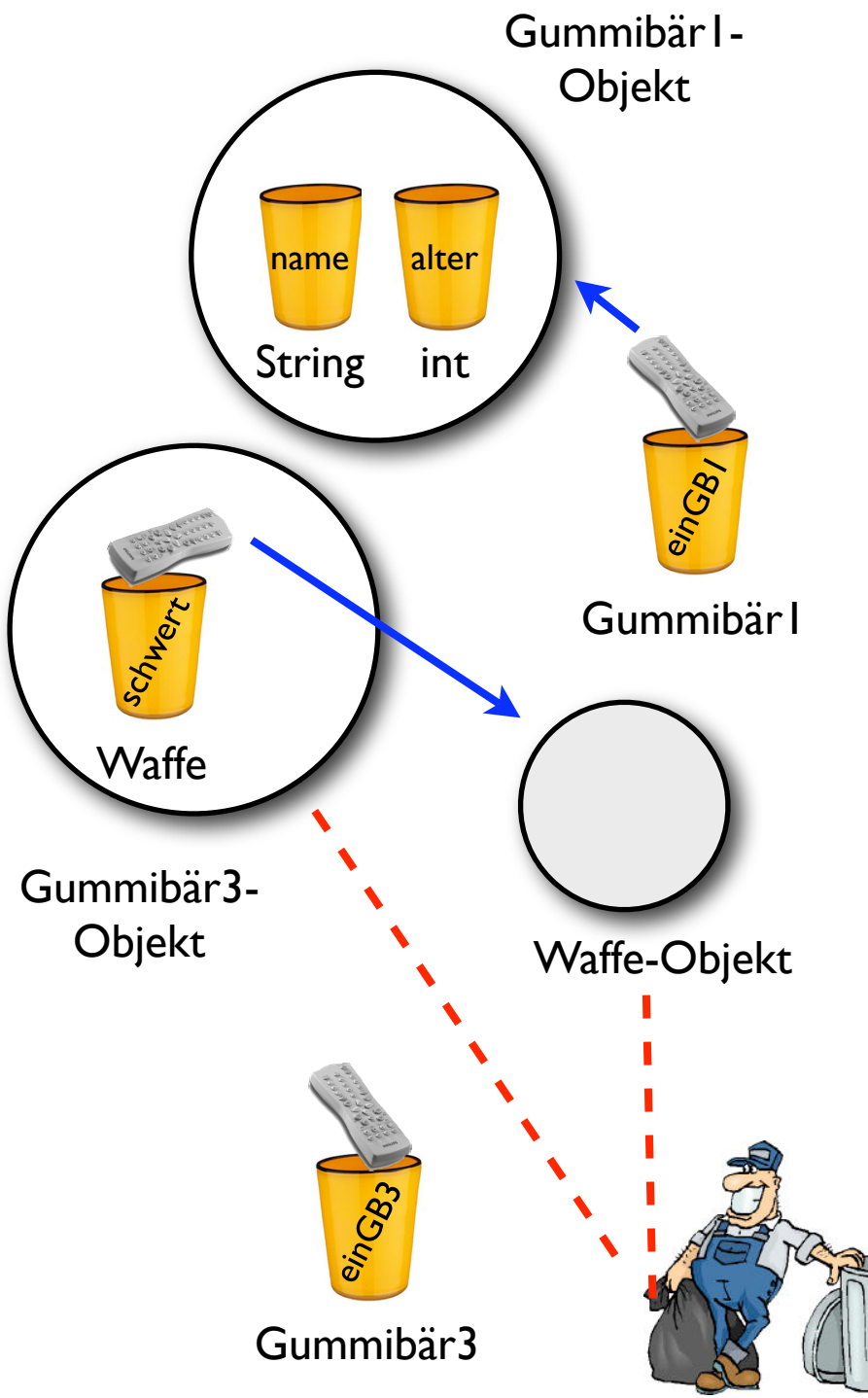
Instanzvariablen leben in Objekten.

