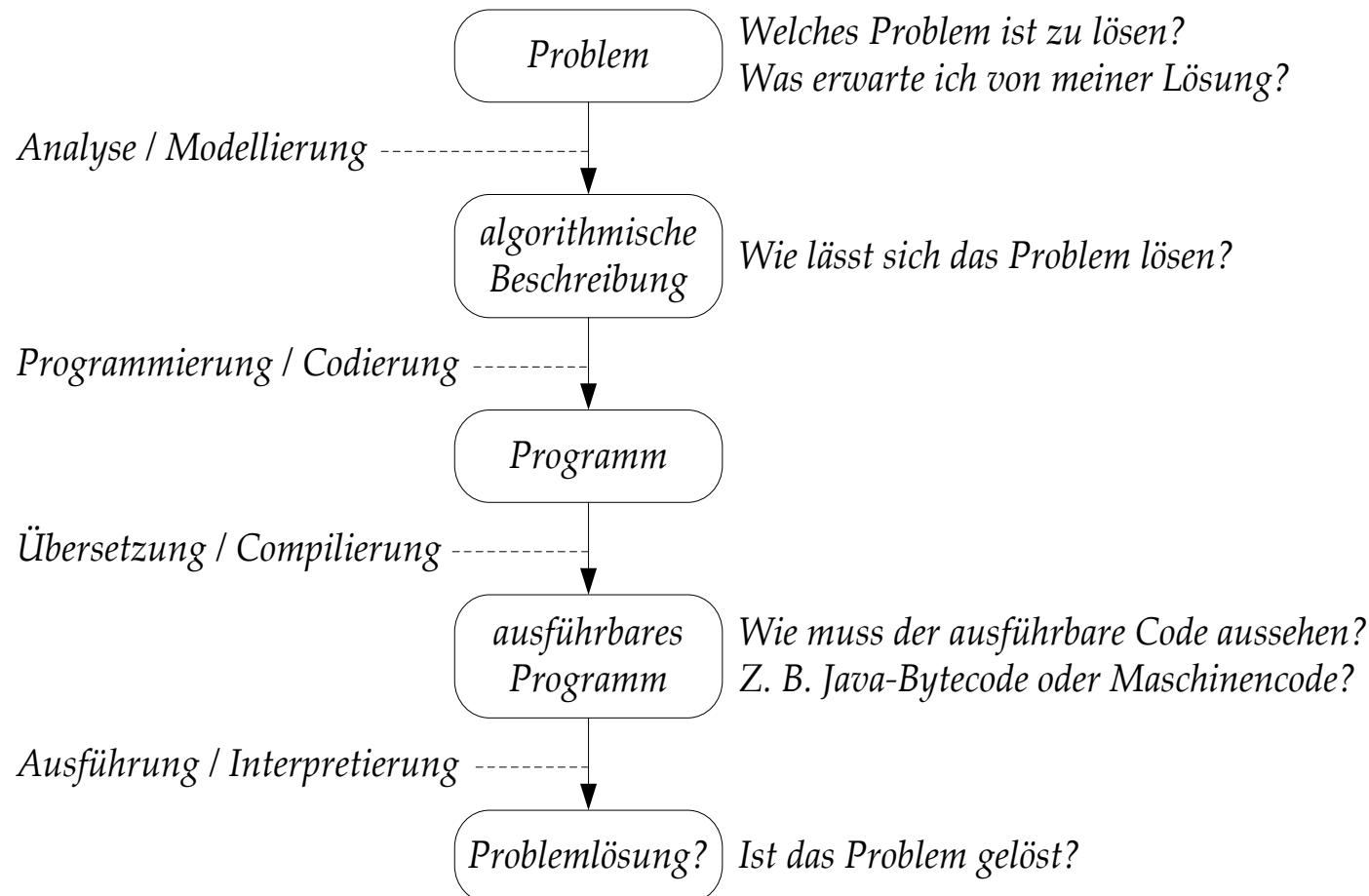
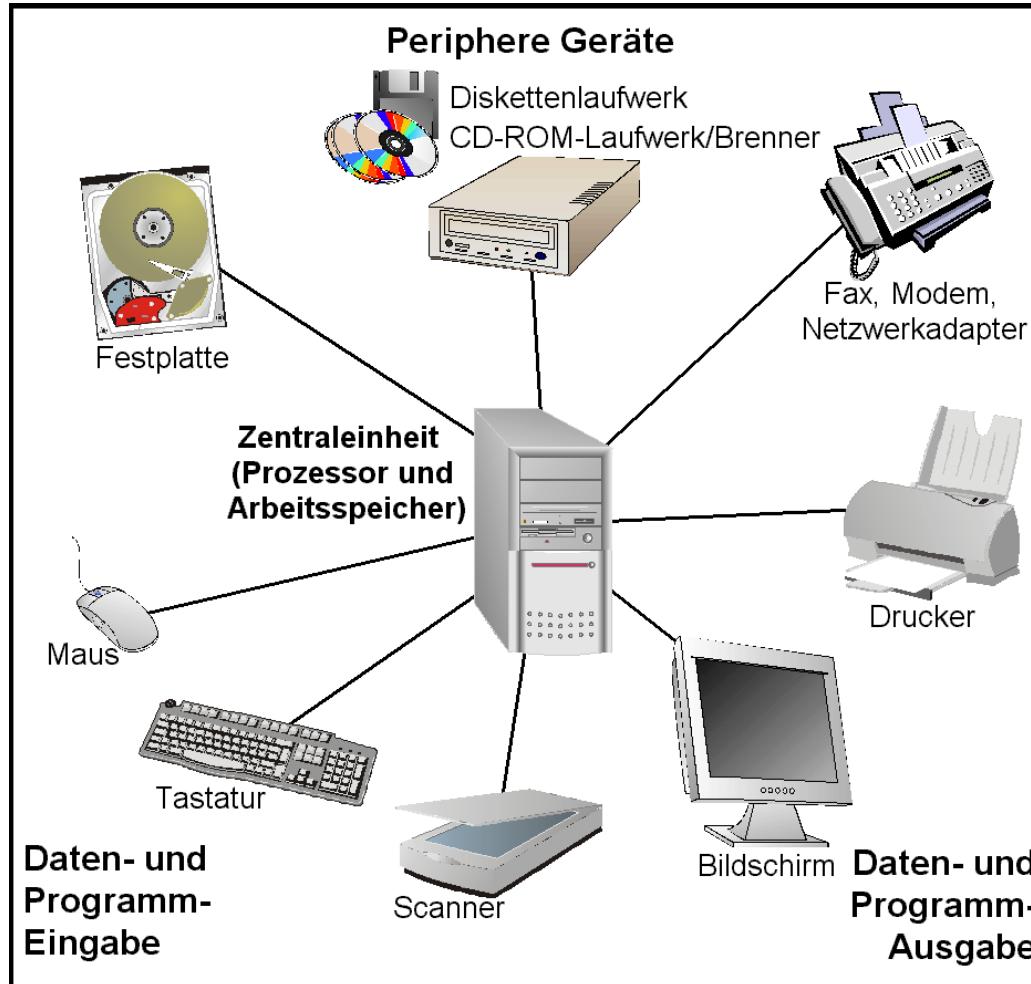
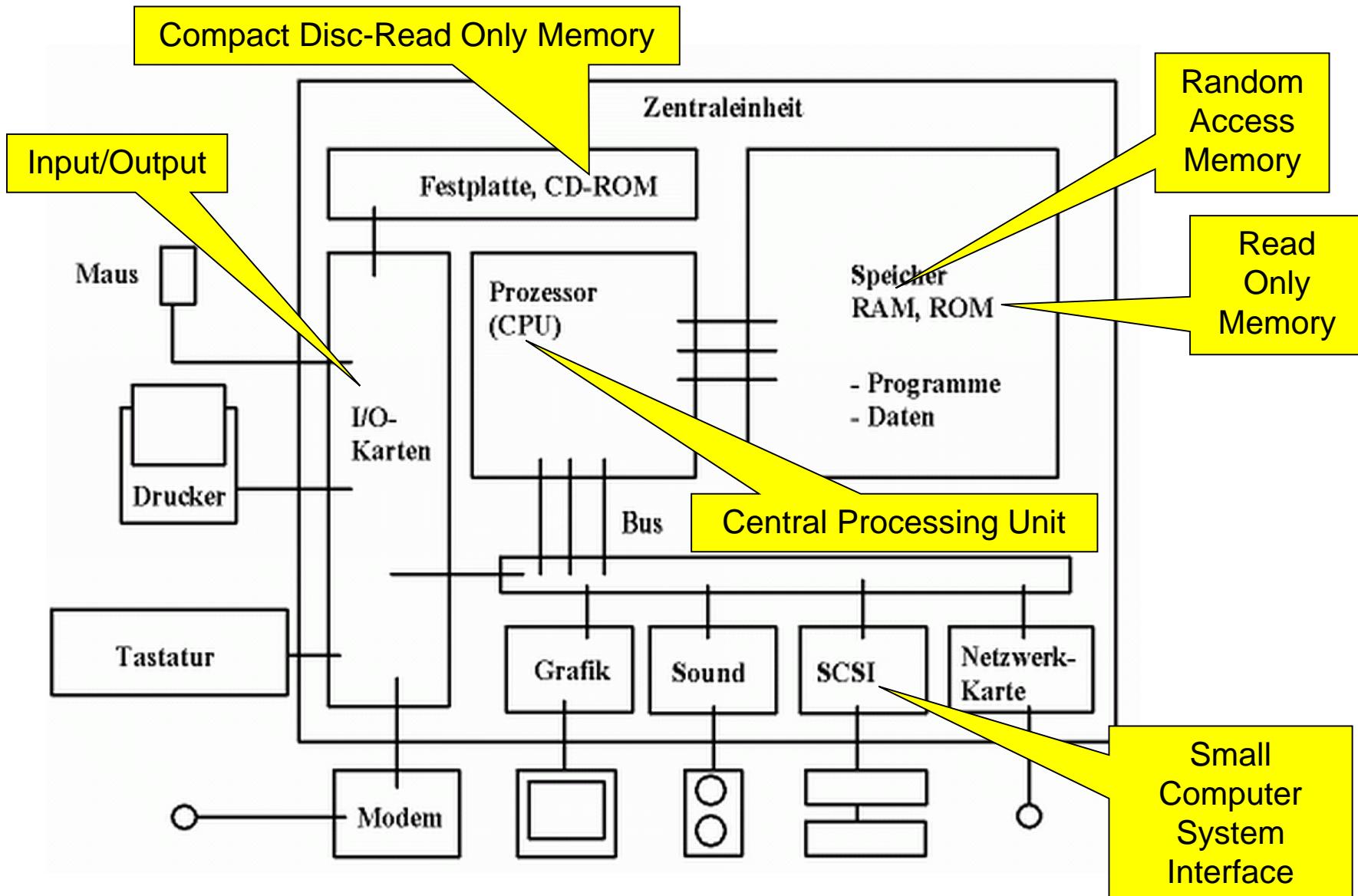


Was heißt Programmieren?

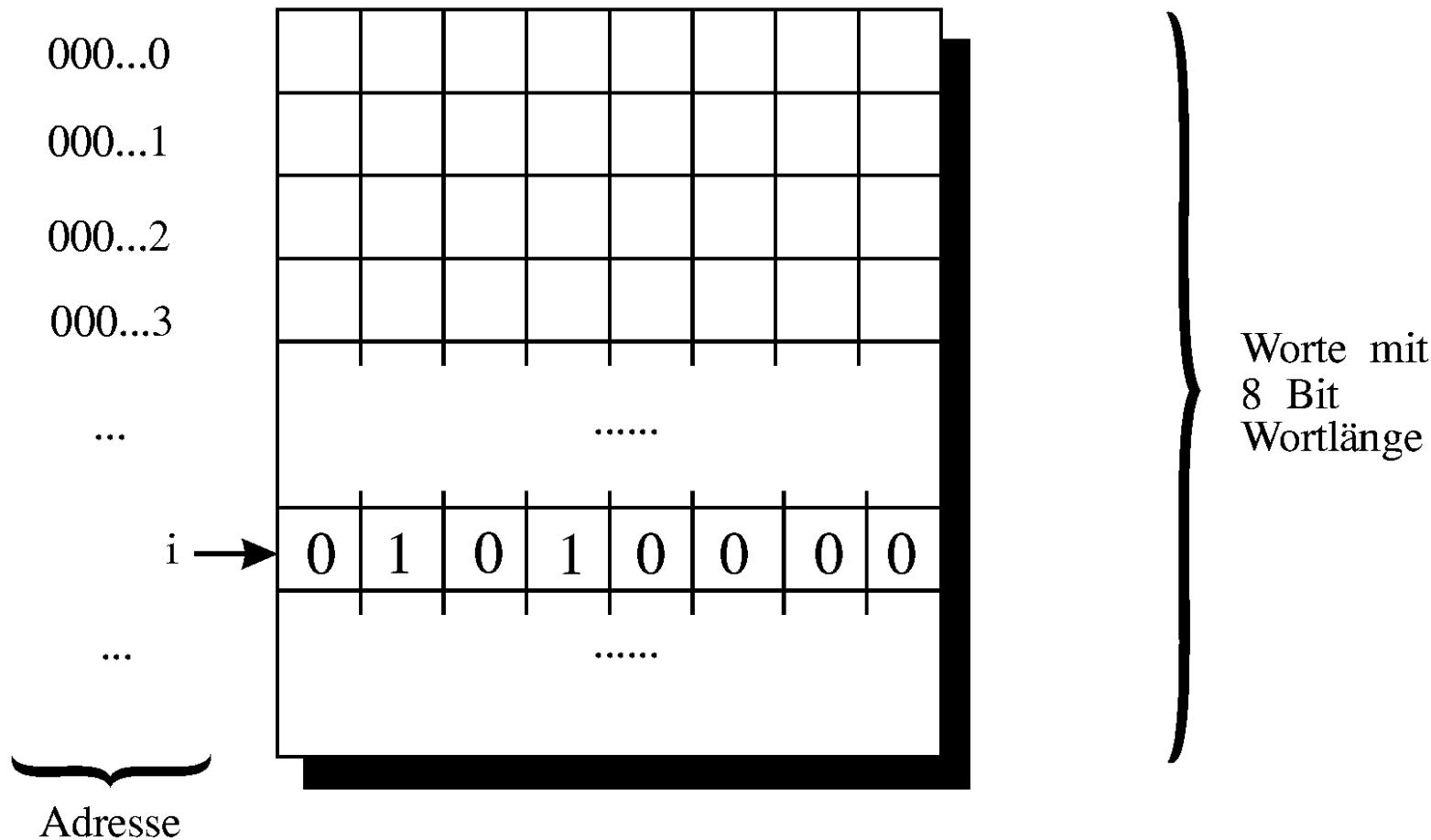
- Umsetzen eines gegebenen Algorithmus in ein lauffähiges Computer-Programm







- Einfaches Schema des Arbeitsspeichers

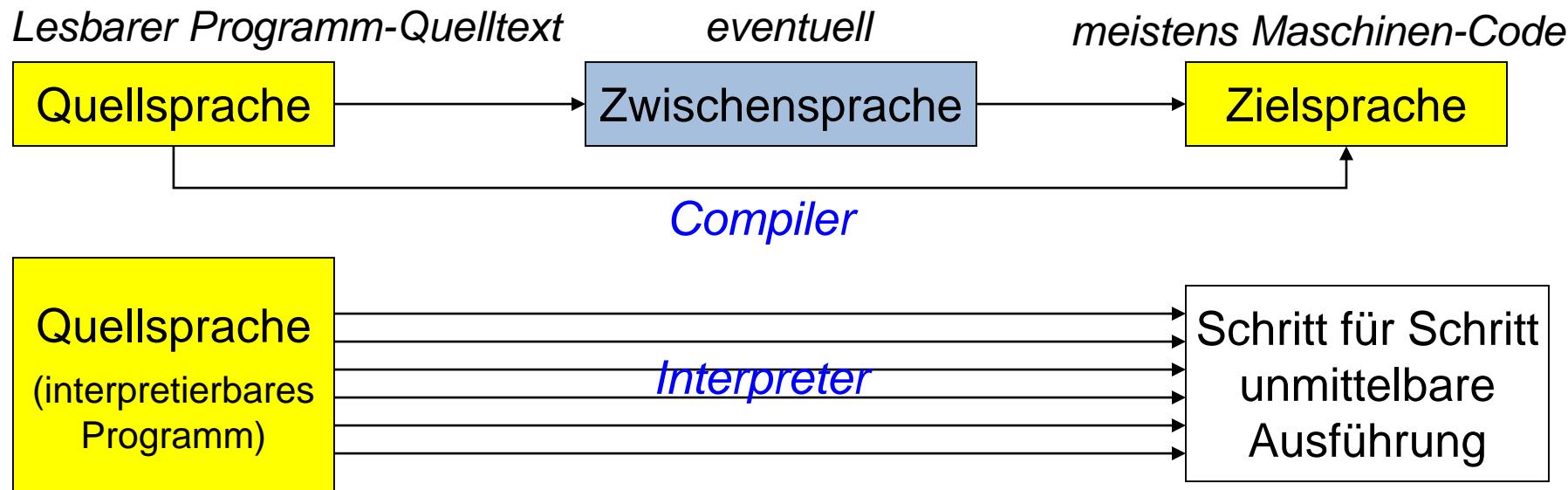


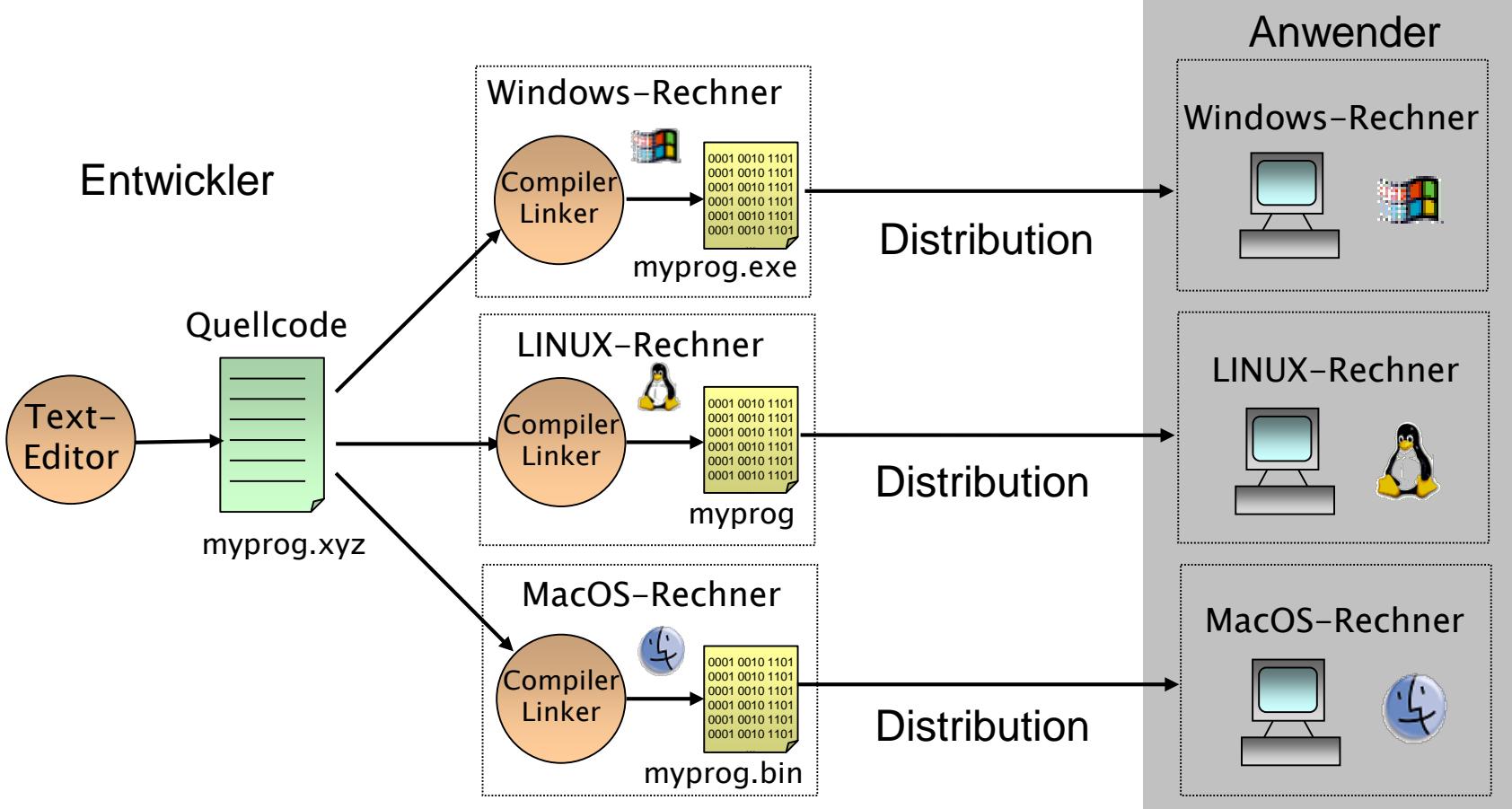
- Datenspeicherung im Arbeitsspeicher
 - einzelne Daten können sich (abhängig von ihrem Typ) aus mehreren Speicherworten zusammensetzen
- Verbindung zu Programmiersprachen
 - Adressen im Speicher erhalten symbolische Namen (z.B. Variablen)
 - Inhalte mehrerer Speicherworte werden kombiniert und ergeben bestimmte Werte (z.B. Werte von Variablen)