Leben & Sterben von Instanz-Referenzvariablen

```
class Gummibär1 {
    String name;        // Instanzvariablen
    int alter;
}
```

```
class Gummibär3 {
    // Instanz-Referenzvariable und neues Objekt
    Waffe schwert = new Waffe ();
}
```

```
// irgendwo im Programm
einGB3 = null;
```



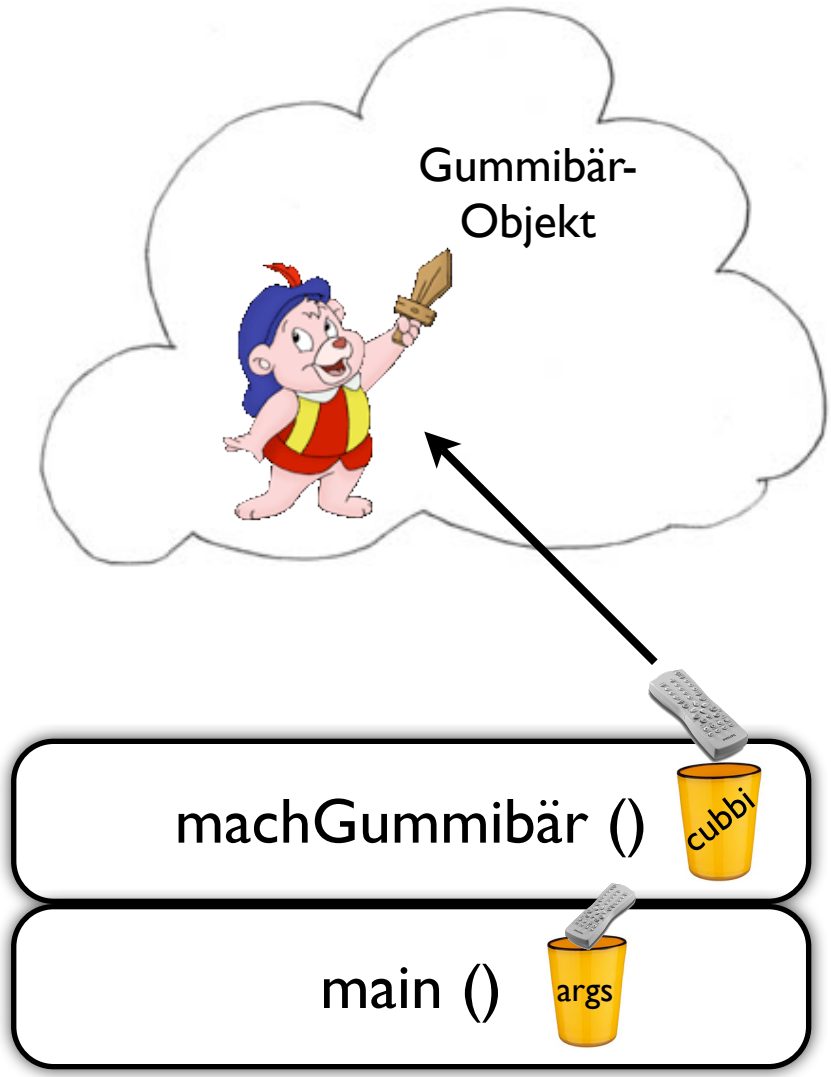
Instanzvariablen leben und sterben mit ihren Objekten.

Leben und Sterben von Objekten

(oder auch "cubbi lebt nur kurz")

Heap-Speicher

```
class GummibärTest {
   public static void main (String[] args) {
      machGummibär ();
   }

   public void machGummibär () {
      Gummibär cubbi = new Gummibär ();
   }
}
```



GummibärTest test = new GummibärTest ();

cubbi ist eine
lokale Referenzvariable!

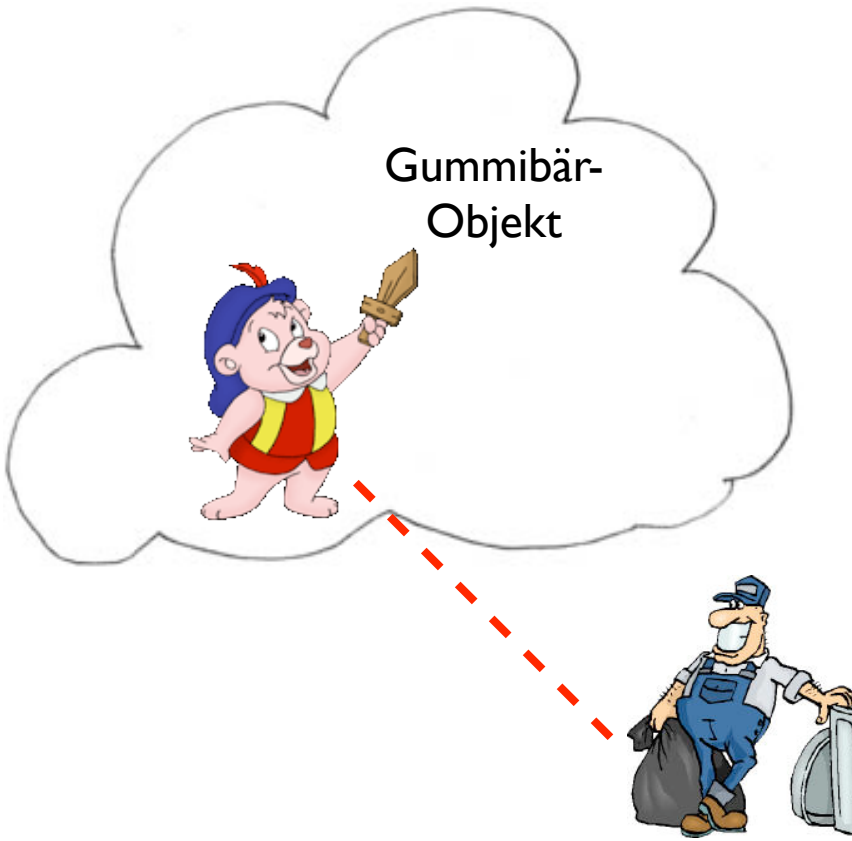
Stack-Speicher

Leben und Sterben von Objekten

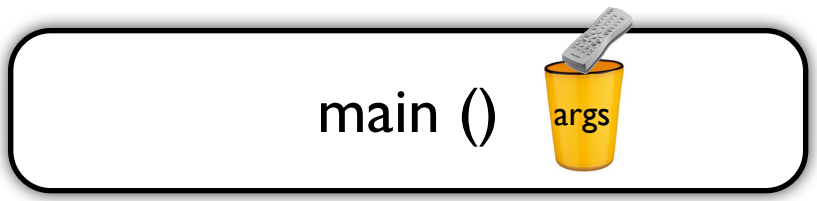
(oder auch "cubbi lebt nur kurz")

Heap-Speicher

```
class GummibärTest {
   public static void main (String[] args) {
      machGummibär ();
   }

   public void machGummibär () {
      Gummibär cubbi = new Gummibär ();
   }
}
```



cubbi "lebte" nur solange machGummibär() auf dem Stack war!



Stack-Speicher

Jetzt sind Sie wieder an der Reihe!

Lesen Sie zu Konstruktoren und Garbage Collection **Kapitel 9**.

Zahlen und Statisches

Statische Methoden

```
// Statische Methoden (z.B. main) brauchen keine Instanz einer
// Klasse, um ausgeführt zu werden.
public static void main (String[] args) {
    // Reguläre Methoden brauchen ein Objekt, um
    // ausgeführt zu werden. Reguläre Methoden werden über
    // den Namen einer Referenzvariablen aufgerufen.
    String text = "Bald ist Weihnachten!";
    String text2 = text.toUpperCase ();
}
```

Referenzvariable

Math-Methoden sind statisch

```
public static int round (float a) {
    // returns the closest int to the argument
}
public static int min (int a, int b) {
    // returns the smaller of two int values
}
public static int abs (int a) {
    // returns the absolute value of an int
}
```

// Das Verhalten
// der Methoden
// hängt nur von
// den Argumenten
// der Methode ab.

```
int gerundet = Math.round (7.6f);
int kleiner  = Math.min (135, 8);
int positiv  = Math.abs (-19);
```

// Statische Methoden
// werden über den Namen
// der Klasse aufgerufen.

```
Math einMathObjekt = new Math();
```

// geht nicht !!!

Geht das?


```
public class Telefonbucheintrag {
    private String name;
    private String strasse;
    private String nummer;

    // Getter & Setter
    public String getName() { return name; }
    ...
    public static void main (String[] args) {
        System.out.println ("Der Name ist " + name);
        System.out.println ("Der Name ist " + getName());
    }
}
```

Nein! Statische Methoden können weder nicht-statische Instanz-Variablen noch nicht-statische Methoden benutzen!

Welches Objekt meinen wir?

Geht das?

// Gesucht: Die Anzahl der Instanzen einer Klasse?

```
public class Telefonbucheintrag {
    private String name;
    private String strasse;
    private String nummer;
    private int anzahlDerEinträge = 0;

    public Telefonbucheintrag () {      // Der Konstruktor zählt
        anzahlDerEinträge++;            // die Anzahl der Einträge
    }

    // Getter & Setter...
}
```

Nein! Instanzvariablen werden in jedem neuen Objekt auf 0 gesetzt.