Klassifizierung

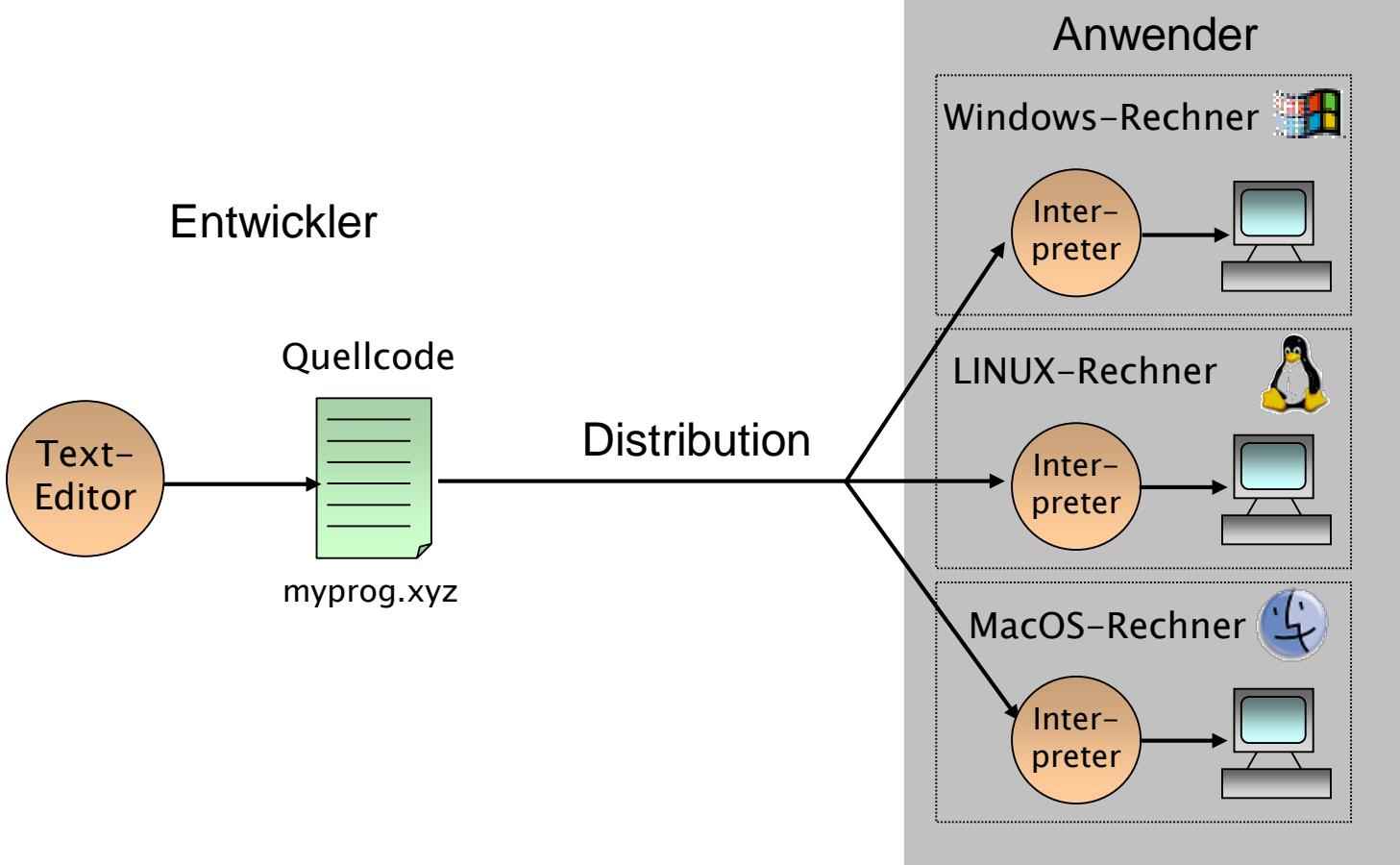
- **Betriebssysteme** (Windows, UNIX, LINUX, MacOS, iOS, Android)
- **Dienstprogramme** (Editor, Datei-Manager, Mailer, Browser)
- **Übersetzer** (Assembler, Compiler, Interpreter)
- **Anwendungsprogramme** (mit Compiler oder Interpreter erzeugt)

Compiler und Interpreter



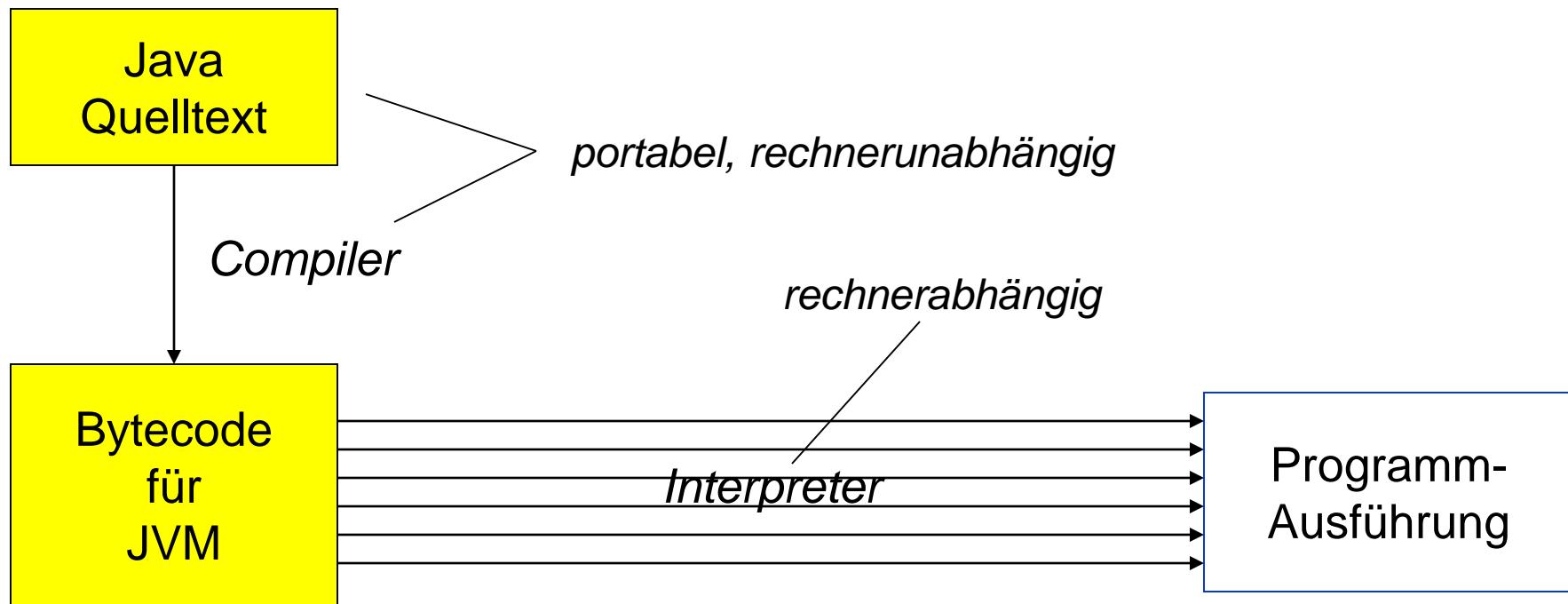


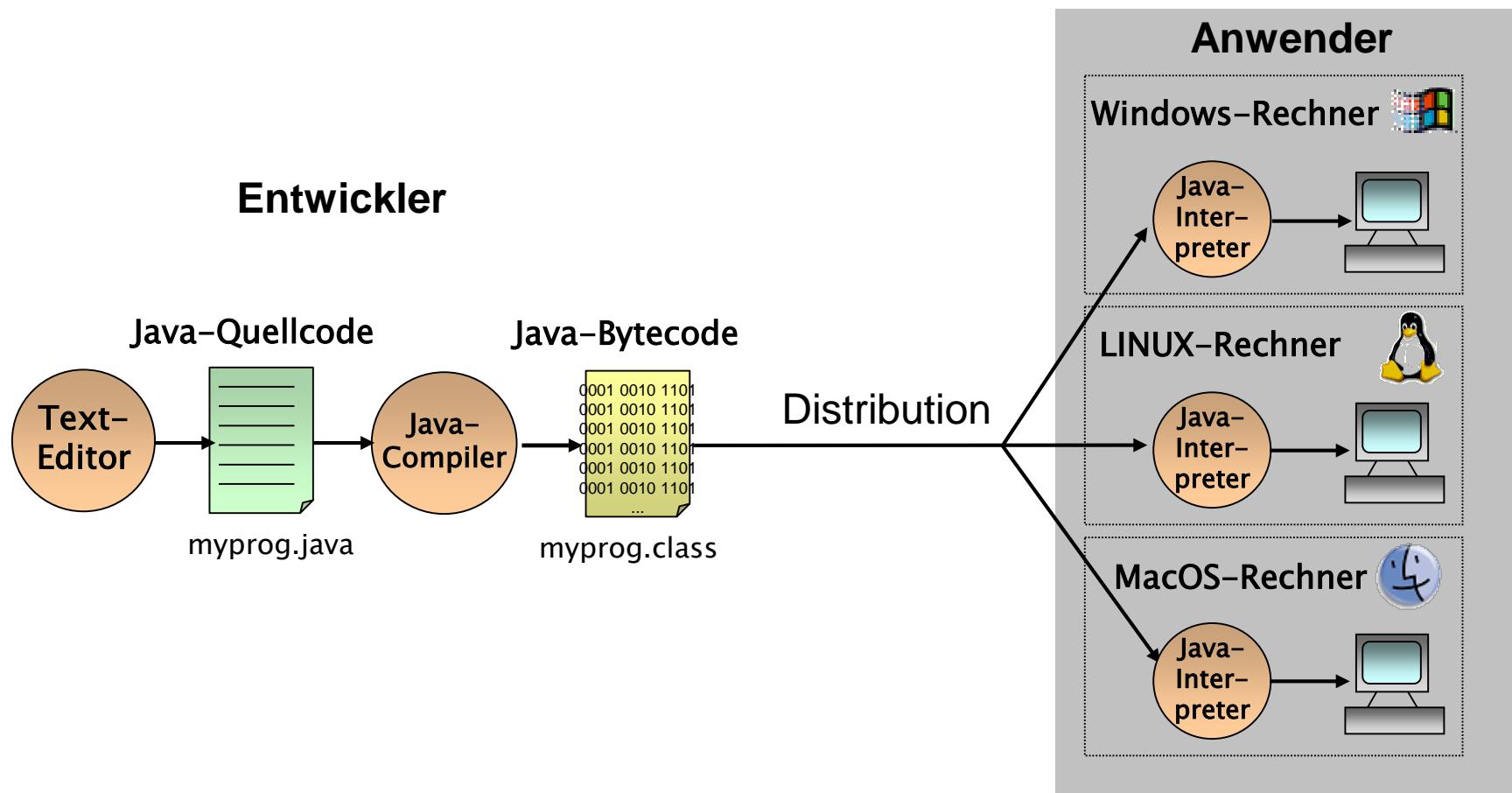
- **Vorteile:** Optimale Nutzung der Prozessoreigenschaften, hohe Abarbeitungsgeschwindigkeit, Quellcode geschützt
- **Nachteile:** Programme spezialisiert auf Zielrechner, zusätzlicher Aufwand für Anpassungen (z.B. grafische Oberflächen), verschiedene Compiler notwendig



- **Vorteile:** Quellcode direkt ausführbar, falls Interpreter verfügbar
- **Nachteile:** langsame Ausführung, verschiedene Interpreter notwendig, Quellcode lesbar

- Kombination von Compiler und Interpreter für Code einer virtuellen Maschine (**Java Virtual Machine - JVM**)





- **Vorteile:** nur ein Compiler, Sprache plattformunabhängig, Quellcode teilweise geschützt
- **Nachteile:** langsame Ausführung, verschiedene Interpreter notwendig

Ein klein wenig Geschichte

```
01101001  
00001001  
10100101  
01010001
```

Maschinensprache

```
LOAD R1 23  
MOVE R7 R2  
ADD 10 PRINT "Hello"  
LOAD 20 SET A = 7  
LOAD 30 COSINE PROC1  
ADD
```

Assembler

Frühe höhere
Programmiersprache (BASIC)

```
public class Hallo  
{  
    public static void main(String[] args)  
    { System.out.println("Hallo"); }  
}
```

Moderne höhere
Programmiersprache (Java)

- **Variable (variable)**

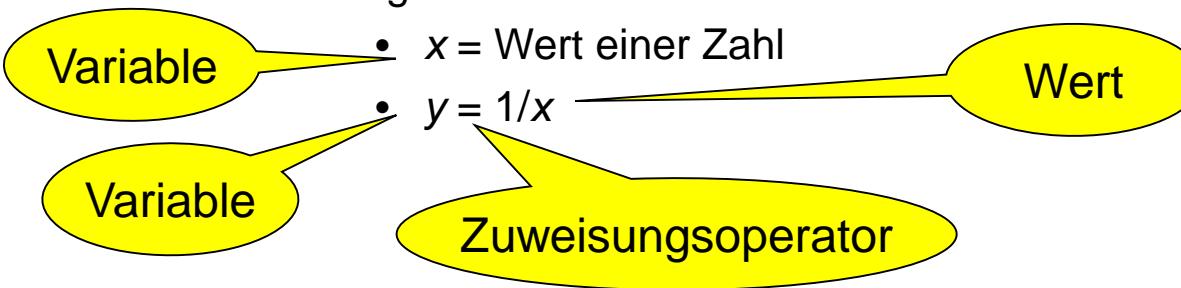
- "Platzhalter" für einen Wert eines bestimmten Typs
 - mit einer Adresse im Hauptspeicher assoziiert
- Beispiel:
 - Algorithmus A: "Kehrwert von 4"
 - dividiere 1 durch 4
 - Algorithmus B: "Kehrwert einer Zahl"
 - setze x auf den Wert der Zahl
 - dividiere 1 durch x

- **Zuweisung (Wertzuweisung) (assignment)**

- weist einer Variable einen Wert zu
- Beispiel:

- Algorithmus B': "Kehrwert einer Zahl"

- $x = \text{Wert einer Zahl}$
- $y = 1/x$



Wert

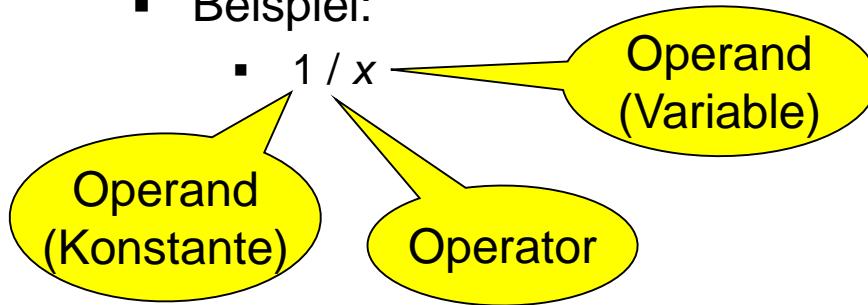
- **Datentyp (Typ) (data type)**

- Bauplan für Daten (Werte von Variablen oder Konstanten), der festlegt,
 - wie die Darstellung der Werte im Speicher erfolgt,
 - welche Operationen für die Werte erlaubt sind,
 - welche Standardwerte (Default-Werte) festgelegt sind.
- Beispiele (Variablen im Speicher – hier nur Idee (ungenau)):

Name	Adresse	Wert (Inhalt)	Typ																																
x	001...0100	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	0	0	0	1	0	1	0	0	0	1	0	1	0	0	ganze Zahl (z.B. short , 16 Bit)																
0	1	0	0	0	1	0	1																												
0	0	0	1	0	1	0	0																												
y	010...0010	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>1</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr></table>	0	1	0	1	0	1	0	1	1	1	0	0	0	1	0	1	1	1	1	0	0	1	0	1	0	1	0	0	0	1	0	1	Gleitkomma-Zahl (z.B. float , 32 Bit)
0	1	0	1	0	1	0	1																												
1	1	0	0	0	1	0	1																												
1	1	1	0	0	1	0	1																												
0	1	0	0	0	1	0	1																												
z	011...1101	<table border="1"><tr><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td><td>1</td><td>0</td><td>1</td></tr><tr><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td><td>0</td></tr></table>	0	1	0	0	0	1	0	1	1	0	0	1	0	0	1	0	Zeichen (Typ char , 16 Bit)																
0	1	0	0	0	1	0	1																												
1	0	0	1	0	0	1	0																												

- **Ausdruck (expression)**

- Kombination von Operanden und Operatoren als "Vorschrift" zur Berechnung eines Werts
- liefert immer einen Wert (Ergebniswert) ab
- Beispiel:
 - $1 / x$



- **Anweisung (statement)**

- Kombination von Ausdrücken und Methoden als "Vorschrift" zur Ausführung einer Aktion
- Beispiele:
 - $x = 5$ Wertzuweisung
 - $y = 1 / x$ Wertzuweisung
 - `print(x)` Ausgabeanweisung (Methodenaufruf "Drucke x")

Java

- Deklaration:
 - boolean a;
 - int b;
- Zuweisung:
 - a = true;
 - b = 5;
- Anweisung (Ausdruck + Zuweisung):
 - b = b + 2;
- Deklaration + Anweisung:
 - int c = b / 2;

C/C++

- Deklaration:
 - bool a; // nur in C++
 - int b;
- Zuweisung:
 - a = true; // nur in C++
 - b = 5;
- Anweisung (Ausdruck + Zuweisung):
 - b = b + 2;
- Deklaration + Anweisung:
 - int c = b / 2;

In C gibt es generell keinen Typ „boolean“ für Wahrheitswerte.
Es wird meistens integer verwendet (0 = false, 1 = true).