Statische Variablen

// Gesucht: Die Anzahl der Instanzen einer Klasse?

```
public class Telefonbucheintrag {
    private String name;
    private String strasse;
    private String nummer;
    public static int anzahlDerEinträge = 0;

    public Telefonbucheintrag () {      // Der Konstruktor zählt
        anzahlDerEinträge++;            // die Anzahl der Einträge
    }

    // Getter & Setter...
}
```

Statische Instanzvariablen gibt es nur einmal in der Klasse. Sie werden beim Laden der Klasse initialisiert.

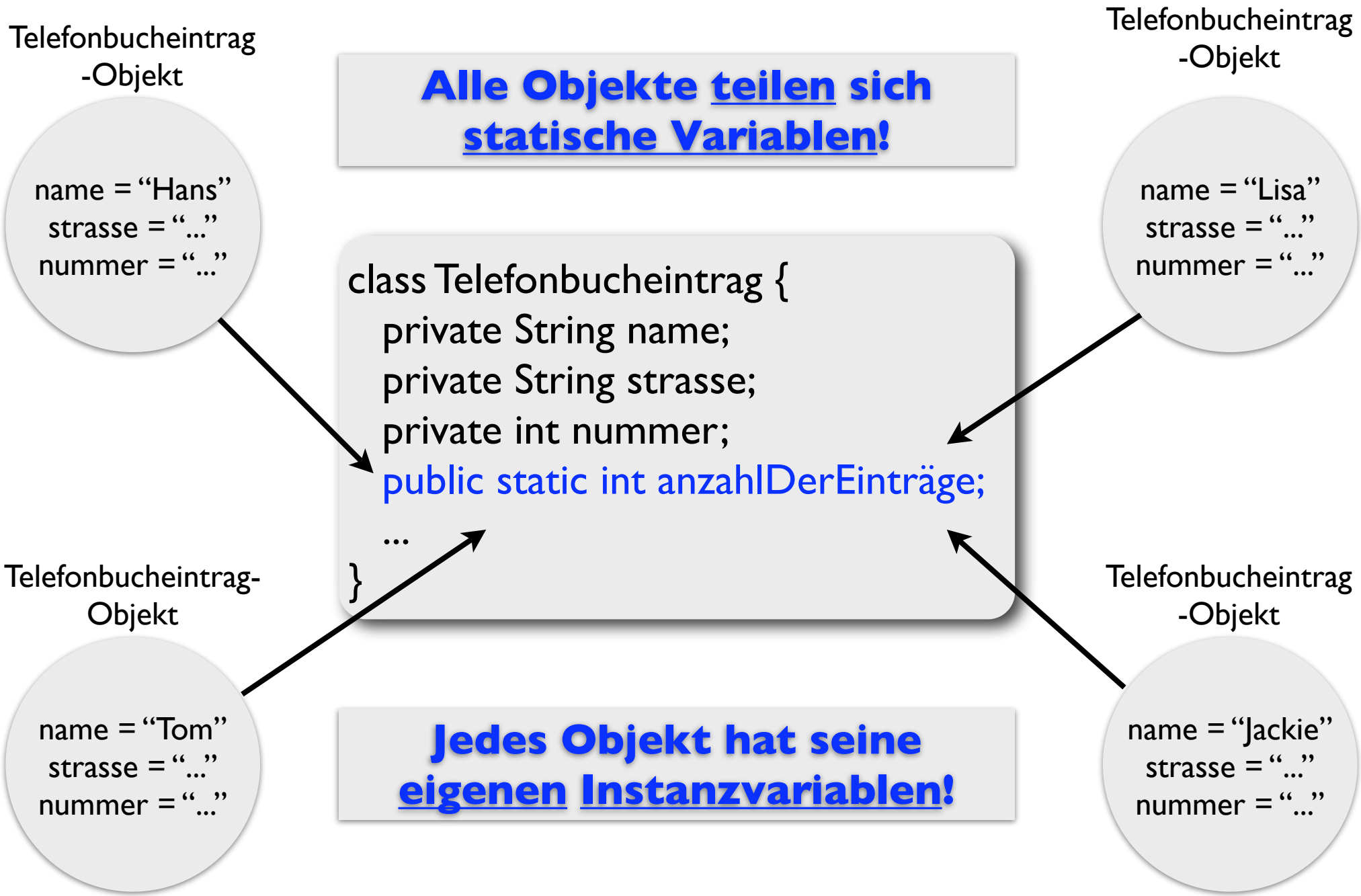
Statische Variablen

Statische Instanzvariablen haben einen Default-Wert.

```
public class Telefonbucheintrag {
    ...
    public static int anzahlDerEinträge;

    public Telefonbucheintrag () {      // Der Konstruktor zählt
        anzahlDerEinträge++;            // die Anzahl der Einträge
    }
    ...
}
```

Statische Instanzvariablen werden über den Namen der Klasse aufgerufen.

```
// irgendwo im Hauptprogramm
System.out.println (Telefonbucheintrag.anzahlDerEinträge);
```


Telefonbucheintrag -Objekt
name = "Hans"
strasse = "..."
nummer = "..."
Alle Objekte teilen sich statische Variablen!
Telefonbucheintrag -Objekt
name = "Lisa"
strasse = "..."
nummer = "..."
class Telefonbucheintrag {
private String name;
private String strasse;
private int nummer;
public static int anzahlDerEinträge;
...
}
Telefonbucheintrag-Objekt
name = "Tom"
strasse = "..."
nummer = "..."
Jedes Objekt hat seine eigenen Instanzvariablen!
Telefonbucheintrag -Objekt
name = "Jackie"
strasse = "..."
nummer = "..."

Statische finale Variablen (Konstanten)

```
public final class Math extends Object {

    public static final double PI = 3.141592653589793;
    ...
}

// Konstanten werden als static final deklariert und
// in GROßBUCHSTABEN geschrieben.

// static: Es gibt sie nur einmal pro Klasse.
// final: Der Wert kann nicht verändert werden.
```

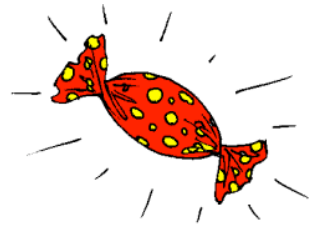
Statische finale Variablen (Konstanten)

```
public final class Math extends Object {

    public static final double PI;
    ...

    // Ein statischer Initialisierer initialisiert statische Variablen.
    // Der Initialisierer wird beim Laden der Klasse ausgeführt.
    static {
        PI = 3.141592653589793;
    }
}
```

Finale Variablen, Methoden und Klassen

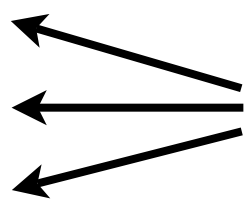
```
public int verdoppele (final int x) {
      x = x * 2;                                  // geht nicht, x ist nicht veränderbar
      return x;                                   // return x * 2; geht aber
}

public class Airbus extends VerkehrsFlugzeug {
   public final void abheben () { ... }          // abheben() ist nicht überschreibbar
}

// Klasse A380 ist nicht mehr erweiterbar
public final class A380 extends Airbus {
      private final anzahlMotoren = 4;           // anzahlMotoren konstant
      public void abheben () { ... }             // überschreiben geht nicht
}
```

Wrapper-Klassen für elementare Datentypen



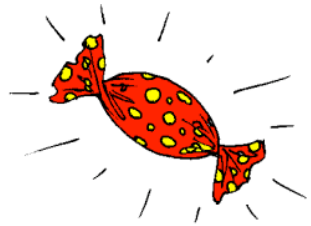
```
Benutzereingabe eingabe = new Benutzereingabe ();
String jahre = eingabe.getBenutzereingabe ("Ihr Alter?");

// Wir brauchen einen int-Wert!

int alter = Integer.parseInt (jahre);  // String → int
```

Byte, Short, Integer, Long
Float, Double
Character
Boolean



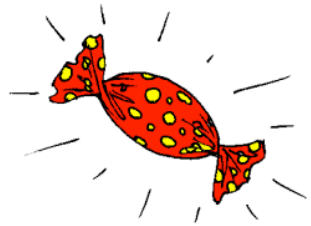
Wrapper-Klassen

Wrapper-Klassen und Autoboxing



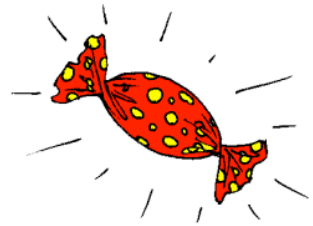
```
Integer einInt  = new Integer (15);          // Integer-Objekt
Integer auchEinInt = new Integer (alter);
Integer nochEinInt = alter;                  // autobox

Double einDouble = 27.0;                     // autobox
double auchEinDouble = einDouble;            // unbox

Float gewinn = 9.50f;
Float mehr = gewinn * 10.0f;                 // boxing & unboxing
```