- **Anweisungs-Sequenz (sequence of statements)**
 - mehrere Anweisungen, die nacheinander ausgeführt werden
- **Anweisungs-Block (block)**
 - logisch zusammengefasste Anweisungen bzw. Programmteile, die als *eine* Anweisung aufgefasst werden können
- **Bedingte Anweisung / Entscheidungsanweisung (conditional statement)**
 - Anweisung mit mehreren Alternativen
- **Wiederholungsanweisung / Schleife (loop)**
 - mehrfach ausgeführter Anweisungsteil (Block)

Grundbegriffe der Programmierung (6)

Java

C/C++

- Schleife mit Block:

```
int a = 1;  
while (a < 256) {  
    a = a * 2  
}
```

- Bedingte Anweisung:

```
if (a > 5) {  
    //do something  
}  
else {  
    //do something else  
}
```

- Schleife mit Block:

```
int a = 1;  
while (a < 256) {  
    a = a * 2  
}
```

- Bedingte Anweisung:

```
if (a > 5) {  
    //do something  
}  
else {  
    //do something else  
}
```

Inhalt und Ziele

- Wie erstellt man einfache Java/C Programme?
- Was sind Grundbestandteile der Sprache?
 - Compiler, Grammatik, Grundelemente der Sprache

```
public class Kehrwert {  
    public static void main (String[] args) {  
        // Meine Kehrwertberechnung  
        double x, y;  
        x = 5;  
        y = 1 / x;  
        System.out.print("Der Kehrwert von 5 ist ");  
        System.out.println(y);  
        System.out.println("Das war's!");  
    }  
}
```

 Wortsymbole, **Schlüsselwörter**, reservierte Wörter

 frei wählbare **Bezeichner** für Namen von Variablen, Methoden, Klassen, ...
(teilweise in der Java-Klassen-Bibliothek vordefinierte Bedeutung)

Wichtig: Jede(s) ausführbare Klasse (Programm) muss eine Methode `main` haben

```
public class Kehrwert {  
    public static void main (String[ ] args) {  
        // Kehrwertberechnung  
        double x, y;  
        x = 5;  
        y = 1 / x;  
        System.out.print("Der Kehrwert von 5 ist ");  
        System.out.println(y);  
        System.out.println("Das war's!");  
    }  
}
```

Klassen- bzw. Programm-
Name

Kommentar
(vom Compiler ignoriert)

Vereinbarungen, Deklarationen

Wertzuweisung mit Konstante



```
public class Kehrwert {  
    public static void main (String[ ] args) {  
        // Kehrwertberechnung  
        double x, y;  
        x = 5;  
        y = 1 / x;  
        System.out.print("Der Kehrwert von 5 ist ");  
        System.out.println(y);  
        System.out.println("Das war's!");  
    }  
}
```

Wertzuweisung mit Ausdruck

Jede Anweisung
wird durch ;
abgeschlossen

Ausgabeanweisungen
Methodenaufrufe

auszugebender
Wert (**double**)

auszugebender
Wert (Text)

Argumentklammer

1. Editieren

Mit einem Texteditor den Programmtext erstellen und in einer Datei speichern: z.B.

```
notepad Kehrwert.java
```

2. Übersetzen

Programmtext mit dem Compiler in Java-Bytecode übersetzen:

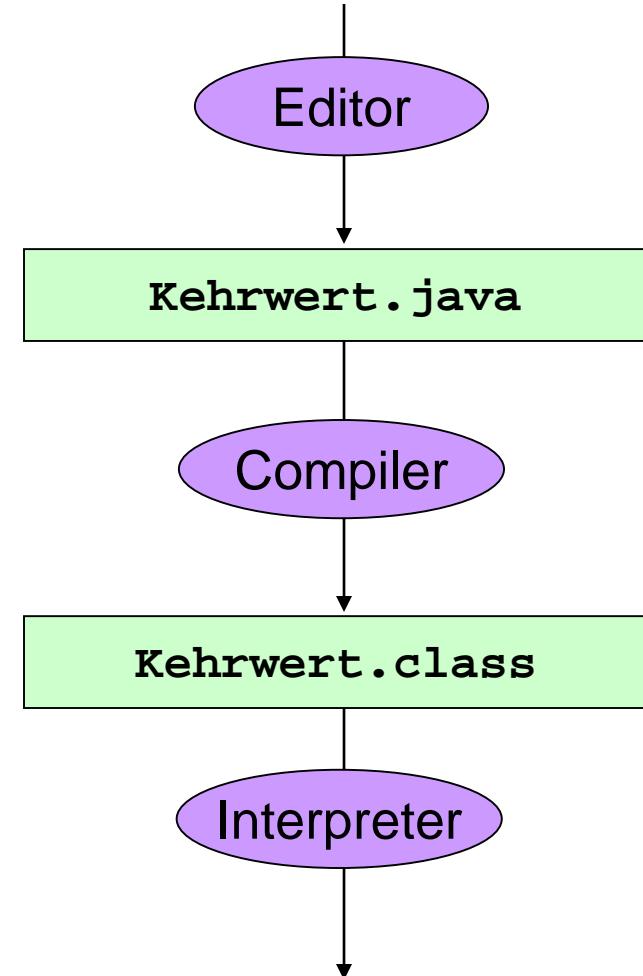
```
javac Kehrwert.java
```

Dabei entsteht die gleichnamige Datei mit Endung ".class"

3. Ausführen

Bytecode mit dem Interpreter ausführen:

```
java Kehrwert
```



```
#include <stdio.h>
int main()
{
    double x,y;
    x = 5;
    y = 1/x;
    printf("Der Kehrwert von 5 ist %f\n", y);
    printf("Das war's!\n");
    return 0;
}
```

Wortsymbole, **Schlüsselwörter**, reservierte Wörter

frei wählbare **Bezeichner** für Namen von Variablen, Methoden, Klassen, ...
(teilweise in der C-Standardsbibliothek vordefinierte Bedeutung)

1. Editieren

Mit einem Texteditor den Programmtext erstellen und in einer Datei speichern: z.B.

```
nano kehrwert.c
```

2. Übersetzen

Programmtext mit dem Compiler in Maschinencode übersetzen:

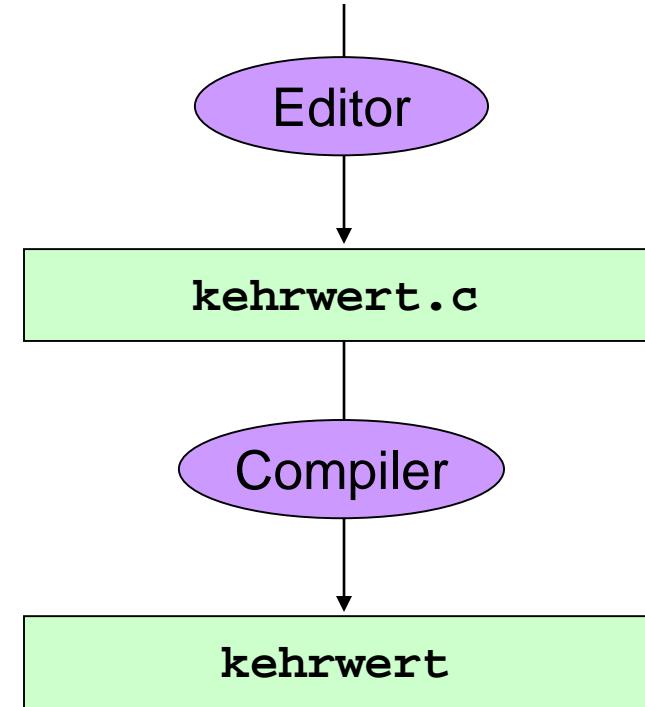
```
gcc -o kehrwert kehrwert.c
```

Dabei entsteht die ausführbaren Datei mit der Name "kehrwert"

3. Ausführen

Auf Konsole ausführen:

```
./kehrwert
```



- Der Compiler übersetzt nur "korrekte" Java-Programme
- Die Korrektheit wird anhand der Grammatik überprüft
- **Grammatik**
 - Regeln für die Bildung von "Sätzen" der Sprache, also Programmen
 - festgelegt durch das Alphabet, die Syntax und die Semantik der Sprache
 - **Alphabet:** definiert den Zeichenvorrat der zur Darstellung von Programmen verwendet werden darf
 - **Syntax:** regelt, welche Zeichenfolgen des Alphabets zulässige Sätze (Programme) der Sprache bilden.
 - **Semantik:** beschreibt die Bedeutung der einzelnen Sprachelemente und die Beziehungen zwischen ihnen, wodurch die Bedeutung des Programms festgelegt wird.

Satz	lexikalisch korrekt?	syntaktisch korrekt?	semantisch korrekt?
"hd\$,:r%."			
"Mein ist Ring das."			
"Rote Angst reiten gezaubert."			
"Das Pferd reitet auf dem Hobbit."			
"Das ist mein Ring."			

- **Java/C-Programm:** prinzipiell lediglich eine Folge von Zeichen aus dem Zeichenvorrat (sogenannte Unicode-Zeichen) von Java/C.
- **Zeichenvorrat** (character set)
 - **Buchstaben** (letters)
 - Lateinische Groß- und Klein-Buchstaben (wie im ASCII- und ISO-Latin-1-Code)
 - der Unterstrich (underscore) : _
 - das Dollarzeichen: \$
 - weitere Buchstaben aus dem Unicode (z.B. griechische Buchstaben)
 - **Ziffern** (digits)
 - Lateinische Ziffern 0 bis 9 (wie im ASCII- und ISO-Latin-1-Code)
 - weitere Ziffern aus dem Unicode (z. B. thailändische Ziffern)

- **Whitespace-Zeichen** (unsichtbare Zeichen)
 - das Leerzeichen (Blank) (space character)
 - das Zeilenendezeichen (end-of-line character)
 - das Tabulatorzeichen (tab character)
 - das Seitenvorschubzeichen (form feed / page break / new page character)
- **Sonderzeichen für Satzzeichen** (Interpunktionszeichen) (punctuation marks)
() { } [] ; , .
- **Sonderzeichen für Operatoren**
= > < ! ~ ? : & | + - * / ^ %
- **Sonderzeichen für Ersatzdarstellungen**
\ (backslash)
- **Sonderzeichen für Anführungszeichen**
, ' " " (single resp. double quotes / quotation marks)

- **Der Compiler**

- muss zunächst die Folge von Zeichen "scannen"
- sucht darin Gruppen von Zeichen
- indem nach Trennern gesucht wird
- weitere Zeichen zwischen zwei Trennern bilden ein sogenanntes **lexikalisches Element (Token)**, ein Wort der Sprache.

- **Mögliche Trenner (delimiters) in Java:**

- Whitespace-Zeichen (siehe oben)
- Kommentare
- Satzzeichen
- Operatoren

- **Kommentare:** jeweils alle Zeichen zwischen und einschließlich der Zeichen

Beginn des Kommentars	Ende des Kommentars
//	Zeilenende
/*	*/
/**	*/

- Traditionelle Kommentare
 - beginnen mit `/*` und enden mit `*/` können sich über mehrere Zeilen erstrecken (siehe Beispiel)
- einzeilige Kommentare (Zeilenkommentare)
 - beginnen mit `//` und enden am Ende der aktuellen Zeile
 - Bei C erst ab C99 unterstützt

- Dokumentationskommentare
 - werden von `/**` und `*/` eingeschlossen
 - können sich über mehrere Zeilen erstrecken
 - dienen zur Dokumentation von Programmen im Quelltext.
 - werden durch das im JDK enthaltene Programm `javadoc` aus dem Quelltext extrahiert und in ein HTML-Dokument verwandelt.
- Hinweis: Beginnen Sie jede Zeile des Textes dieser Kommentare mit einem zusätzlichen `*`.
 - Hierdurch wird der Kommentar optisch vom Quellcode abgesetzt

Beispiel JavaDoc Kommentar:

```
/**  
 * Dieses Programm berechnet den Wert von 7+11  
 */  
  
public class Kommentare {  
  
    /**hier kommt unsere main-Methode*/  
  
    public static void main ( String[ ] args ){  
  
        int summe; //Deklaration  
  
        summe = 7 + 11; //Addition  
  
        System.out.println(summe);  
  
        /* an dieser Stelle endet der Inhalt  
         * der Methode main */  
  
    }  
}
```

```
/** Dokumentationskommentare */
```

Beispiel Klasse classname:

unter `classname.java` speichern

`javac classname.java` erzeugt Bytecode `classname.class`

`javadoc classname.java` erzeugt `classname.html`

Kommentare (5)

The screenshot shows a Firefox browser window with three tabs, all titled "Kommentare". The main content area displays JavaDoc documentation for a class named "Kommendare".

Class Kommentare

```
java.lang.Object
└ Kommentare
```

Dieses Programm berechnet den Wert von 7+11

Constructor Summary

```
Kommendare()
```

Method Summary

```
static void main(java.lang.String[] args)
    hier kommt unsere main-Methode
```

Methods inherited from class java.lang.Object

```
clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait,
```

Constructor Detail

Kommendare

```
public Kommendare()
```

Method Detail

main

```
public static void main(java.lang.String[] args)
    hier kommt unsere main-Methode
```

Package Class Tree Deprecated Index Help

PREV CLASS NEXT CLASS
SUMMARY: NESTED | FIELD | CONSTR | METHOD

FRAMES NO FRAMES
DETAIL: FIELD | CONSTR | METHOD

A blue arrow points from the "hier kommt unsere main-Methode" comment in the Method Summary section to the corresponding line in the code example on the right.

Compiler, Grammatik, Grundelemente
- Kommentare

Beispiel JavaDoc Kommentar:

```
/*
 * Dieses Programm berechnet den Wert von 7+11
 */

public class Kommentare {
    /*hier kommt unsere main-Methode*/
    public static void main ( String[] args ){
        int summe; //Deklaration
        summe = 7 + 11; //Addition
        System.out.println(summe);
        /* an dieser Stelle endet der Inhalt
         * der Methode main */
    }
}
```

- Kommentare können nicht geschachtelt werden
- `/*` und `*/` haben keine besondere Bedeutung in Zeilenkommentaren
- `//` hat keine besondere Bedeutung in Kommentaren, die mit `/**` oder `/*` beginnen
- Die Bedeutung eines Kommentarzeichens ist abhängig vom Kontext

Übungen

- Warum sind ordentliche Kommentare wichtig?
- Kann innerhalb eines mehrzeiligen mit `/*` beginnenden Kommentars `*/` vorkommen?
- Wo endet der folgende Kommentar?
`/* Dieser Kommentar /* // /** endet hier */`

- **Bezeichner (identifiers)**: beliebig lange Zeichenfolge aus Buchstaben und Ziffern, beginnend mit einem Buchstaben (Unterscheidung Groß- und Kleinschreibung)
- **Schlüsselwörter / Wortsymbole (keywords)**: reservierte Wörter mit vordefinierter Bedeutung. Können **nicht** als Bezeichner verwendet werden.
- **Literele (literals)**: repräsentieren konstante Werte von Datentypen
 - Ganzzahl- oder Gleitkomma-Werte (**integer, floating point number**)
 - logische Werte (**true** und **false**)
 - Zeichen-Werte (**char**) (in zwei ' -Zeichen eingeschlossen)
 - Zeichenkettenwerte (**string**) (in zwei " -Zeichen eingeschlossen)
 - die Nullreferenz (**null**, später in der Vorlesung)
- **Satzzeichen / Interpunktionszeichen** . , ;) (} {] [
(**punctuation marks**)
- **Operatoren**

Bezeichner (identifier)

Beliebige Folge von Buchstaben (z.B. A,...,Z,a,...,z,_,\\$) und Ziffern (0,...,9)

$\overbrace{a_1 a_2 \dots a_i \dots a_n}^{\text{a sequence of letters and digits}}$

muss mit einem Buchstaben beginnen (z.B., A, ..., Z, a, ..., z, _, \$)

dürfen weder Schlüsselwörter, noch Literale (true, false, ...) sein

Methoden in `java.lang.Character`:

Java

- `Character.isJavaIdentifierStart(char)`
- `Character.isJavaIdentifierPart(char)`
- `Character.isWhitespace(char)`

- **Operatoren (operators)**

- werden durch Operatorzeichen dargestellt
 - können auch durch zwei, drei oder vier Zeichen angegeben sein.
 - z. B.: + && >> >>=

- Operatoren sind sowohl Trenner als auch Tokens (*Token = lexikalisches Element / Bitfolge definierter Struktur*)

- Die Zeichensequenz

a+b,c-d

besteht für den Compiler aus den lexikalischen Elementen

a (ein Bezeichner), **+** (ein Operator), **b** (ein Bezeichner),
, (ein Satzzeichen), **c** (ein Bezeichner), **-** (ein Operator) und
d (ein Bezeichner).

- Sinnvoll für Lesbarkeit von Programmen

- zusätzliche Trennzeichen
 - im Beispiel **a + b, c - d**

- **Regeln:**

- Innerhalb von Bezeichnern, Literalen oder Schlüsselwörtern sowie zusammengesetzten Operator-Symbolen darf kein Trennzeichen stehen.
- Unmittelbar aufeinanderfolgende Bezeichner, Zahlen oder Schlüsselwörter müssen durch mindestens ein Trennzeichen getrennt werden.

- Beispiele für zulässige Bezeichner:

Zahl_Maximum	IstBuchstabeOderZahl
AlterInJahren	GehaltInDM
WahlergebnisInProzent	_j23
\$85	αεκ

- Beispiele für unzulässige Bezeichner:

14i	class	int	gamma{14}	true
Character.isJavaLetter()				
Character.isJavaLetterOrDigit()				

```
public class MitStil {  
    public static void main (String[ ] args){  
        int a, b;  
        a = 2;  
        b = 3;  
        a = a * b;  
        b = a % b;  
        System.out.println(a);  
        System.out.println(b);  
    }  
}
```



```
public class ohnestil {public static  
void main (String[ ]args){int A,b;A  
=2;b=3;A=A*b;b=A%b;System.out.println  
(A);System.out.println(b);}}
```



Regel 1: Pro Zeile eine Anweisung!

Regel 2: Rücke zusammenhängende Teile ein!

Regel 3: Die **zusammengehörigen geschweiften Klammern** eines Blocks stehen alleine in ihrer Zeile und in der gleichen Spalte! (programmierfreundliche Variante – im Buch steht platzsparende Variante)

Regel 4: Die Befehlsblöcke in **switch-Blöcken** werden gemäß obigen Regeln formatiert!

Regel 5: **Klassennamen** beginnen immer mit einem **Großbuchstaben**! Setzt sich ein Klassennamen aus mehr als einem Wort zusammen, beginnt jedes dieser Worte mit einem Großbuchstaben.

Regel 6: **Variablen-** und **Methodennamen** beginnen immer mit einem **Kleinbuchstaben**! Setzen sich Namen aus mehr als einem Wort zusammen, beginnt jedes Wort mit einem Großbuchstaben.

Regel 7: **Konstanten** werden ausschließlich in **Großbuchstaben** geschrieben. Setzt sich eine Konstante aus mehreren Worten zusammen, so werden diese durch Unterstriche getrennt.

Regel 8: Eine Methode mit einem Rückgabewert heißt Funktion. Eine Funktion besitzt stets nur **eine einzige return-Anweisung**. Diese steht in der letzten Zeile des Methodenblocks und lautet `return result;`

Einfache Datentypen - Grundfragestellungen

- Warum ist es sinnvoll Typen von Daten zu unterscheiden?
- Was sind einfache Datentypen?
- Wie müssen wir mit diesen Datentypen umgehen?

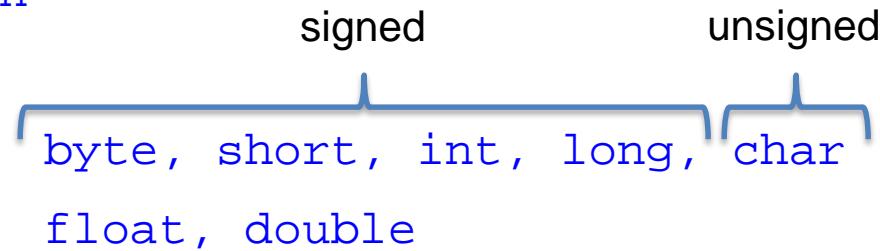
- Jedem deklarierten Bezeichner muss genau ein **Typ** zugeordnet sein
- Dadurch wird implizit festgelegt:
- welche Operationen für den Bezeichner definiert sind
 - wie viel Speicherplatz zu reservieren ist
 - welche Werte dem jeweiligen Speicherinhalt entsprechen

- Mit jedem Typ ist ein **Wertebereich** (domain) festgelegt
 - Menge der Werte, welche eine Variable des Typs annehmen kann
- Java/C sind Sprachen mit strenger **Typprüfung** (type-checking)
 - Jeder Variable und jedem Ausdruck ist ein zur Übersetzungszeit bekannter Typ zugeordnet
- Generell wird zw. “strongly typed” und “weakly typed” Programmiersprachen unterschieden
 - Keine einheitliche Definition
 - siehe http://en.wikipedia.org/wiki/Strong_and_weak_typing

- Java unterscheidet

1. Einfache (elementare) Datentypen (primitive data types)

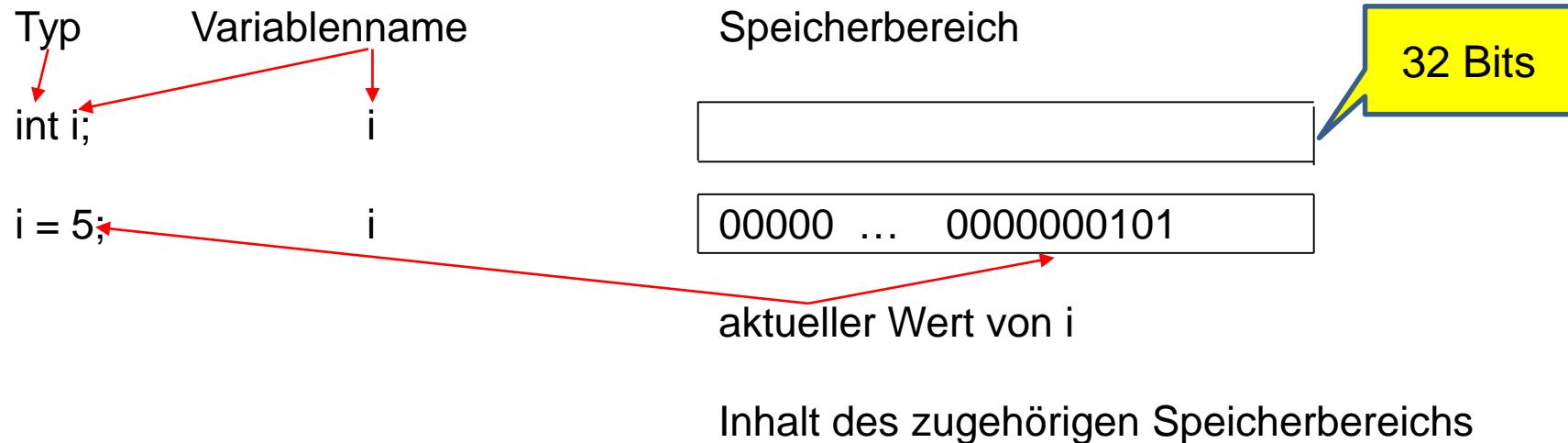
- der logische Typ `boolean`
- numerische Typen
 - ganzzahlige Typen
 - Gleitkommatypen



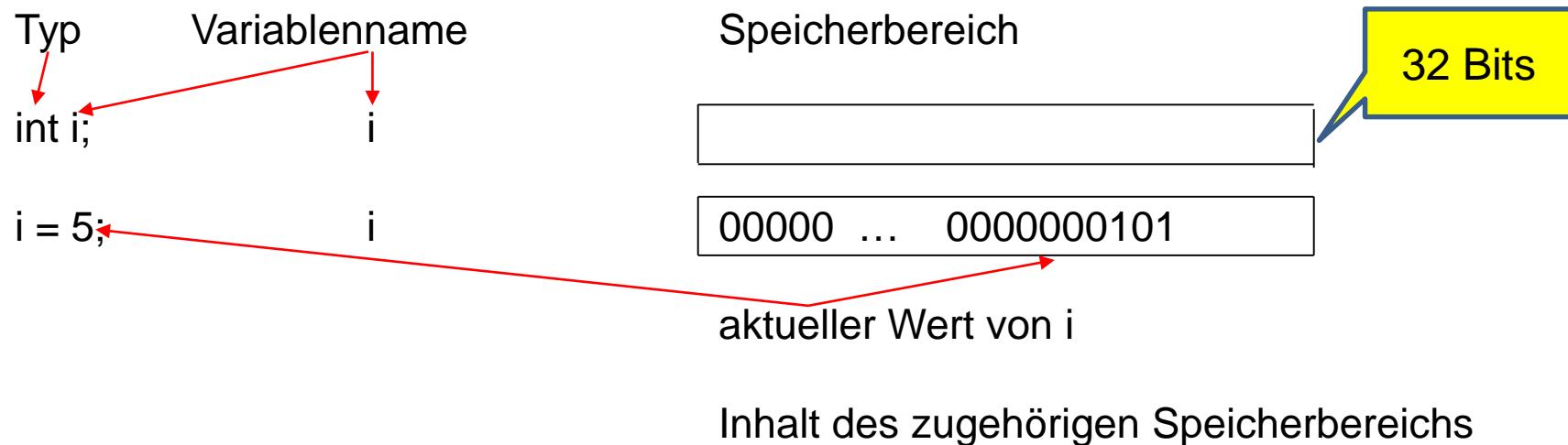
2. Referenztypen (reference types)

- Klassen, Interfaces und Felder (später)
- Bemerkung
 - `char` ist gleichzeitig auch ein Zeichentyp
 - Die Klasse `String` spielt eine Sonderrolle für Zeichenketten

Einfache (elementare) Datentypen



Einfache (elementare) Datentypen



Referenztypen (z.B. Klassen, Interfaces und Felder)



- für Fließkommazahlen / Gleitpunktzahlen (**floating point numbers**)
- Java realisiert hier den IEEE-754-Standard für die Typen `float` (einfache Genauigkeit realisiert mit 4 Byte) und `double` (doppelte Genauigkeit realisiert mit 8 Byte)
- Darstellung von Gleitkommatypen → siehe Anlage
- Gleitkommatypen (Länge in C/C++ ist Plattform-abhängig):

Typname	größter positiver Wert	kleinster positiver Wert	Länge
<code>float</code>	≈3.40282347E38	≈1.4E-45	32 Bits
<code>double</code>	≈1.7976931348623157E308	≈4.9E-324	64 Bits

- Gleitkomma-Literalkonstanten

- bestehen aus Mantisse Exponential-Anteil Typ-Suffix

- nur die Mantissen-Angabe ist zwingend

- für float

1.5f 1.f .35f 1e7f 1.3e7f

- für double

1.5 1. .35 1e7 1.3e7d

e bzw. E mit anschließender Zahl X steht für die Multiplikation mit 10^X

Beispiele: 1.78e4 entspricht $1.78 \cdot 10^4 = 1.78 \cdot 10000 = 17800$

2.4e-3 entspricht $2.4 \cdot 10^{-3} = 2.4 \cdot 0.001 = 0.0024$

- Negative Zahlen werden erzeugt, indem man vor die entsprechende Zahl ein Minuszeichen setzt. Dies ist dann ein Ausdruck und kein Literal.
- Achtung: Mit 32 oder auch 64 Bits kann nicht jede Zahl zwischen +3.4028235E38 und -3.4028235E38 exakt dargestellt werden → Rundungsfehler

Gleitkommatypen (3)

Java

C/C++

- Ein Gleitkommaliteral ist vom Typ **double**, es sei denn es endet mit **f** oder **F**. In diesem Fall besitzt es den Typ **float**.

Beispiel:	3.141592E38f	→ Typ float
	3.141592e40F	→ Typ float
	3.141592E300	→ Typ double
	3.141592e300d	→ Typ double
	3.141592e300D	→ Typ double
	.0E+3d	→ Typ double

- Aber nachfolgend liegen keine Gleitkommaliterale vor:

0,235	3.0d+3	
3.5.7	22.+2	Die meisten sind Ausdrücke
+.17	-1.43	

Logischer Datentyp

- Logischer Datentyp
 - Typname `boolean` (Java)
 - Typname `bool` (C++)
 - Länge ist typischerweise 1 Byte
 - Literalkonstanten: `false` und `true`

Java

C++

In C gibt es generell keinen Typ „boolean“.

C

Es wird meistens `integer` verwendet (`0 = false, 1 = true`).

- Zeichen-Datentypen (character data types)

- einzelne Zeichen

- Typname **char**
 - Länge 16 Bits (2 Bytes in Java, meist 1 Byte in C)
 - Literalkonstanten (in Hochkommas eingeschlossen)

'a' 'x' '+' 'ä'

'\n' Ersatzdarstellung für 'Zeilenende'

'\t' Ersatzdarstellung für 'Tabulator'

'\'' Ersatzdarstellung für '' (da unzulässig)

'\"' Ersatzdarstellung für '\"' (aber zulässig)

'\\' Ersatzdarstellung für '\\' (da unzulässig)

- eigentlich kein einfacher Datentyp, siehe später: Klasse **String**
- Literalkonstanten (in Doppelhochkommas eingeschlossen)
`"HalliHallo"`
`"öakljölkj"`
`"Ach du lieber Himmel"`
`"Ach\ndu\nlieber\nnHimmel,"` (eigentlich mehrzeilig)

Bemerkung: In C gibt es keine Strings!

Es werden Felder (Arrays) vom Typ **char** verwendet.

C

Bisher haben wir nur **lokale Variablen** in der Methode `main` verwendet.

Für diese muss stets explizit ein Wert festgelegt werden.

In vielen Fällen wird später Variablen automatisch ein Wert zugewiesen, auch wenn keine explizite Wertzuweisung erfolgt - siehe später **Instanzvariablen**.

Solche Werte werden **Default-Werte** genannt. Für die bisher behandelten Typen sind die folgenden Default-Werte festgelegt:

Default-Werte

- 0 bei numerischen Datentypen
- (ganzzahlig 0, Fließkomma 0.0)
- `false` bei `boolean`
- \u0000 bei `char`

Bemerkung: In C gibt es keine Default-Werte!

C

- Beispielhafte Deklarationen mit Initialisierungen

```
byte b = 5;  
short s = 50;  
int i = 500;  
long l = 5000;  
float f = 1.5f;  
double d = 2.5;  
char c = 'a';  
boolean x = true;
```

char – 16 Bits

- Frage: Welche der nachfolgenden Zuweisungen ist (un-)zulässig?

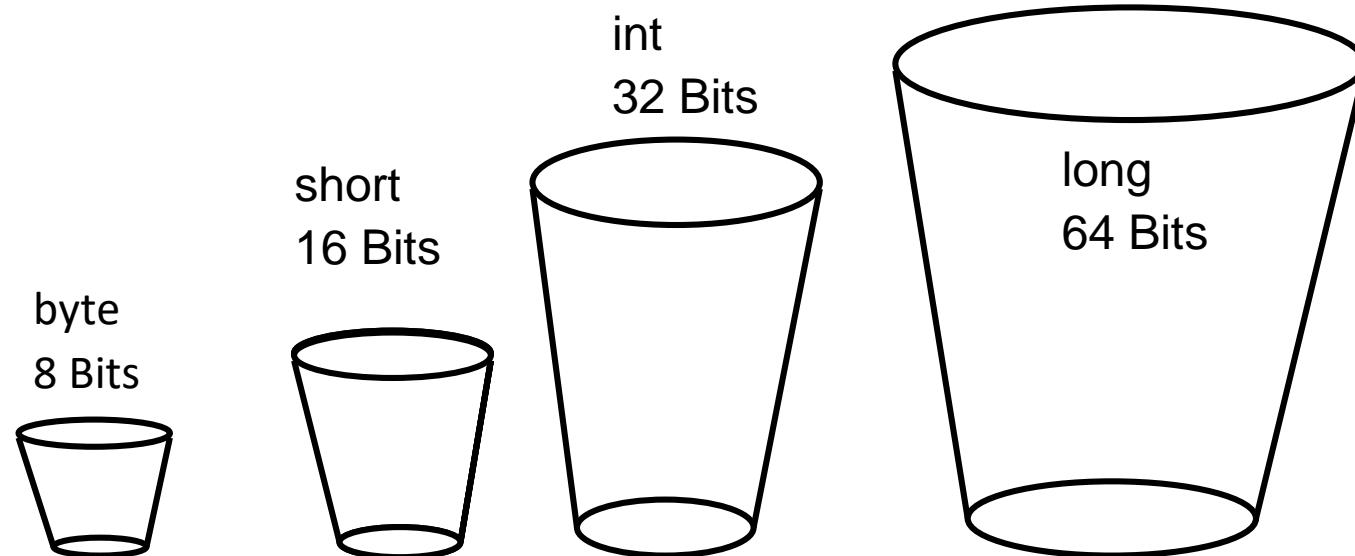
i = b; //	erlaubt
f = l; //	erlaubt aber Datenverlust möglich
i = c; //	erlaubt
s = f; //	unzulaessig!
i = l; //	unzulaessig!
i = x; //	unzulaessig!

Typen kann man sich als Becher, oder auch als Schablonen vorstellen.

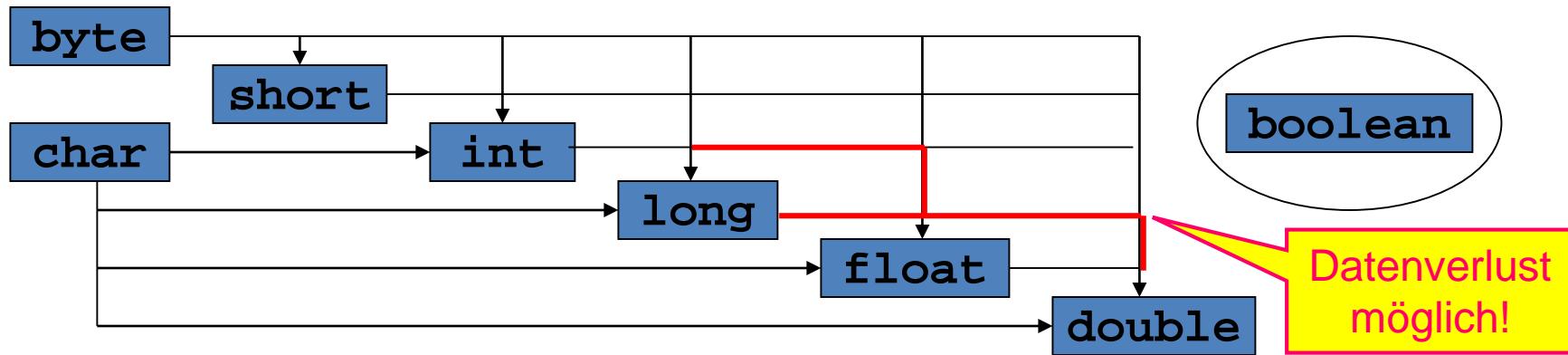
Aber: Wenn Sie den Inhalt eines großen Bechers in einen kleinen schütten, so geht oft etwas verloren.

Die Größe eines Bechers bezieht sich auf den Wertebereich (value range).

Beachten Sie, dass `int` und `float` beide 32 Bits haben und `long` und `double` beide 64 Bits haben. Die Darstellung der betreffenden Zahlen ist aber eine andere – Idee Schablone



1. Regel für die **automatische (implizite)** Typumwandlung



2. Explizite Typumwandlung (eventuell mit Datenverlust!)

- durch Voranstellen des eingeklammerten Typnamens
- in den Beispielen:

```
s = (short) f;
```

```
i = (int) l;
```

```
i = (int) x; // unzulässig, auch umgekehrt!
```

- Beispiel für einen Ausdruck:
 - `3.5 * x - 2 / (4 + y)`
- Ein Ausdruck (**expression**) setzt sich zusammen aus
 - Operanden (Konstanten, Variablen, Methodenaufrufe)
 - Operatoren (+, -, *, ...)

und *liefert stets einen Wert ab!*
- Abhängig vom *Typ der Operanden* sind nur bestimmte Operatoren zulässig, z.B.
 - arithmetische Operatoren für die *numerischen Datentypen*
 - Vergleichs-Operatoren für die *numerischen Datentypen*
 - logische Operatoren für den *logischen Datentyp*
 - der Operator + für Zeichenketten (*Typ String*)
- Man unterscheidet
 - einstellige (monadische, unäre) Operatoren (z.B. +, -, ++, --)
 - zweistellige (dyadische, binäre) Operatoren (z.B. +, *, &&, <, <=)
 - dreistellige (triadische, ternäre) Operatoren (nur einer: ?:)

1. Präfix Notation

<Operator><Operand>

Beispiel: `-a`

2. Postfix Notation

<Operand><Operator>

Beispiel: `n++`

3. Infix Notation

<Operand><Operator><Operand>

Beispiel `a+b`

oder

<Operand>?<Operand>:<Operand>

Beispiel: bedingte Zuweisung 3-stellig `max = (a>b)?a:b;`

- Elementare (also einfachste) Ausdrücke sind
 - nur ein Operand und kein Operator
 - z.B.
 - x
 - 1.74
 - Math.PI
 - Math.E
 - nur ein einfacher Methodenaufruf
 - z.B.
 - Math.sin(x)
 - Math.sqrt(2)
- Die Argumente von Methodenaufrufen können natürlich selbst wieder Ausdrücke sein
- Elementarfunktionen der Mathematik (z.B. exp, sin, cos, tan, ...) sowie Näherungen für die Konstanten e und π sind in Java in die Klasse Math ausgelagert

- Die Operatoren:

- + Identität (ohne Wirkung)
- Negation
- ++ Inkrement (Erhöhung um 1)
- Dekrement (Erniedrigung um 1)
- Verwendung von +, -, ++ und -- in Präfix-Notation (vor-gestellt)
- Verwendung von ++ und -- in Postfix-Notation (nach-gestellt)

- Beispiele für `int x = 5`

`x` vor der Auswertung

5	5	5	5	5	5
---	---	---	---	---	---

Ausdruck

+x	-x	++x	--x	x++	x--
----	----	-----	-----	-----	-----

Wert des Ausdrucks

5	-5	6	4	5	5
---	----	---	---	---	---

`x` nach der Auswertung

5	5	6	4	6	4
---	---	---	---	---	---

- Beispiel

- Programmfragment:

```
int a, b, c, d;  
  
a = 5;  
  
a++;  
  
b = ++a;  
  
c = a++;  
  
d = a--;  
  
a = a++;
```

```
a ← 5  
a ← 6  
a ← 7, Wert rechts: 7, b ← 7  
Wert rechts: 7, c ← 7, a ← 8  
Wert rechts: 8, d ← 8, a ← 7  
Wert rechts: 7, a ← 7, a ← 8
```

- Welche Werte haben a b c d am Ende?
- Für Gleitkommatypen analog

- Die Operatoren:

- + Addition
- Subtraktion
- * Multiplikation
- / Division
- % Rest (bei Division mit ganzzahligem Ergebnis)

- Beispiele

- `int i = 7 / 3;` // ==> i = 2
- `int j = 7 % 3;` // ==> j = 1
- `double d = 8.2 / 4.0;` // ==> d = 2.05
- `double e = 8.2 % 4.0;` // ==> e = 0.2
= 0.19999999999999993
- `int x = 5;`
- `int y = (x++) + (x++);` // ==> y = 11
// ==> x = 7

- Der Operator + für Strings
 - bewirkt das Aneinanderhängen von Zeichenketten
- Zusammengesetzte Zuweisungsoperatoren
(compound assignment operators)
 - zur Abkürzung von Zuweisungen mit einfachen Operationen
 - die Anweisung `a += b;` // entspricht der Anweisung `a = a + b;`
 - die Anweisung `x |= y;` // entspricht der Anweisung `x = x | y;`
 - **Vorsicht:** Die Lesbarkeit eines Programms verschlechtert sich!

- Die Operatoren:

- < Test auf "kleiner"
- > Test auf "größer"
- <= Test auf "kleiner oder gleich"
- >= Test auf "größer oder gleich"
- == Test auf "gleich"
- != Test auf "ungleich"

- Beispiel:

```
int x = ...;  
boolean b = (x <= 5);  
System.out.println("5.2 <= 4 is " + (5.2 <= 4));
```

- Ausgabe: 5.2 <= 4 is false

Logische Operatoren

Java

C/C++

- Die Operatoren:

&	logisches UND
	logisches ODER
^	logisches exklusives ODER (entweder oder)
&&	logisches UND
	logisches ODER
!	logische NEGATION (Nicht)

- Doppelzeichen && und || ermöglichen vorzeitigen Abbruch der Ausdrucks-Auswertung, sobald das Ergebnis feststeht
- Beispiel:

```
int a = InputHelper.readInteger("a = ");
int b = InputHelper.readInteger("b = ");
boolean wahrOderFalsch = (a != 0) & (b/a > 5);
```

Java

- Die Operatoren:

&	logisches UND
	logisches ODER
^	logisches exklusives ODER (entweder oder)
&&	logisches UND
	logisches ODER
!	Logische NEGATION (Nicht)

- Doppelzeichen **&&** und **||** ermöglichen vorzeitigen Abbruch der Ausdrucks-Auswertung, sobald das Ergebnis feststeht
- Beispiel:

```
int a = InputHelper.readInteger("a = ");
int b = InputHelper.readInteger("b = ");
boolean wahrOderFalsch = (a != 0) & (b/a > 5);
```

- Was passiert bei Eingabe von 0 für **a**?