ArrayList für Wrapper-Klassen



```
ArrayList<Integer> alleZahlen = new ArrayList<Integer> ();

alleZahlen.add (new Integer (23));          // speichert Objekte
alleZahlen.add (17);                        // autobox
int zahl = alleZahlen.get (0);              // unbox

ArrayList<Boolean> wahrOderFalsch = new ArrayList<Boolean> ();
wahrOderFalsch.add ("true");
wahrOderFalsch.add (false);                 // autobox

// geht nicht, elementarer Datentyp !!!
ArrayList<int> alleZahlen = new ArrayList<int> ();
```

Methoden der Wrapper-Klassen



```
String drei = "3";
int dreiAlsInt = Integer.parseInt (drei);
double preis = Double.parseDouble ("13.99");

Integer auchDrei = new Integer(3);
dreiAlsInt = auchDrei.intValue();

Double vermögen = 324325535.99;                    // autobox
double meinVermögen = vermögen.doubleValue();
boolean ichBinReich = new Boolean("true").booleanValue();

String geld = "" + meinVermögen;                   // double → String
String geldAlsString = Double.toString (meinVermögen);
String hexString = Integer.toHexString (15);    // "f"
```

Zahlen formatieren

% [Argumentnummer] [Schalter] [Breite] [.Genauigkeit] **Typ**

// _123,456.789 (mit einem führenden Leerzeichen)
String.format ("**%,12.3f**", 123456.7890);

// **%** fügt hier das Argument ein
// Schalter **,** für Tausendertrennzeichen
// Mindestbreite der Ausgabe (evtl. mit Leerzeichen auffüllen)
// **.**Genauigkeit der Nachkommastellen
// Typ ist **d** für Dezimalzahlen, **f** für Fließkommazahlen
// oder **c** fur Zeichen, **x** für Hexadezimalzahl

// Die Teile in [] können fehlen, aber der Typ ist zwingend notwendig!

Zahlen formatieren

```
String.format ("%d", 15.7);                  // falscher Typ

String.format ("%d", 15);                    // 15

String.format ("%.3f", 15.74321);            // 15.743        15,743

String.format ("%6.1f", 15.74321);           // __15.7        __15,7

String.format ("%,d", 1574321);              // 1,574,321     1.574.321

String.format ("von %,d bis %.2f", 1234, 5678.9);   // von 1.234
                                                    // bis 5678,90
```

Wie lautet die Formatierung?

SMS mit Ergebnis an:

Die SMS soll den genauen Format-Spezifizierer für die Ausgabe **34,256.0** enthalten.

% [Argumentnummer] [Schalter] [Breite] [.Genauigkeit] Typ

String.format ("?????", 34255.99);

String.format ("%,.1f", 34255.99);

Datumsangaben: Die Klasse **Date**

```
Date heute = new Date ();                    // Datumsobjekt erzeugen

// Mon Dec 11  16:49:59 CET 2006
String.format ("%tc", heute);

// nur die Zeit 04:51:56 PM
String.format ("%tr", heute);

// Monday 11 December
String.format ("%tA %td %tB", heute, heute, heute);

// letztes Argument noch einmal benutzen mit <
String.format ("%tA %<td %<tB", heute);
```

Datumsmanipulationen: Die Klasse **Calendar**

```
// Die statische Methode getInstance() liefert ein Unterklassen-
// Objekt von Calendar zurück, z.B. den gregorianischen Kalender.
Calendar cal = Calendar.getInstance();

// auf den Klausurtermin setzen :-)
cal.set (2006, 11, 15, 18, 00);          // Monate beginnen bei 0

long ms = cal.getTimeInMillis ();    // Anzahl ms seit 01.01.1970
ms += 1000 * 60 * 60 * 2;            // 2 Std. addieren
cal.setTimeInMillis (ms);            // Ende der Klausur
cal.add (cal.Date, 7);               // 7 Tage addieren
cal.set (cal.Date, 24);              // auf X-Mas setzen
```


Exception

Was stimmt hier nicht?


```
// Division ohne Rest
public int dividiere (int zähler, int nenner) {

    return zähler / nenner;

}

dividiere (12, 4);
dividiere (11, 0);  // riskante Berechnung
```