Programmabsturz wegen Division durch 0

- Was passiert, wenn statt **&** ein **&&** verwendet wird?

alles ok: **(a != 0)** liefert **false**, und **(b/a>5)** wird nicht ausgewertet

& logisches UND		
	true	false
true	true	false
false	false	false

&& logisches UND		
	true	false
true	true	false
false	false	false

Unterschied:

A & B : beide Ausdrücke A und B werden ausgewertet

A && B : liefert **A** den Wert **false**,
so wird **B nicht mehr ausgewertet**

logisches ODER		
	true	false
true	true	true
false	true	false

logisches ODER		
	true	false
true	true	true
false	true	false

Unterschied:

A | B : beide Ausdrücke A und B werden ausgewertet

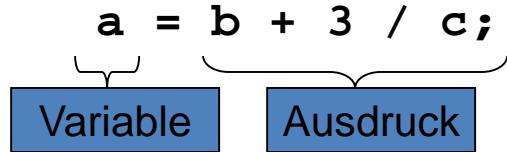
A || B : liefert A den Wert **true**,
so wird **B nicht mehr ausgewertet**

^ exklusives ODER		
	true	false
true	false	true
false	true	false

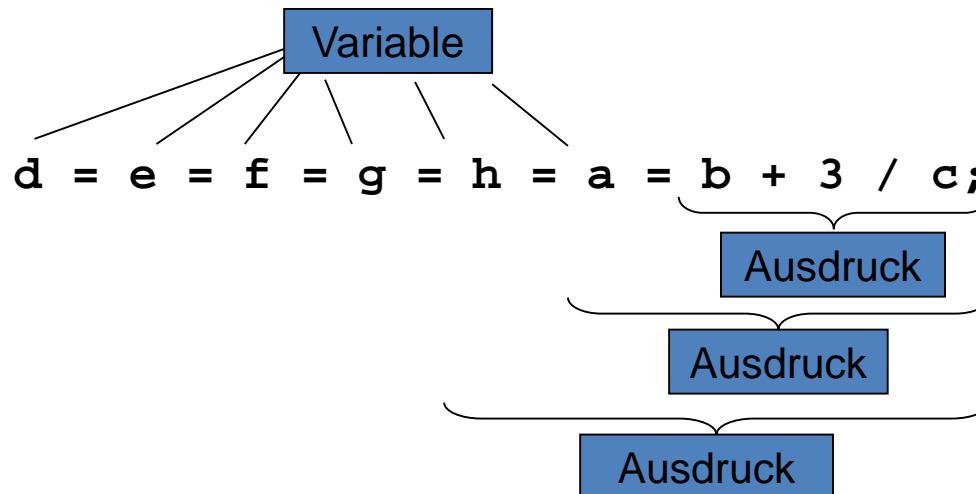
 logisches ODER		
	true	false
true	true	true
false	true	false

! logische NEGATION	
true	false
false	true

- Die Wertzuweisung als Ausdruck
 - Normalerweise ist die Wertzuweisung eine Anweisung



- In Java: Die Zuweisung ist selbst wieder ein Ausdruck. Der Wert eines Zuweisungsausdrucks ist der neue Wert der jeweils linken Seite
- Daher ist folgende **Mehrfachzuweisung** erlaubt:



- Regeln für die Auswertungsreihenfolge in Ausdrücken
 - Operatoren mit höherer Priorität vor Operatoren mit niedrigerer Priorität
 - mehrere zweistellige Operatoren mit gleicher Priorität von links nach rechts (links-assoziativ)
 - mehrere einstellige Operatoren und Zuweisungsoperatoren von rechts nach links (rechts-assoziativ)
 - elementare Ausdrücke und Ausdrücke in Klammern vor den Operatoren, die diese weiterverknüpfen
 - linker Operanden-Ausdruck immer vor dem rechten Operanden-Ausdruck
 - Beispiel: Wie wird der folgende Ausdruck ausgewertet?
 - `Math.sqrt(3.5+x)*5/3-(x+10)*(x-4.1) < 0`

2

1

3

4

8

5

7

6

9

Operator-Prioritäts-Stufen

Java

C/C++

Postfixoperatoren

[] . () ++ --

einst. Operatoren

++ -- + - ~ !

Erzeugung/Cast

new (Typname)

multiplikativ

* / %

additiv

+ -

shift

<< >> >>>

relational

< > <= >= instanceof

Gleichheit

== !=

logisches/bitweises UND

&

logisches/bitweises XOR

^

logisches/bitweises ODER

|

logisches UND

&&

logisches ODER

||

Bedingung

? :

Zuweisung

= += -= *= /= %= &= ^= |= <=>= >>>=

höhere

niedrigere

Merkregel: einstellig vor multiplikativ vor additiv vor vergleichend vor logisch vor zuweisend

Syntaxregel für `final`-Variablen / symbolische Konstanten

```
final <VARIABLENTYP> <VARIABLENBEZEICHNER> = <AUSDRUCK>;
```

Beispiel:

```
final double WECHSELKURS = 0.723;
```

Unzulässig wäre dann: `WECHSELKURS = 1.47;`

Syntaxregel für const-Variablen

```
const <VARIABLENTYP> <VARIABLENBEZEICHNER> = <AUSDRUCK>;  
oder  
<VARIABLENTYP> const <VARIABLENBEZEICHNER> = <AUSDRUCK>;
```

Beispiel:

```
const double WECHSELKURS = 0.723;
```

Unzulässig wäre dann: WECHSELKURS = 1.47;

- Bisher bekannte Arten von Anweisungen
 - Deklaration z.B. `int a;`
 - Wertzuweisung z.B. `a = 5;`
 - Methodenaufruf z.B. `System.out.println(a);`
 - Leere Anweisung z.B. `;`
- Hierdurch können wir nur rein lineare Programmstrukturen erzeugen
- Dieses Vorlesungskapitel beschäftigt sich mit weiteren Strukturierungsmöglichkeiten

- Ein **Block** ist eine durch Klammern { und } zusammengefasste Folge von Anweisungen

{

Anweisung₁

Anweisung₂

...

Anweisung_k

}

- Blöcke können geschachtelt sein, d.h. jede Anweisung kann selbst wieder durch einen Block ersetzt werden
- Sie dienen der Strukturierung von Programmen

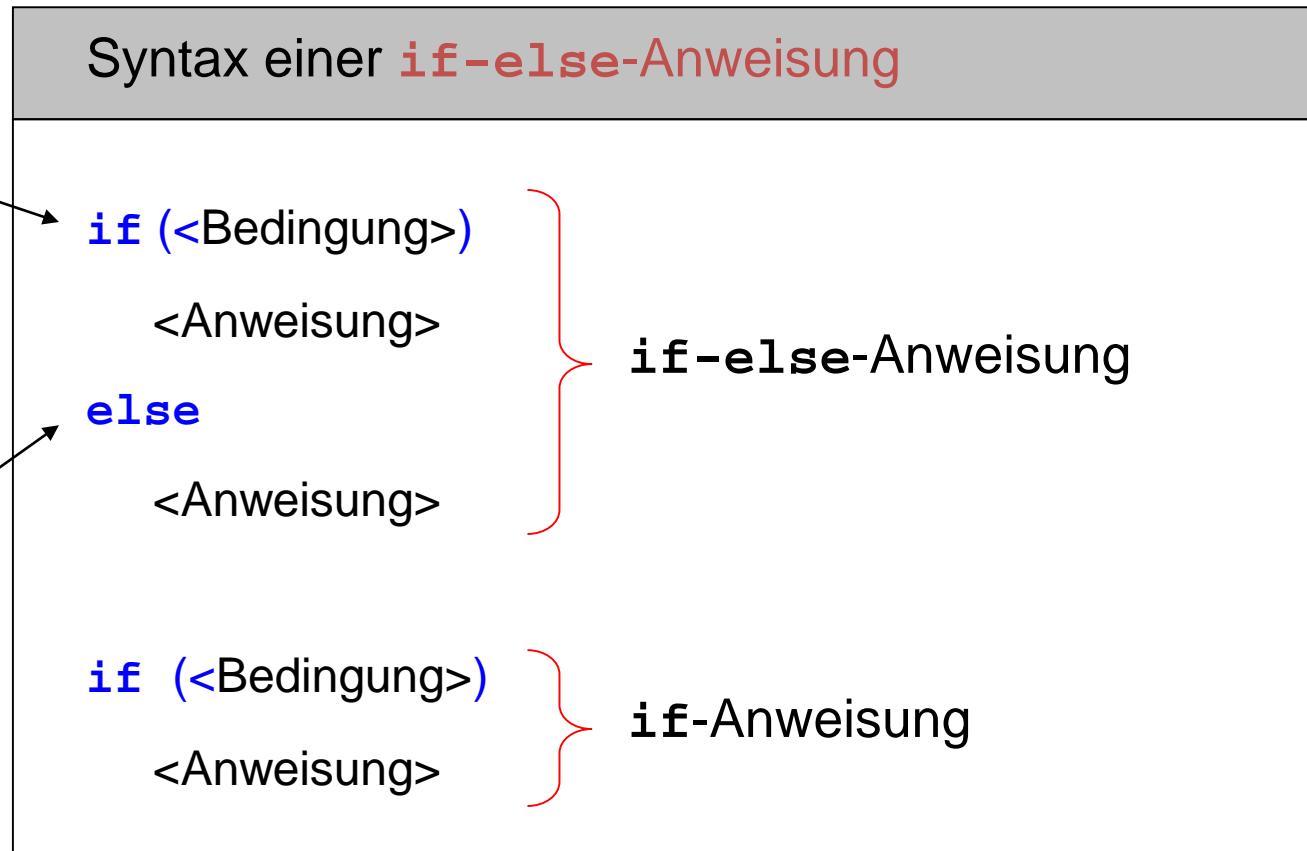
```
{ int i = 1;  
    {                                // Start des ersten Blocks  
        int j = 2;                  // j ist lokal im ersten Block  
        ...                          // i ist hier auch bekannt  
    }                                // Ende des ersten Blocks  
    {                                // Start des zweiten Blocks  
        long k;                    // k ist lokal im zweiten Block  
        ...                          // i ist bekannt, j unbekannt  
    }                                // Ende des zweiten Blocks  
    ...                            // i ist bekannt, j und k nicht  
}  
  
                                // i, j und k sind hier unbekannt
```

**Variablen sind nur bis zum Ende des Blocks,
in dem sie definiert wurden, gültig!**

- Die bisher (z.B. ohne Verzweigungen und Schleifen) erzeugbaren Programme haben eine sehr einfache Gestalt:

```
public class Bezeichner {  
    public static void main (String [] args) {  
        Anweisung1  
        Anweisung2  
        ...  
        Anweisungn-1  
        Anweisungn  
    }  
}
```

- Anweisungen können hier auch durch Blöcke oder geschachtelte Blöcke ersetzt werden

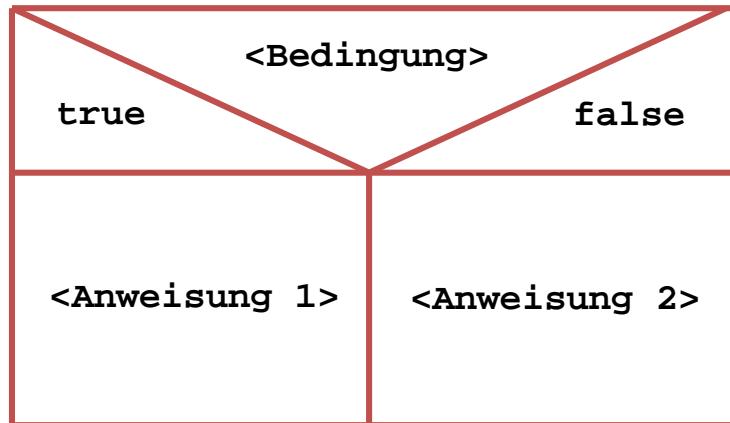


<Bedingung> = Boolescher Ausdruck

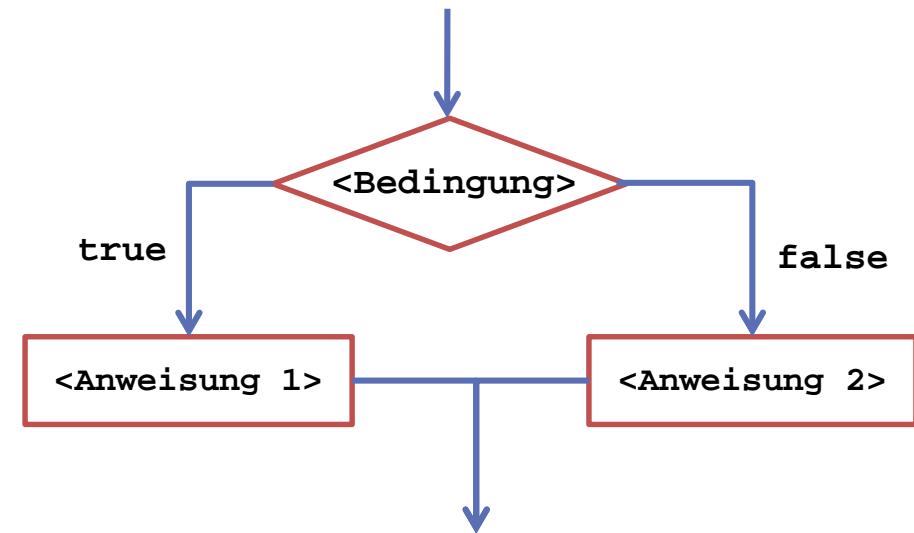
Die Anweisungen können durch Blöcke ersetzt werden

```
if (<Bedingung>) {  
    <Anweisung>  
    ...  
    <Anweisung>  
}  
  
else {  
    <Anweisung>  
    ...  
    <Anweisung>  
}
```

- Zunächst wird die Bedingung ausgewertet
- Ist sie **wahr (true)**, so führt man den **if**-Teil aus und überspringt den **else**-Teil
- Ist sie **falsch (false)**, so überspringt man den **if**-Teil und führt nur den **else**-Teil aus



Struktogramm



Programmablaufplan (PAP)

```
// (c)

double x = InputHelper.readDouble("x = ");
double y = InputHelper.readDouble("y = ");
if (x > 0)
    if (y > 0)
        System.out.println("beide positiv");
    else
        System.out.println("nur x positiv");
else
    if (y > 0)
        System.out.println("nur y positiv");
    else
        System.out.println("beide nicht positiv");
```

Regel: Ein **else** wird dem nächstliegenden vorangehenden **if** zugeordnet, dem noch kein **else** zugeordnet ist und das in keinem separaten Block liegt.
Besser ist, zu klammern!

Übung: Zu welchem **if** gehört das **else**?

```
int a;  
  
int b;  
  
...           // Anweisungen  
  
if (a > 0)  
  
    if (b > a)  
  
        System.out.println("b > 1");  
  
else  
  
    System.out.println( "Fehler: a ist <= Null" );
```

zum zweiten **if**

Klammern können Semantik
präzisieren

```
// (d) Mehrfachverzweigung

Scanner input = new Scanner(System.in);

System.out.println("Punktzahl: ");

int punkte = input.nextInt();

String note;

if (punkte > 50)

    note = "sehr gut";

else if (punkte > 41)

    note = "gut";

else if (punkte > 33)

    note = "befriedigend";

else if (punkte > 25)

    note = "ausreichend";

else

    note = "nicht bestanden";

System.out.println("Note: " + note);
```



```
// (d) Mehrfachverzweigung

Scanner input = new Scanner(System.in);

System.out.println("Punktzahl: ");

int punkte = input.nextInt();

String note;

if (punkte > 50)

    note = "sehr gut";

else if (punkte > 41)

    note = "gut";

else if (punkte > 33)

    note = "befriedigend";

else if (punkte > 25)

    note = "ausreichend";

else

    note = "nicht bestanden";

System.out.println("Note: " + note);
```



```
// (e) Nicht compilierbar!  
  
int u = 5, v = 2, w = 4;  
  
int z;  
  
if (u = v + w)  
    z = 27;  
else  
    z = z + 27;
```

Zuweisung anstelle von Vergleich!
Dadurch int-Wert in der Klammer.

```
// (f) Compilierbar, aber logisch problematisch!  
  
int z = 100;  
  
boolean bu = true, bv = false, bw = false;  
  
if (bu = bv && bw)  
    z = 42;  
else  
    z = z + 42;  
  
System.out.println("z = " + z);  
System.out.println("bu = " + bu);
```

Zuweisung anstelle von Vergleich!
Dadurch ist der Wert in der Klammer
durch den neuen Wert von bu gegeben.

Beispiel: Wo endet diese if-else-Anweisung?

Welches Problem taucht hier auf?

```
int x = (new Scanner(System.in)).nextInt();
if (x == 0) {
    System.out.println("x ist gleich 0");
}
else
    System.out.println("x ist ungleich 0");
    System.out.println("1/x liefert " + 1/x);
    System.out.println("Division durchgefuehrt");
```



Hier

Problem: Division durch 0. Es fehlen Klammern!

switch-Anweisung

Java

C/C++

Im Gegensatz zur `if`-Anweisung erlaubt die `switch`-Anweisung eine Auswahl unter mehreren Alternativen

Syntax switch-Anweisung

```
switch (<switch-Ausdruck>) {  
    case <Konstante>: <Anweisungsfolge>; break;  
    case <Konstante>: <Anweisungsfolge>; break;  
    ...  
    case <Konstante> : <Anweisungsfolge>; break;  
    default : <Anweisungsfolge>  
}
```

Semantik: Berechne Wert des `switch`-Ausdrucks und prüfe der Reihe nach die Konstanten bis eine Übereinstimmung gefunden wird. Stimmt der Wert erstmals mit der i-ten Konstante überein, so gelangt i-te Anweisungsfolge zur Ausführung.

Beispiel: **switch**-Anweisung

```
System.out.println("Zeichen eingeben: ");

char t = (new Scanner(System.in)).nextChar();

int s = 7;

switch (t) {

    case 'X':
        s = 25;
        break;

    case 'Y':
        s = 40;
        break;

}

System.out.println("s = " + s);
```

Eingabe:	Ausgabe:
x	s = 25
y	s = 40
x	s = 7
r	s = 7

```
switch (<switch-Ausdruck>) {  
    case <Konstante>: <Anweisungsfolge> break;  
    case <Konstante> : <Anweisungsfolge> break;  
    ...  
    case <Konstante> : <Anweisungsfolge> break;  
    default : <Anweisungsfolge>  
}
```

Der **default**-Zweig ist optional. Es darf höchstens ein **default**-Zweig auftreten.

Die Anweisungsfolgen können auch Blöcke sein.

break ist optional.

- Befindet sich ein **break** hinter der betreffenden Anweisung, so wird die **switch**-Anweisung beendet, *andernfalls* werden auch die weiteren Anweisungsfolgen ausgeführt (*bis zum nächsten break*).
- Stimmt der Wert des **Switch**-Ausdrucks mit keinem der Werte hinter dem Schlüsselwort **case** überein, so wird die Anweisung hinter dem **default** ausgeführt.
- Danach wird die **switch**-Anweisung beendet.

Beachte:

Vergessene **break**-Anweisungen führen oft zu schwer auffindbaren Fehlern!

```
Scanner input = new Scanner(System.in);
System.out.print("a = ");
int a = input.nextInt();
int b;
switch (a) {
    case 1:
        b = 10;
        a = 100;
    case 2:
    case 3:
        b = 25;
        a = 1500;
        break;
    case 4:
        b = 40;
        a = 1000000;
    default:
        b = 0;
        a = 0;
}
System.out.println("a = " + a + "    b = " + b);
```

Eingabe:	Ausgabe:
a = 3	a = 1500 b = 25
a = 2	a = 1500 b = 25
a = 1	a = 1500 b = 25
a = 4	a = 0 b = 0
a = 7	a = 0 b = 0

- **Aufgabe:** Schreiben Sie ein Programm, welches durch eine `switch`-Anweisung aus der Nummer eines Monats die Länge des betreffenden Monats ermittelt. Lassen Sie dabei Schaltjahre unberücksichtigt.

```
System.out.println("Monat = ");
int monat = (new Scanner(System.in)).nextInt();
int tage = 0;
switch ( monat ) {
    case 4:
    case 6:
    case 9:
    case 11: tage=30; break;
    case 2: tage=28; break;
    default: tage=31; break;
}
System.out.println("Tage = " + tage);
```

Worin unterscheiden sich die beiden Anweisungen?

```
System.out.println("Punkte = ");
int punkte = (new Scanner(System.in)).nextInt();
char note;
if      (punkte >= 80)                      note = '1';
else if (punkte >= 70)                      note = '2';
else if (punkte >= 60)                      note = '3';
else if (punkte >= 50)                      note = '4';
else                                         note = '5';

switch ((punkte / 10)) {
    case 10:
    case 9:
    case 8: note = '1'; break;
    case 7: note = '2'; break;
    case 6: note = '3'; break;
    case 5: note = '4'; break;
    default: note = '5'; break;
}
```

switch-Anweisung

- Ist der Wert des **switch**-Ausdrucks während des Ablaufs einer **switch**-Anweisung unveränderlich?

```
int x = 1;  
  
switch(x) {  
  
    case 1: x = 3;  
  
    case 4: x = 5; break;  
  
    case 3: x = 22;  
  
}  
  
System.out.println(x); // Welchen Wert hat x?
```

Syntax while-Anweisung

```
while (<Bedingung>) {  
    <Anweisungsfolge>  
}  
  
while (<Bedingung>)  
    <Anweisung>
```

Schleifenkörper

Die <Bedingung> ist ein Ausdruck vom Typ boolean

Semantik:

- prüfe die Bedingung
- ist diese wahr, so wird der Körper ausgeführt
- ist die Bedingung falsch, so wird die Schleife verlassen, d.h. die Anweisungen nach dem Schleifenkörper werden bearbeitet

Die **while**-Anweisung bezeichnet man auch als **abweisende Schleife**

Synonyme: **while**-Anweisung / **while**-Schleife / **Schleife mit Eingangsprüfung**

```
// (a) Berechnung der Quersumme
//      (z.B.: Quersumme von 1234 ist 10)
Scanner input = new Scanner(System.in);
System.out.print("Positive ganze Zahl: ");

long a = input.nextLong();

System.out.print("Die Quersumme von " + a + " ist ");
    0
long qs = ;
    a > 0
while (      ) {
    qs = qs + a % 10;
    a = a / 10;

}
System.out.println(qs);
```

%10 liefert die letzte Dezimalstelle
/10 hängt die letzte Dezimalstelle ab

```
// (b) Berechnung der Fakultät:  
//      k! = k * (k-1) * (k-2) * ... * 3 * 2 * 1  
Scanner input = new Scanner(System.in);  
System.out.print("Positive ganze Zahl: ");  
long k = input.nextLong();  
System.out.print(k + " ! = ");  
long f = 1;  
while (k > 1) {  
    f = f * k;  
    k = k - 1;  
}  
System.out.println(f);
```

Beispiele: `while`-Schleife

(c) Größter Gemeinsamer Teiler (GGT):

Gesucht ist der größte gemeinsame Teiler zweier positiver ganzer Zahlen

Lösungsansatz: Euklidischer Algorithmus (Erinnerung an V01)

mathematische Beobachtung:

Seien $a > b > 0$ ganze Zahlen und $g = \text{GGT}(a,b)$,
dann ist g auch $\text{GGT}(a - b, b)$.

Iterationsschritt: ersetze a durch $a - b$

In irgendeinem Schritt sind die aktuell betrachteten Zahlen
gleich. Sie gleichen dann dem GGT der Ausgangszahlen.

Kern des Programms:

```
Scanner input = new Scanner(System.in);
System.out.print("Ganze Zahl a = ");
int a = input.nextInt();
System.out.print("Ganze Zahl b = ");
int b = input.nextInt();
System.out.print("Der GGT von " + a + " und " + b + " ist " );
```

```
while(a != b) {
    if (a > b)
        a -= b;
    else
        b -= a;
}
```

```
System.out.println(a);
```

Vorsicht Falle!

```
// (d)
    double x = 15;          // Endlosschleife!
    while (x > 10);        // Rumpf der while-Schleife besteht
    x = x - 10;             // nur aus leerer Anweisung (wg. ;)

// (e)
    while (x > 0)
    x = Math.sqrt(x);      // Nur die erste Anweisung gehört
    x = x - 10;             // zum Rumpf der while-Schleife

// (f)
    while (true)           // Endlosschleife
    x = 5;

// (g)
    while (false)          // Compilierungs-Fehler,
    x = 5;                  // da Anweisung unerreichbar
```

Wie stoppt man eine Endlosschleife?

```
while (true) {  
  
    // Anweisungen  
  
    // hier sollte ein break  
  
    // oder ein return stehen,  
  
    // return wird später behandelt  
}
```

Beispiel:

```
int x = 1;
while (true) {
    System.out.println(x);
    if (x == 10) break;
    // unterbricht die aktuelle Schleife
    x = x + 1;
}
```

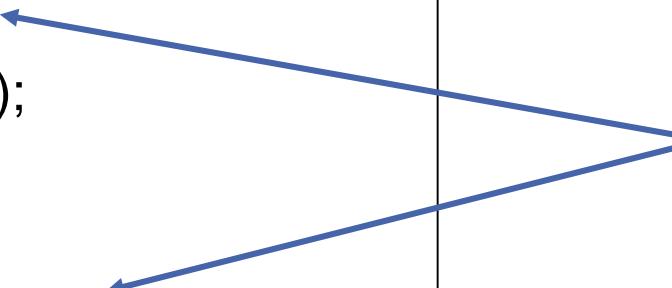
do-Anweisung - Schleife mit Ausgangsprüfung

Syntax: do-Anweisung

```
do
    <Anweisung>
  while (<Bedingung>);

do {
    <Anweisungsfolge>
} while (<Bedingung>);
```

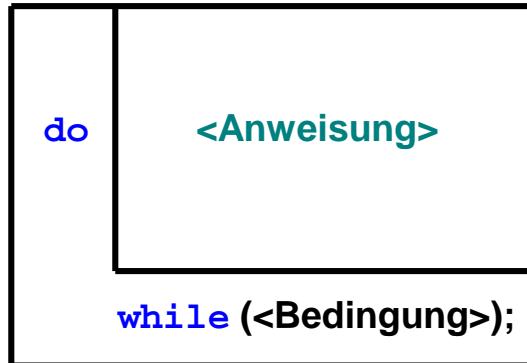
Die <Bedingung> ist ein Ausdruck
vom Typ **boolean**



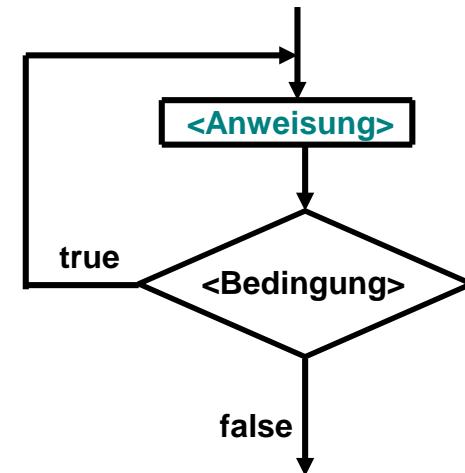
Anweisungen des
Schleifenkörpers

Semantik:

Nach der Initialisierung startet die do-Schleife mit der Abarbeitung des Schleifenkörpers. Erst danach wird die Bedingung geprüft. Bei positivem Ausgang des Tests wird wieder der Schleifenkörper durchlaufen usw. Bei negativem Ausgang wird die Schleife beendet.



Struktogramm



Programmablaufplan (PAP)

Beispiel:

```
int i = 1;  
do {      // Schleifenkörper wird mindestens einmal ausgeführt,  
          // auch dann, wenn die Bedingung nicht erfüllt ist  
    System.out.println(i);  
    i = i + 1;  
} while (i < 6);
```

Syntax:

```
for ( Initialisierungsliste  
      (kann entfallen) ; Logischer Ausdruck  
      (kann entfallen) ; Update-Liste  
      (kann entfallen) )
```

Komma-Liste von Ausdrücken bzw.
lokalen Variablen Deklarationen mit
Initialisierung

Anweisung bzw.
Block mit Anweisungsfolge

Komma-Liste von
Ausdrücken

Wirkung:

1. Die Initialisierungsliste wird ausgeführt.
2. Der logische Ausdruck wird ausgewertet.
3. Ist der Wert `true`, dann wird die Anweisung bzw. der Block (Anweisungsfolge) ausgeführt, danach die Update-Liste ausgeführt und wieder zu Schritt 2 gesprungen.
4. Ist der Wert jedoch `false`, so ist die `for`-Anweisung beendet.

Weitere
Nebenbedingungen
folgen am Beispiel

Semantik der **for**-Schleife:

Die Schleife ist äquivalent zu folgender **while**-Schleife:

```
{<Initialisierung>;
    while (<Bedingung>) {
        <Anweisung bzw. Block>          // des Schleifenkörpers
        <Aktualisierung>;
    }
}
```

Übung: Zeigen Sie, dass sich **alle** Arten von Schleifen
äquivalent durcheinander ersetzen lassen.

```
// (c) Einmaleins  
  
long j, k, jMax, kMax;  
Scanner input = new Scanner(System.in);  
System.out.println("Obere Schranke fuer j: ");  
jMax = input.nextLong();  
System.out.println("Obere Schranke fuer k: ");  
kMax = input.nextLong();  
System.out.println("Die Produkte des Einmaleins");
```

Ausgaben der Form:

```
Obere Schranke fuer j: 5  
Obere Schranke fuer k: 3  
Die Produkte des Einmaleins  
1 * 1 = 1  
1 * 2 = 2  
1 * 3 = 3  
2 * 1 = 2  
2 * 2 = 4  
2 * 3 = 6  
3 * 1 = 3  
3 * 2 = 6  
...
```

Ausgaben:
20
17
14
11
8
5
2

```
// (c) Einmaleins
long j, k, jMax, kMax;
Scanner input = new Scanner(System.in);
System.out.println("Obere Schranke fuer j: ");
jMax = input.nextLong();
System.out.println("Obere Schranke fuer k: ");
kMax = input.nextLong();
System.out.println("Die Produkte des Einmaleins");
for (j = 1; j <= jMax; j++)
    for (k = 1; k <= kMax; k++)
        System.out.println(j + " * " + k + " = " + j*k);
```

Ausgaben der Form:

Obere Schranke fuer j: 5
Obere Schranke fuer k: 3
Die Produkte des Einmaleins
1 * 1 = 1
1 * 2 = 2
1 * 3 = 3
2 * 1 = 2
2 * 2 = 4
2 * 3 = 6
3 * 1 = 3
3 * 2 = 6
...

Vorsicht Falle!

```
// (d)
for (;;) {                                // Endlosschleife!
    System.out.println("Netzstecker ziehen!");
}

// (e)
for (int a = 0; a < 5; a++) {
    System.out.println(a + " " + a*a*a);
}
System.out.println(a); // unzulässig!
                                // Variable a ist nur in
                                // der Schleife bekannt
```

break-Anweisung

Syntax: break-Anweisung

```
break <Bezeichner>opt;
```

- Eine **break**-Anweisung sollte nur in **switch**-, **while**-, **do** oder **for**-Anweisungen stehen.
- Der **<Bezeichner>** muss Marke einer markierten **switch**-, **while**-, **do**- oder **for**-Anweisung sein, welche das **break** enthält

Semantik (ohne Marke): Die **break**-Anweisung beendet die Ausführung der **innersten**, das **break** umschließenden Schleife bzw. **switch**-Anweisung

Beispiele:

```
while ( Bedingung ) {  
  
    // Anweisungen  
  
    if (...) {  
        break;  
    }  
  
    // Anweisungen  
  
}  
// 1. Anweisung nach der Schleife
```



search:

```
for (i = 0; i < arrayOfInts.length; i++) {  
    for (j = 0; j < arrayOfInts[i].length; j++) {  
        if (arrayOfInts[i][j] == searchfor) {  
            foundIt = true;  
            break search; _____  
        }  
    }  
}  
// 1. Anweisung nach der Schleife _____ ←
```

continue-Anweisung

Syntax: **continue**-Anweisung

```
continue <Bezeichner>opt ;
```

- darf nur in einer **while**-, **do**- oder **for**-Anweisung vorkommen
- der <Bezeichner> muss Marke einer markierten Wiederholungsanweisung (**while**-, **do**- oder **for**-Anweisung) sein, welche das **continue** enthält

Semantik (hier nur ohne Marke):

Bewirkt einen Sprung an das Ende der innersten, das `continue` umschließenden Schleife und beginnt gegebenenfalls eine erneute Iteration.

Bei einer `while`- oder `do`-Schleife wird also die Abbruchbedingung (boolescher Ausdruck) ausgewertet, bei einer `for`-Schleife werden vorher noch die Aktualisierungs-Ausdrücke ausgewertet.

Je nach Ausgang wird der Schleifenkörper erneut ausgewertet oder die Schleife verlassen.

Beispiel:



```
while ( <Bedingung> ) {  
    // <Anweisungsfolge>  
    if ( <Bedingung> ) {  
        continue; _____  
    }  
    // <Anweisungsfolge>  
}  
// erste <Anweisung> nach Schleife
```

```
while (...) {  
    ...  
    if (...)  
        break;  -----|  
    }  
-----|
```

```
while (...) { -----|  
    if (...)  
        continue;  -----|  
    ...  
}
```

```
for (i = 0; i < 100; i++) {  
    if (...)  
        break;  -----|  
    ...  
}
```

```
for (i = 0; i < 10; i++) {  
    if (...)  
        continue;  -----|  
    ...  
}
```

- Die Anweisung **break** beendet die Ausführung der innersten (das **break** umschließenden) Schleife.
- Die Anweisung **continue** beendet den aktuellen Durchlauf der innersten (das **continue** umfassenden) Schleife.

```
for (int i = 0; i < 50; i++) {  
    if (i == 26)  
        break;  
    if (i % 9 != 0)//i ist nicht durch 9 teilbar  
        continue;  
    System.out.println(i);  
}  
  
int i = 0;  
while (true) {  
    i++;  
    int j = i * 30;  
    if (j == 990)  
        break;  
    if (i % 10 != 0)//i ist nicht durch 10 teilbar  
        continue;  
    System.out.println(i);  
}
```

Ausgaben:

0
9
18
10
20
30

- Mit den bisher bekannten Mitteln ist es schwer große Datensammlungen anzulegen
- Beispiele hierfür sind:
 - Personaldateien
 - Stücklisten
 - Wetterdaten
 - Terminkalender

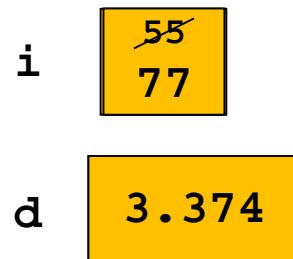
Bisher: elementare Datentypen, statisch im Speicher

im Programm

symbolisch notiert

im Speicher

```
int i;  
double d;  
i = 55;  
d = 3.374;  
i = 77;
```



Adresse

i: 133
d: 214

Inhalt

≡
55 77
≡
3.374
≡

Jetzt: eigene Datentypen, dynamisch im Speicher → Referenzdatentypen

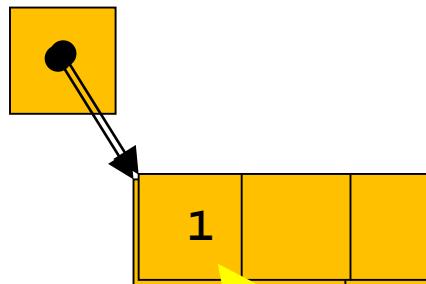
im Programm

symbolisch notiert

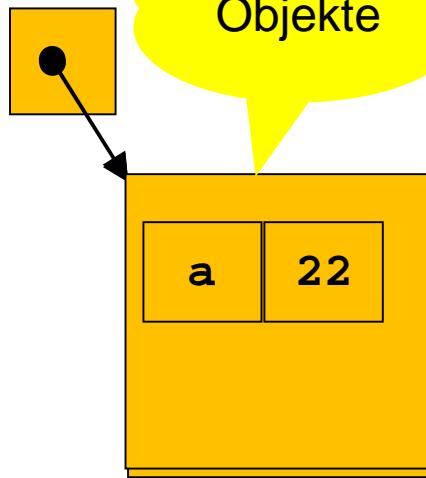
im Speicher

```
int[] iv;  
  
MeinTyp mv;  
  
iv = new int[3];  
  
mv = new MeinTyp();  
  
iv[0] = 1;  
  
mv.a = 22;
```

iv



mv



Adresse

iv: 1227

8433

9177

mv: 2735



- Felder sind Objekte mit mehreren **Komponenten gleichen Typs**
- Die Komponenten eines Feldes haben keinen Namen, man greift über den Feldnamen und in **[]** geklammerte **Indizes** auf sie zu
- **Beispiel:** Berechnung der Varianz von **n** einzulesenden Werten x_1, x_2, \dots, x_n (z.B. für **n = 100**):

Mittelwert $\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i = \frac{1}{n} \cdot (x_1 + x_2 + \dots + x_n)$

Varianz $v = \frac{1}{n} \sum_{i=1}^n (x_i - \bar{x})^2 = \frac{1}{n} \cdot ((x_1 - \bar{x})^2 + (x_2 - \bar{x})^2 + \dots + (x_n - \bar{x})^2)$

Beispielprogramm für $n = 16$ ohne Feld - kurze Variante mit Feld

```

import java.util.Scanner;
public class Varianz1 {
    public static void main(String[] args){
        int n = 16;
        double x1, x2, x3, x4, x5, x6, x7, x8,
               x9, x10, x11, x12, x13, x14, x15, x16;
        double mw, va;
        Scanner input = new Scanner(System.in);
        System.out.println(n + " Werte eingeben:");
        x1 = input.nextDouble();
        x2 = input.nextDouble();
        x3 = input.nextDouble();
        x4 = input.nextDouble();
        x5 = input.nextDouble();
        x6 = input.nextDouble();
        x7 = input.nextDouble();
        x8 = input.nextDouble();
        x9 = input.nextDouble();
        x10= input.nextDouble();
        x11= input.nextDouble();
        x12= input.nextDouble();
        x13= input.nextDouble();
        x14= input.nextDouble();
        x15= input.nextDouble();
        x16= input.nextDouble();
        mw = (x1 + x2 + x3 + x4 +
              x5 + x6 + x7 + x8 +
              x9 + x10 + x11 + x12 +
              x13 + x14 + x15 + x16) / n;
        va = Math.pow(x1-mw,2) + Math.pow(x2-mw,2) +
             Math.pow(x3-mw,2) + Math.pow(x4-mw,2) +
             Math.pow(x5-mw,2) + Math.pow(x6-mw,2) +
             Math.pow(x7-mw,2) + Math.pow(x8-mw,2) +
             Math.pow(x9-mw,2) + Math.pow(x10-mw,2) +
             Math.pow(x11-mw,2) + Math.pow(x12-mw,2) +
             Math.pow(x13-mw,2) + Math.pow(x14-mw,2) +
             Math.pow(x15-mw,2) + Math.pow(x16-mw,2);
        va = va / n;
        System.out.println("Varianz: " + va);
    }
}