Riskantes Verhalten abfangen

```
// Division ohne Rest
public int dividiere (int zähler, int nenner) {
    try {  // riskanter Code
        return zähler / nenner;
    } catch (ArithmeticException ex) {
        System.out.println (ex.getMessage());
        ex.printStackTrace();
    }  // getMessage() und printStackTrace() geben Infos aus
}
```

Aufruf: dividiere (12, 4)

```
   // Division ohne Rest
→ public int dividiere (int zähler, int nenner) {
→     try {  // riskanter Code
→         return zähler / nenner;
      } catch (ArithmeticException ex) {
          System.out.println (ex.getMessage());
          ex.printStackTrace();
      }  // getMessage() und printStackTrace() geben Infos aus
→ }
```

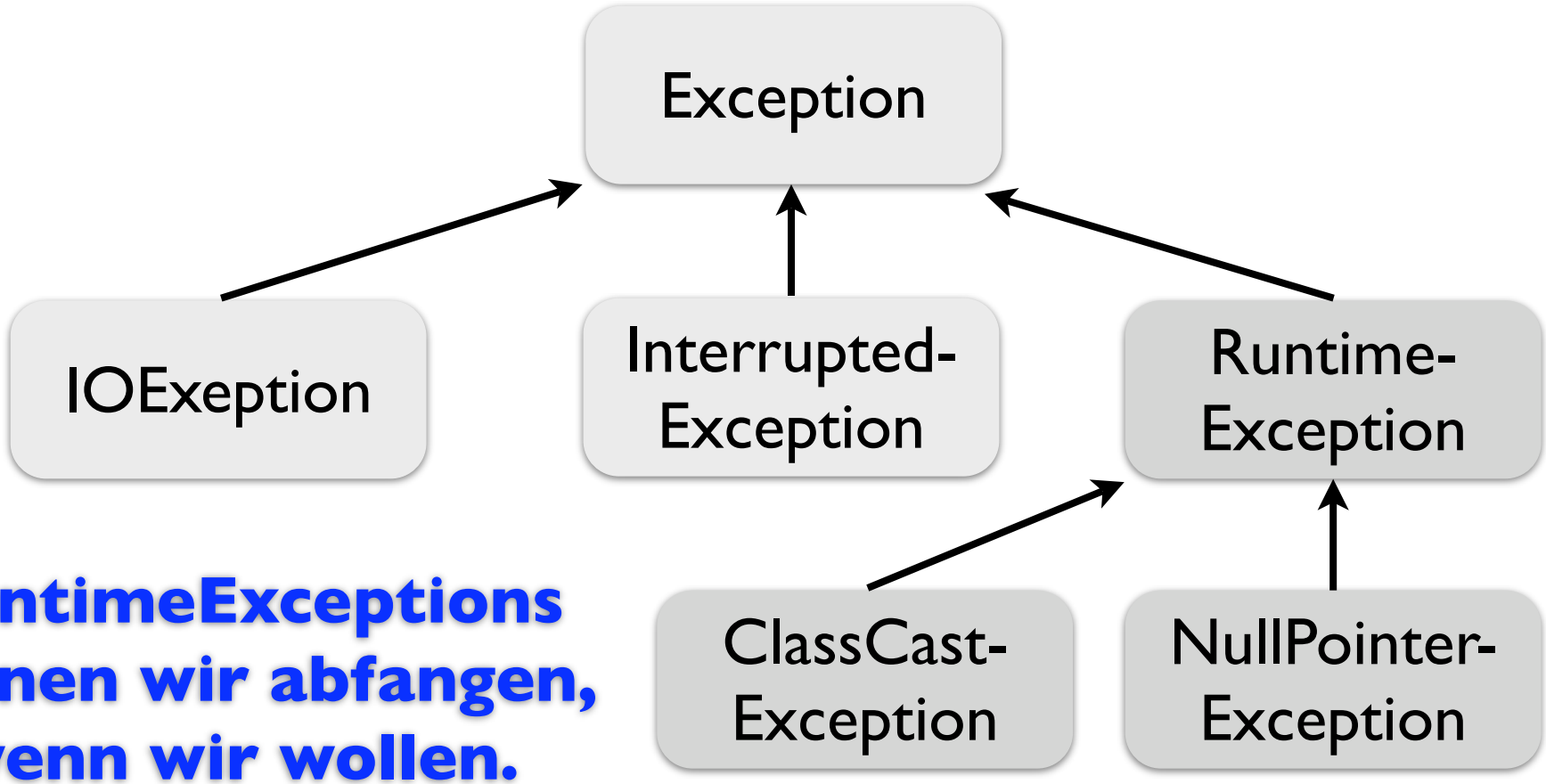
Aufruf: dividiere (11, 0)

```
   // Division ohne Rest
→ public int dividiere (int zähler, int nenner) {
→     try {  // riskanter Code
→         return zähler / nenner;
→     } catch (ArithmeticException ex) {
→         System.out.println (ex.getMessage());
→         ex.printStackTrace();
      }  // getMessage() und printStackTrace() geben Infos aus
→ }
```