```

```

import java.util.Scanner;
public class Varianz2 {
    public static void main(String[] args){
        Scanner input = new Scanner(System.in);
        System.out.print("Wieviele Werte? n = ");
        int n = input.nextInt();
        double[] x = new double[n];

        double mw, va;
        for (int i=0; i < n; i++) {
            System.out.print("x[" + i + "] = ");
            x[i] = input.nextDouble();
        }

        mw = 0;
        for (int i=0; i < n; i++)
            mw = mw + x[i];
        mw = mw / n;
        va = 0;
        for (int i=0; i < n; i++)
            va = va + Math.pow(x[i]-mw,2);

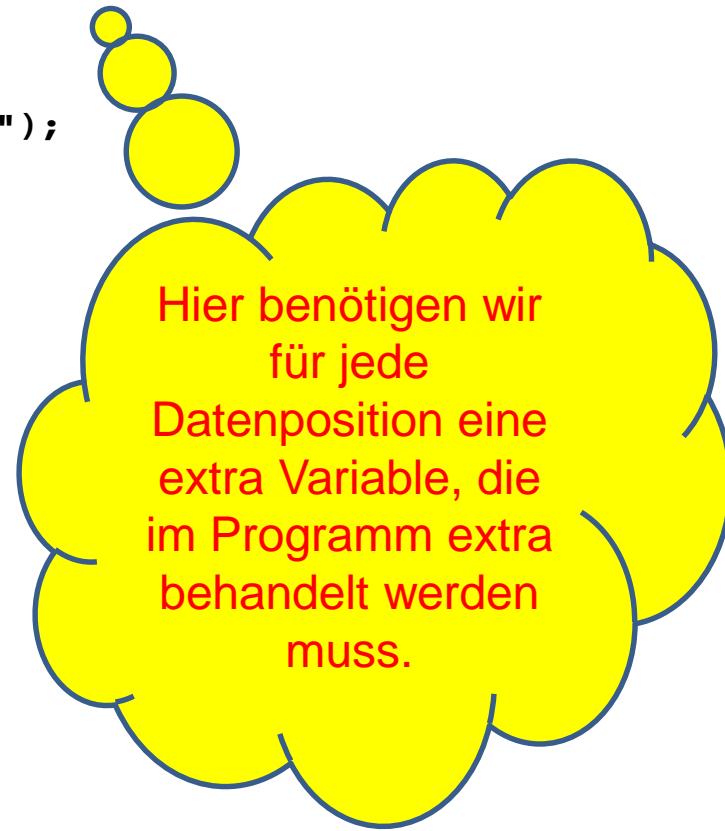
        va = va / n;
        System.out.println("Varianz: " + va);
    }
}

```

Java



```
import java.util.Scanner;
public class Varianz1 {
    public static void main(String[] args){
        int n = 16;
        double x1, x2, x3, x4, x5, x6, x7, x8,
               x9, x10, x11, x12, x13, x14, x15, x16;
        double mw, va;
        Scanner input = new Scanner(System.in);
        System.out.println (n + " Werte eingeben:");
        x1 = input.nextDouble();
        x2 = input.nextDouble();
        x3 = input.nextDouble();
        x4 = input.nextDouble();
        x5 = input.nextDouble();
        x6 = input.nextDouble();
        x7 = input.nextDouble();
        x8 = input.nextDouble();
        x9 = input.nextDouble();
        x10= input.nextDouble();
        x11= input.nextDouble();
        x12= input.nextDouble();
        x13= input.nextDouble();
```



```
x14= input.nextDouble();
x15= input.nextDouble();
x16= input.nextDouble();

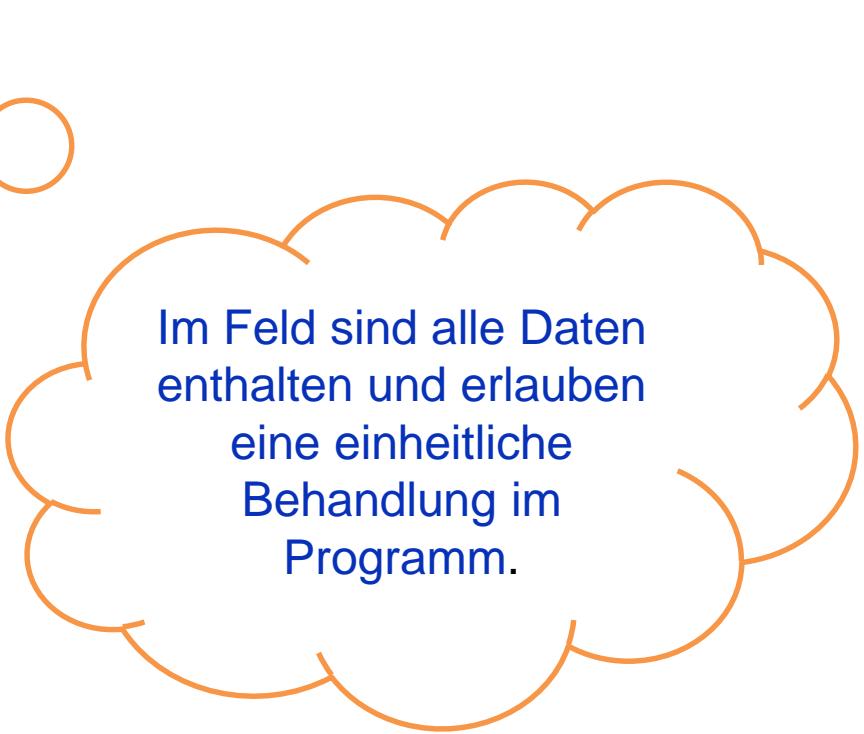
mw = (x1 + x2 + x3 + x4 +
      x5 + x6 + x7 + x8 +
      x9 + x10 + x11 + x12 +
      x13 + x14 + x15 + x16) / n;

va = Math.pow(x1-mw,2) + Math.pow(x2-mw,2) +
     Math.pow(x3-mw,2) + Math.pow(x4-mw,2) +
     Math.pow(x5-mw,2) + Math.pow(x6-mw,2) +
     Math.pow(x7-mw,2) + Math.pow(x8-mw,2) +
     Math.pow(x9-mw,2) + Math.pow(x10-mw,2) +
     Math.pow(x11-mw,2) + Math.pow(x12-mw,2) +
     Math.pow(x13-mw,2) + Math.pow(x14-mw,2) +
     Math.pow(x15-mw,2) + Math.pow(x16-mw,2);

va = va / n;
System.out.println("Varianz: " + va);
}
```



```
import java.util.Scanner;  
  
public class Varianz2 {  
  
    public static void main(String[] args){  
  
        Scanner input = new Scanner(System.in);  
  
        System.out.println("Wieviele Werte? n = ");  
  
        int n = input.nextInt();  
  
        double[] x = new double[n];   
  
        double mw, va;   
  
        for (int i=0; i < n; i++)   
            x[i] = input.nextDouble();   
  
        mw = 0;   
  
        for (int i=0; i < n; i++)   
            mw = mw + x[i];   
  
        mw = mw / n;   
  
        va = 0;   
  
        for (int i=0; i < n; i++)   
            va = va + Math.pow(x[i]-mw,2);   
  
        va = va / n;   
  
        System.out.println("Varianz: " + va);   
    }   
}
```



Im Feld sind alle Daten enthalten und erlauben eine einheitliche Behandlung im Programm.

- **Def: Feld (Array)**
 - beschränkte Sammlung von Elementen **identischen Typs**
 - Feldelemente sind indiziert (beginnend mit **0** bis zu einer oberen **Grenze**)
- Typ und Grenze sind in der Deklaration bzw. beim Einrichten des Feldes zu spezifizieren

- Eine Feldvariable (Variable eines Feldtyps) wird deklariert durch Angabe des Komponententyps gefolgt von leeren eckigen Klammern und dem Namen (Bezeichner) der Variable

Beispiele:

```
double[] x;
```

```
int[] a;
```

```
boolean[] z;
```

null-Referenz

x null

Referenzvariable

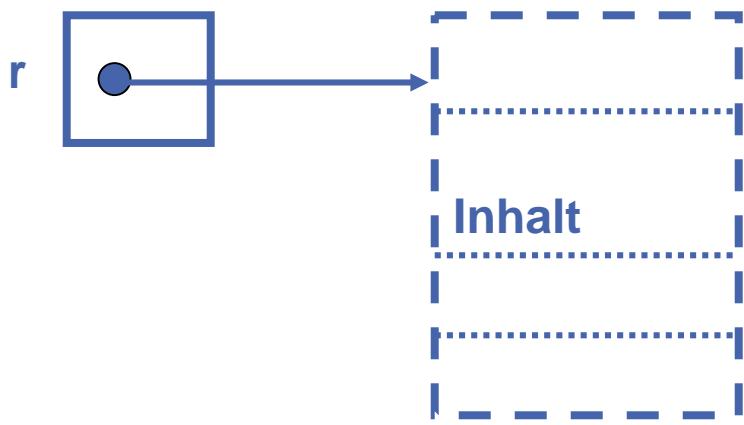
a null

z null

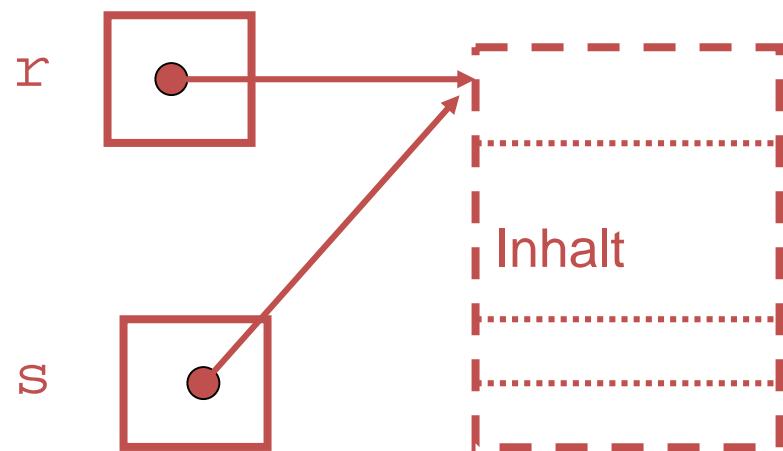
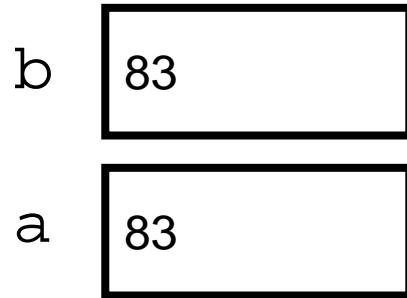
Graphische Notation für Variablen **einfacher Typen**:



Graphische Notation für Variablen von **Referenztypen**:



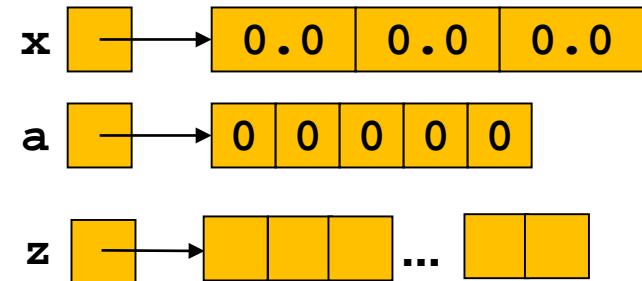
- Zwei Variablen einfachen Typs sind zu einem Zeitpunkt t gleich, wenn sich ihr Inhalt zum Zeitpunkt t gleicht ($b == a$).



Zwei Referenzvariablen sind gleich, wenn die Referenzen (d.h. bei der visuellen Darstellung die Pfeile) die gleichen Ziele haben ($r == s$)

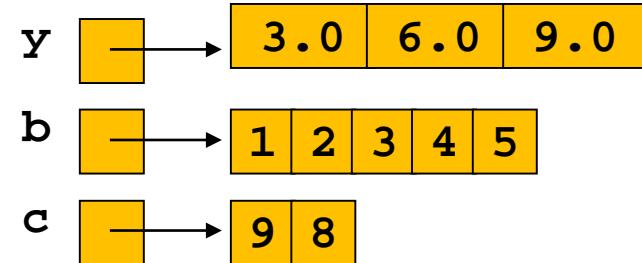
- **Dynamisch:** Ein Feld wird erzeugt durch den **new-Operator** gefolgt vom Komponententyp und in eckigen Klammern eingeschlossenen Längenangaben

```
x = new double[3];
a = new int[5];
System.out.print("n = ");
int n = input.nextInt() // java.util.Scanner
z = new boolean[n];
```



- **Statisch:** Ein Feld wird **in Verbindung mit seiner Deklaration** erzeugt durch Aufzählung seiner Komponenten in geschweiften Klammern

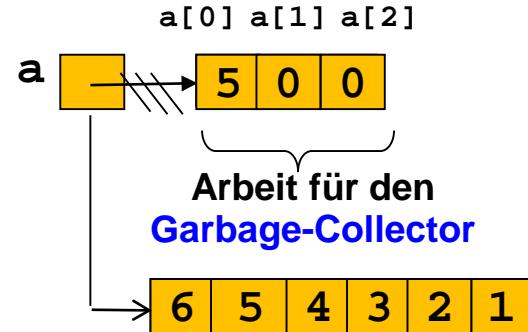
```
double[] y = {3.0,6.0,9.0};
int[] b = {1,2,3,4,5};
int[] c = new int[] {9,8};
```



- Ein Feld der Länge n hat n Komponenten, die erste hat den Index 0 und die letzte den Index $n-1$
- Der Zugriff auf die Komponenten erfolgt über mit [und] geklammerten Index-Ausdrücke

Beispiele:

```
a = new int[3];
a[0] = 5;
a = new int[6];
for (int i=0; i<6; i++)
    a[6-i-1] = i+1;
```



- Ein Zugriff außerhalb der Indexgrenzen verursacht einen Laufzeitfehler: **ArrayIndexOutOfBoundsException**
z.B. wäre $a[-1]$ oder $a[100]$ unzulässig

- Zusätzlich zu den Feldkomponenten besitzt ein Feldobjekt eine spezielle Komponente (Instanz-Variable), die jeweils die aktuelle Feldlänge enthält. Sie heißt **length** und wird mit einem **.** angesprochen

- Beispiel:

```
// variable „input“ ist vom Typ java.util.Scanner
System.out.print("Länge von x:");
int n = input.nextInt();
double[] x = new double[n];
System.out.print("Länge von y:");
n = input.nextInt();
double[] y = new double[n];

// Jetzt die Komponenten einlesen:
for (int i=0; i<x.length; i++) {
    System.out.print("x[" + i + "] = ");
    x[i] = input.nextDouble();
}
for (int i=0; i<y.length; i++) {
    System.out.print("y[" + i + "] = ");
    y[i] = input.nextDouble();
}
```

Deklaration eines Array (Syntax)

```
<Typ>[ ] <Feldname> = new <Typ>[<Grenze>];  
  
<Typ>[ ] <Feldname>;                      // eigentliche Deklaration  
<Feldname> = new <Typ>[<Grenze>]; // Einrichten des Feldes
```

Grenze bezeichnet man auch als Feldlänge

<Typ> → Typ oder Klasse

Indizierung der Feldelemente läuft von 0 bis <Grenze>-1

Deklaration eines Array (Syntax)

```
<Typ>[ ] <Feldname> = new <Typ>[<Grenze>];
```

```
<Typ>[ ] <Feldname>;
```

```
<Feldname> = new <Typ>[<Grenze>]; // Einr
```

Dynamische Initialisierung

```
<Typ>[ ] <Feldname> = {<Werte>};
```

Statische Initialisierung
Feld-Initialisierer
(array initializer)

<Typ> → Typ oder Klasse

Indizierung der Feldelemente läuft von 0 bis <Grenze>-1

<Werte> ist eine Folge

w_0, w_1, \dots, w_{k-1} von Anfangswerten

Die Indexgrenzen erhält man aus deren Anzahl **k**

- Welche Fehler sollte man vermeiden?
- Typische Fehler sind:
 - fehlende Initialisierung
 - fehlerhafte Zuweisungen (Wert vs. Referenz)
 - Über- oder Unterschreitung der Feldgrenzen
 - Verwendung falscher Typen

```
int[] feld;  
  
//feld[0] = 1;  
  
//System.out.println(feld);  
  
feld = new int[10];  
feld[0] = 0;  
  
System.out.println(feld[0]);  
  
if (feld[0]==0)  
    System.out.println("feld[0] = " + feld[0]);  
  
System.out.println("feld = " + feld);
```

Variable feld might
not have been
initialized

Variable feld might
not have been
initialized

Adresse:
[I@1a1c887

Wo liegt hier das Problem?

```
int max = 4;  
  
int[] feld;  
  
feld = new int[max];  
  
for (int i=0; i <= max; i++){  
    feld[i] = i;  
}
```

Laufzeitfehler:
java.lang.ArrayIndexOut
OfBoundsException

< geht
besser ist
i < feld.length

- **Größe des Feldes**
 - durch die Speicherkapazität des Rechners und die Größe des Komponententyps beschränkt
 - fixiert, wenn das Feld angelegt wird, jedoch noch **nicht nach der Deklaration!**
 - kann nach dem **Einrichten des Feldes nicht verändert** werden
- bei der Erzeugung eines Feldes wird ein Verweis auf den Speicherbereich angelegt, in welchem die Feldelemente gespeichert werden
- Man beachte jedoch die Möglichkeit, einer **Variable** Felder **unterschiedlicher Länge** zuzuweisen

- Hier werden der Variable **a** Felder unterschiedlicher Länge zugewiesen!

```
int[] a = new int[3];
```

```
int[] b;
```

```
b = new int[5];
```

```
a = b;
```

Feldlänge 3

Feldlänge 5

Feldlänge 5

?

Der Zuweisungsoperator kann für ganze Felder benutzt werden.
Allerdings wird dabei nur die Referenz kopiert! Separate Kopien erzeugt man durch *cloning* (siehe später).

Was wird ausgegeben?

- Beim Vergleichen und Kopieren von Feldern muss Referenzstruktur beachtet werden!

```
int[] a = {1, 2, 3, 4};  
  
int[] b = {1, 2, 3, 4};  
  
int[] c = a;  
  
int[] d = new int[4];
```

```
for (int i=0; i<a.length-1; i++)  
    d[i] = a[i];
```

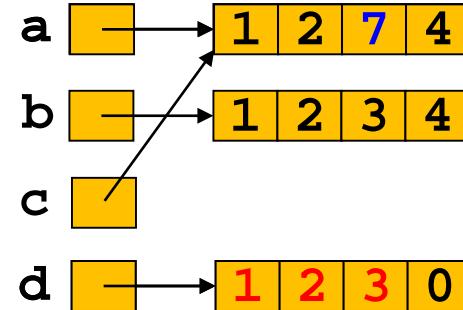
```
a[2] = 7;
```

```
System.out.println(a == b);
```

```
System.out.println(a == c);
```

```
System.out.println(a == d);
```

```
System.out.println(b[2] + " " + c[2] + " " + d[3]);
```



Ausgabe:

false

true

false

3 7 0

Semantik von **<Feld>[<Ausdruck1>] = <Ausdruck2>;**

<Ausdruck1> auswerten liefert **Wert1**

// beachte Inkrementen aus Ausdruck1

<Ausdruck2> auswerten liefert **Wert2**

Zuweisung **<Feld>[Wert1] = Wert2;** ausführen

// beachte Inkrementen aus Ausdruck2

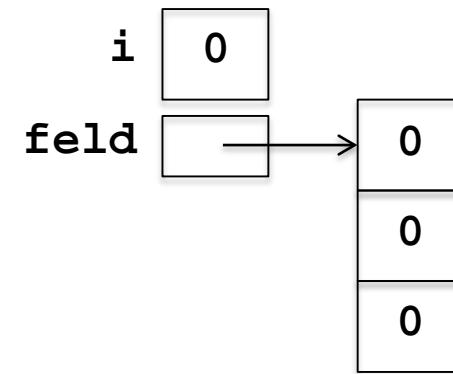
Was passiert hier?

```
int i = 0;  
  
int [] feld = {0,0,0};  
  
feld[++i + i++] = i++ + ++i;  
  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```