Riskantes Verhalten abfangen

```
try {
    // hier kommt der riskante Code hinein
} catch (ExceptionTyp1 ex) {
    // Behandlung von ExceptionTyp 1
} catch (ExceptionTyp2 ex) {
    // Behandlung von ExceptionTyp 2
} finally {
    // dieser Code wird immer am Ende ausgeführt,
    // egal ob eine Ausnahme eingetreten ist oder nicht
}
```

Exceptions sind Objekte



RuntimeExceptions können wir abfangen, wenn wir wollen.

Für alle anderen Exceptions überprüft der Compiler, daß wir sie abfangen!

Beispiele

Alle Exceptions **müssen** abgefangen werden:

IOException,
FileNotfoundException,
CloneNotSupportedException,
UnsupportedAudiofileException

RuntimeExceptions **müssen nicht** abgefangen werden:

ArithmeticException, ClassCastException,
NullPointerException, IndexOutOfBoundsException,
NumberFormatException

Unerwartete Ereignisse melden

Das Benzin ist aus!!!

```
A380 eineA380 = new A380();
...
eineA380.abheben();
eineA380.fliegen();   // mitten im Flug geht das Benzin aus
eineA380.landen();
```

Eigene Exceptions definieren und werfen

```
public class KeinBenzinMehr extends Exception {}

public class A380 extends Airbus {
    private int benzin;  // speichert die getankte Benzinmenge

    public void fliegen () throws KeinBenzinMehr {
        ...
        if (benzin < 1) throw new KeinBenzinMehr();
        ...
    }
}
```

Unerwartete Ereignisse melden

```
A380 eineA380 = new A380();
...
eineA380.abheben();
try {
    eineA380.fliegen();  // riskanter Code
} catch (KeinBenzinMehr ex) {
    // sollte das Benzin ausgehen, dann in Luft betanken lassen
}
eineA380.landen();
```

Polymorphe Exceptions

```
public class MeineException1 extends Exception { }
public class MeineException2 extends MeineException1 { }
public class MeineException3 extends MeineException2 { }

public class Testfall1 {
    // diese Methode wirft irgendeine Exception
    public void werfeException () throws MeineException1 {
        if (bedingung1) throw new MeineException1();
        if (bedingung2) throw new MeineException2();
        if (bedingung3) throw new MeineException3();
    }
}
```

Polymorphe Exceptions

```
// Mehrfach-catch-Blöcke von Unterklasse nach Superklasse ordnen
try {
    Testfall test = new Testfall();
    test.werfeException();              // riskanter Code
} catch (MeineException3 ex) {
    // fängt MeineException3 ab
} catch (MeineException2 ex) {
    // fängt MeineException2 ab
} catch (MeineException1 ex) {
    // fängt MeineException1 ab
} catch (Exception ex) {
    // fängt alle anderen Exception-Typen hier ab
}
```

Polymorphe Exceptions

```
// Die "Superklassen-Exception" fängt alles ab !!!
try {
    Testfall test = new Testfall();
    test.werfeException();              // riskanter Code
} catch (Exception ex) {
    // fängt alle Exception-Typen bereits hier ab
} catch (MeineException3 ex) {
    // MeineException3 kommt hier nicht an
} catch (MeineException2 ex) {
    // MeineException2 kommt hier nicht an
} catch (MeineException1 ex) {
    // MeineException1 kommt hier nicht an
}
```

Exceptions weiterreichen

```
// erster() behandelt nicht die Exception, die zweiter() wirft.
// erster() reicht eine Exception an seinen Aufrufer weiter,
// wenn er deklariert, daß er die Exception wirft.
public void erster() throws MeineException1 {
  zweiter ();
}

public int zweiter() throws MeineException1 {
  if (...) throw new MeineException1();
}
```

Jetzt sind Sie wieder an der Reihe!

Lesen Sie zu Zahlen, Statisches und Exceptions
Kapitel 10 und 11.