```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```

i 0

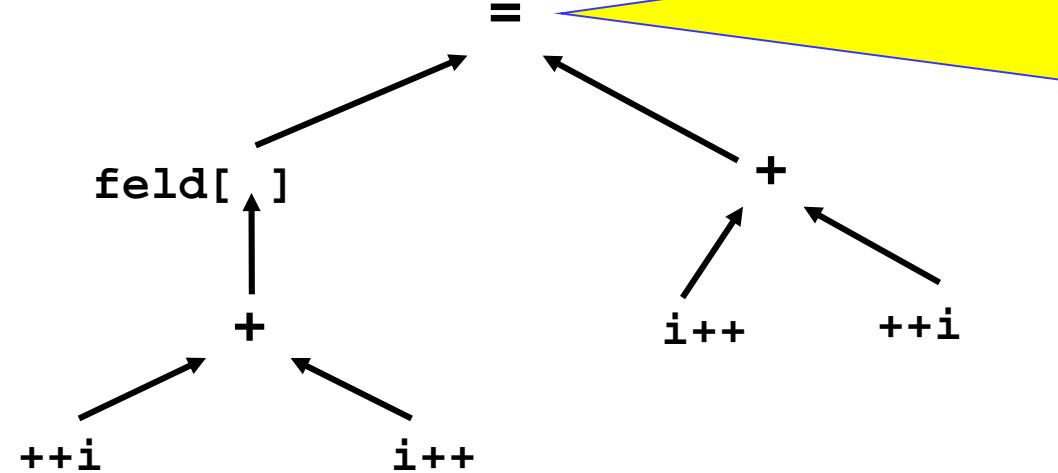
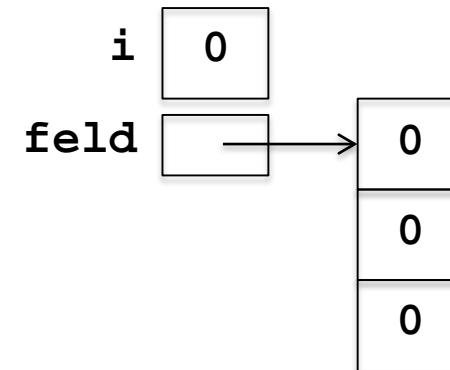
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



```

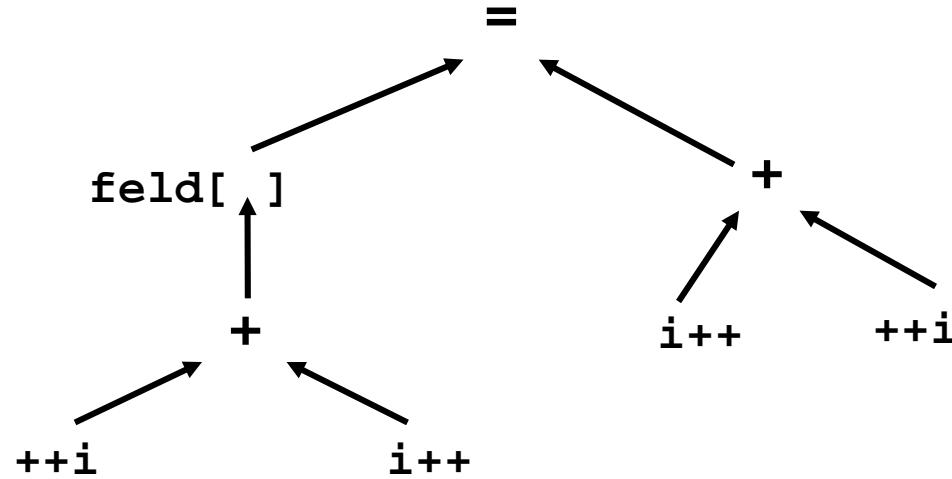
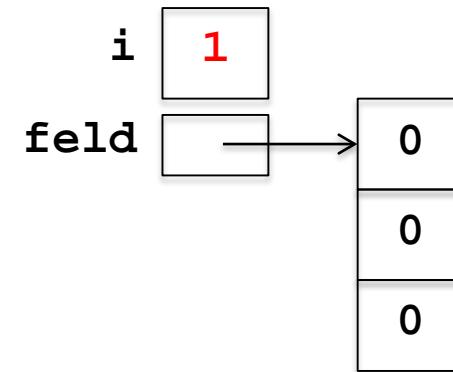
int i = 0;
int[] feld = {0,0,0};
feld[++i + i++] = i++ + ++i;
System.out.println(feld[0]);
System.out.println(feld[1]);
System.out.println(feld[2]);
System.out.println(i);

```

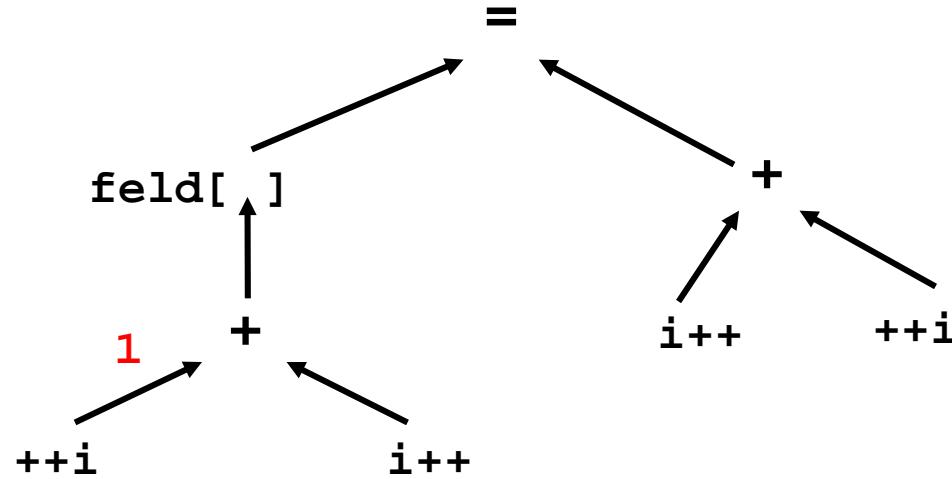
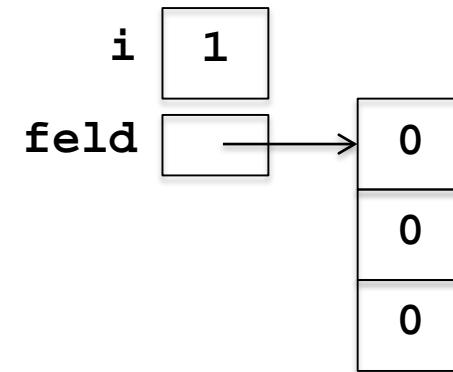


Hinweis: Der dem Ausdruck zugeordnete Baum wird in einem **Postorder-Durchlauf** (Linker Teilbaum, Rechter Teilbaum, Wurzel: Abkürzung **LRW**) durchlaufen. (siehe später)

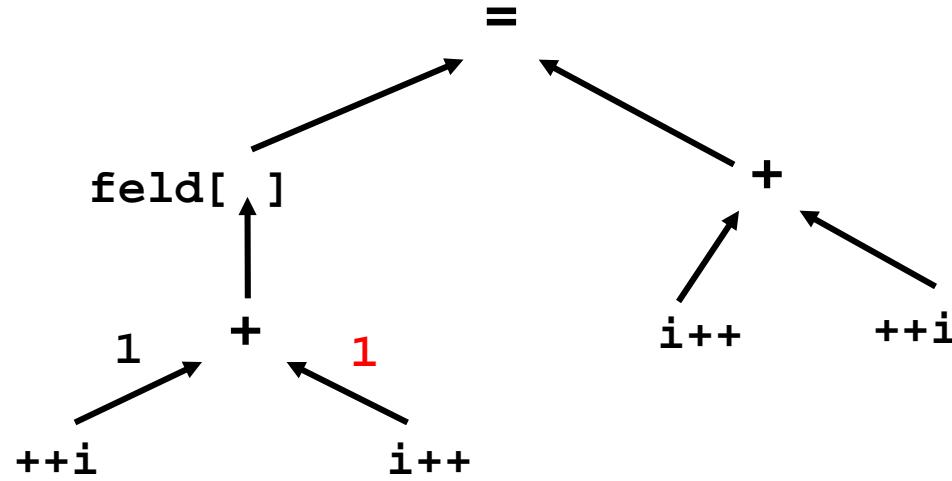
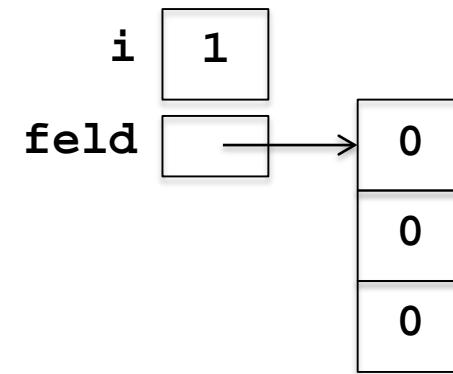
```
int i = 0;  
int[] feld = {0,0,0};  
feld[  
++i + i++ = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



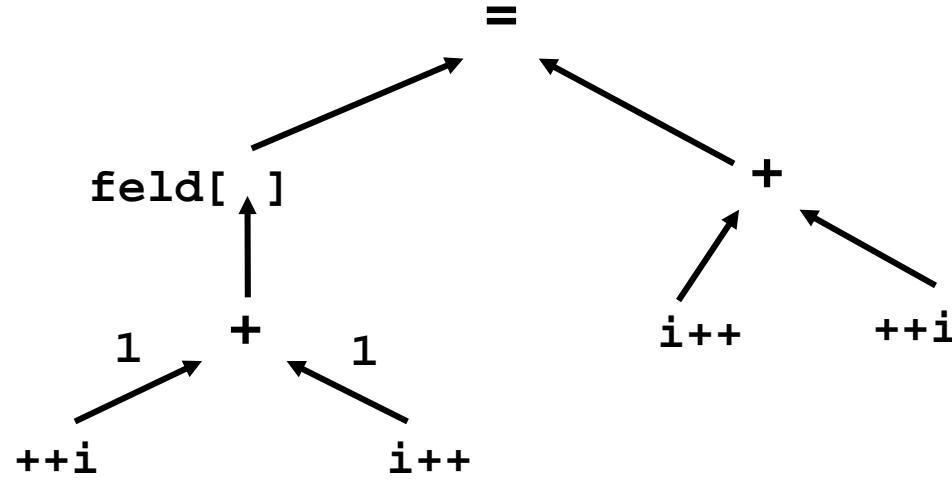
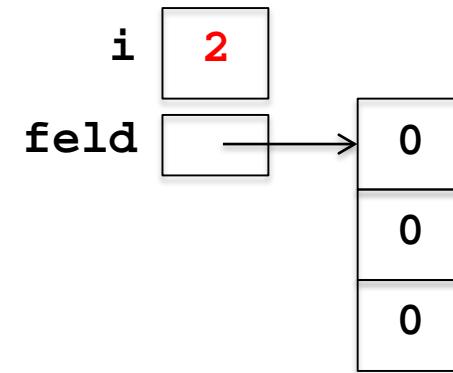
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



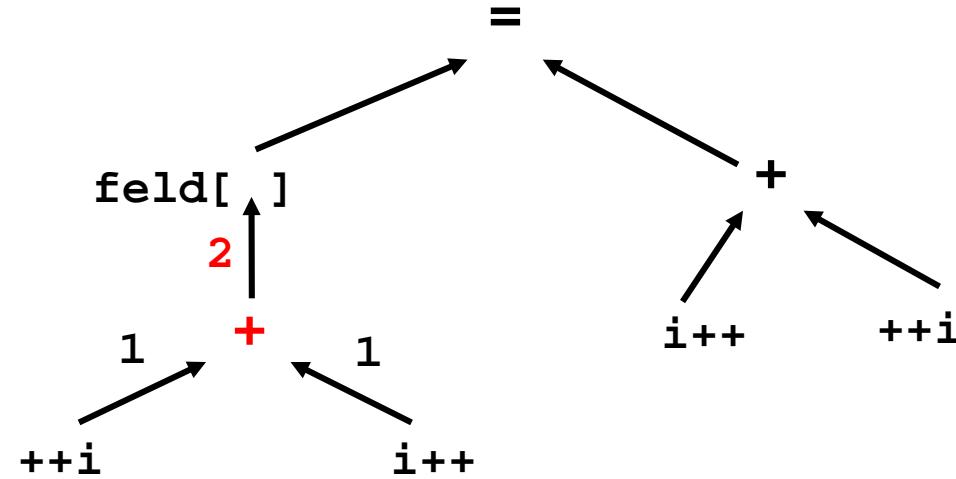
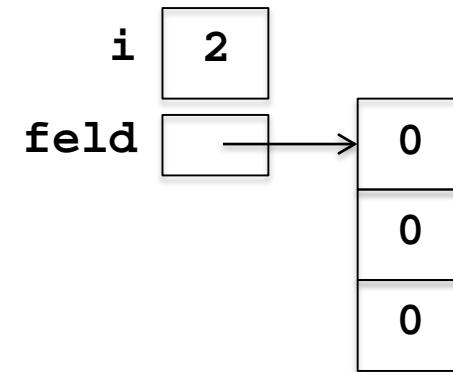
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



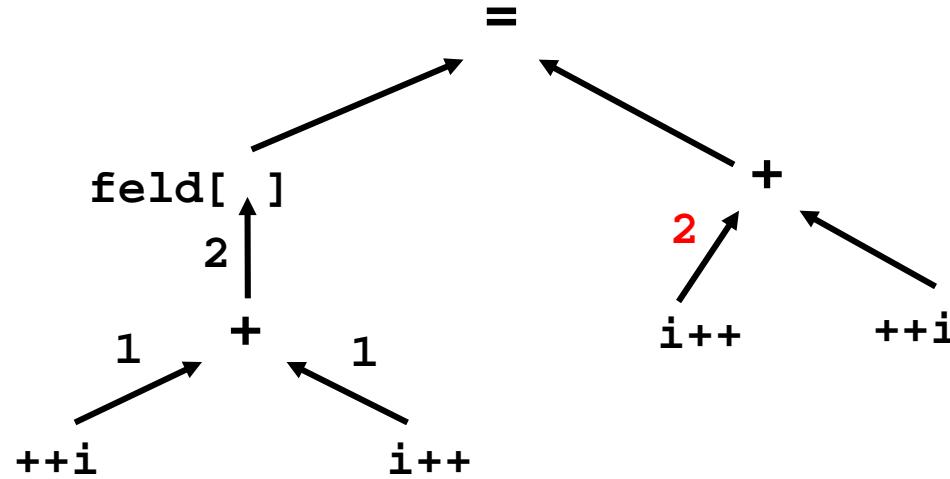
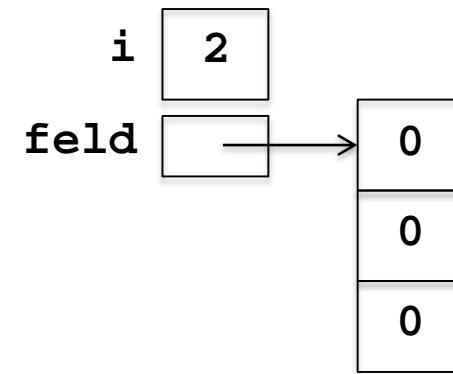
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



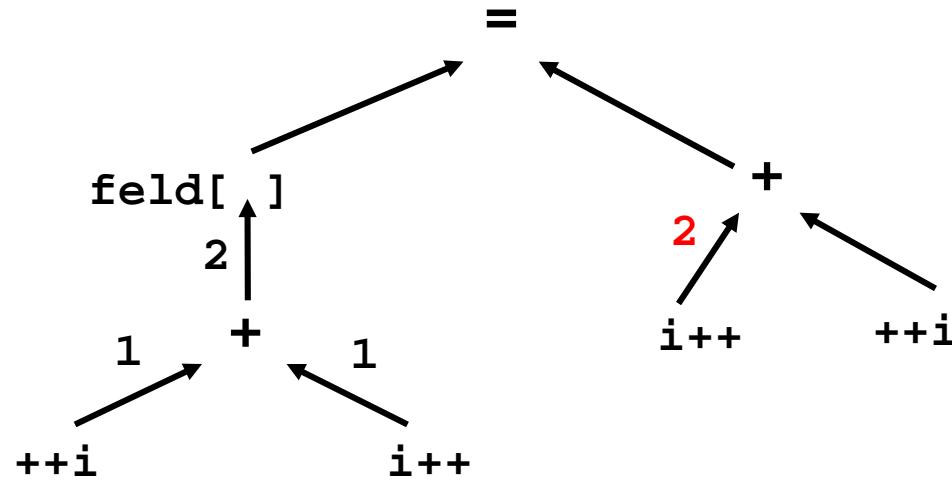
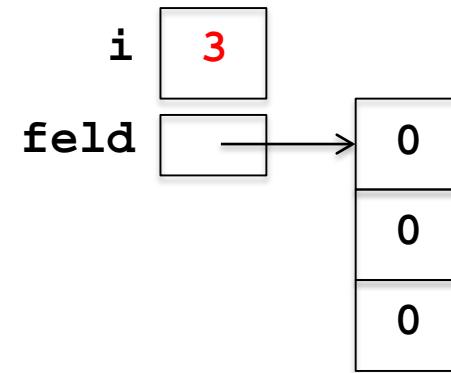
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



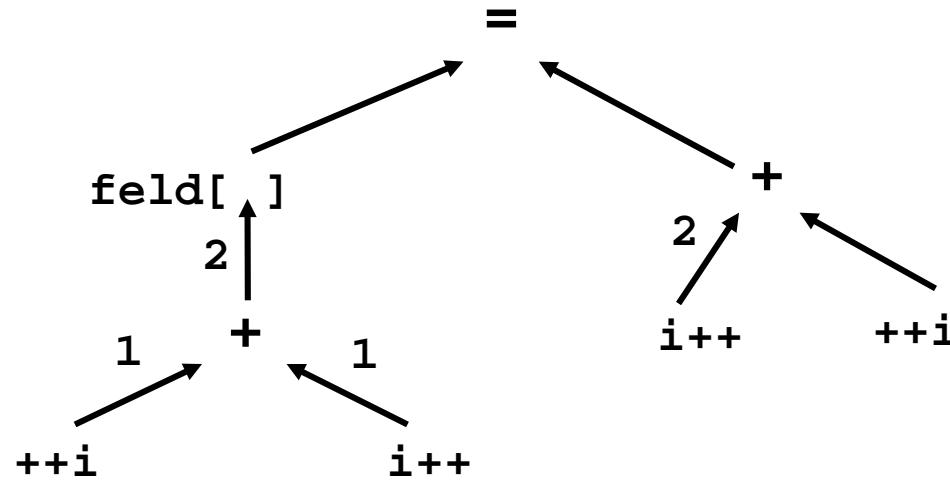
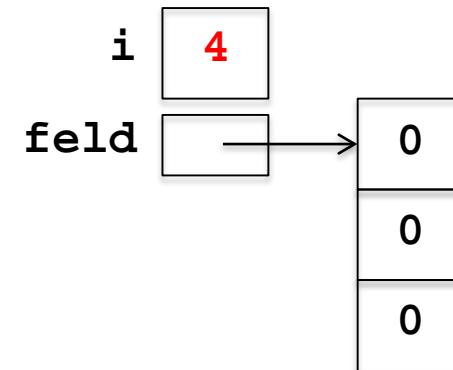
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



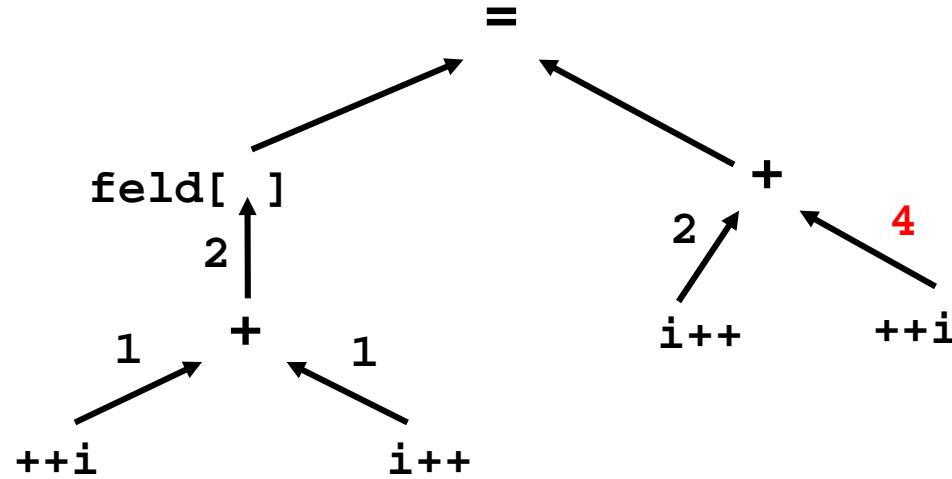
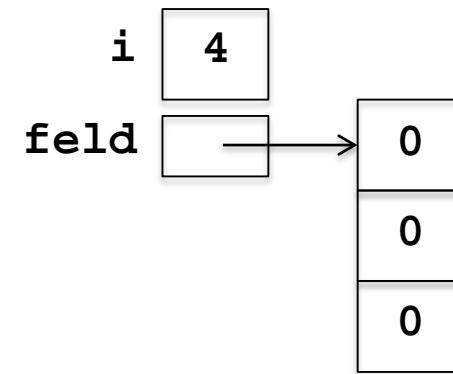
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



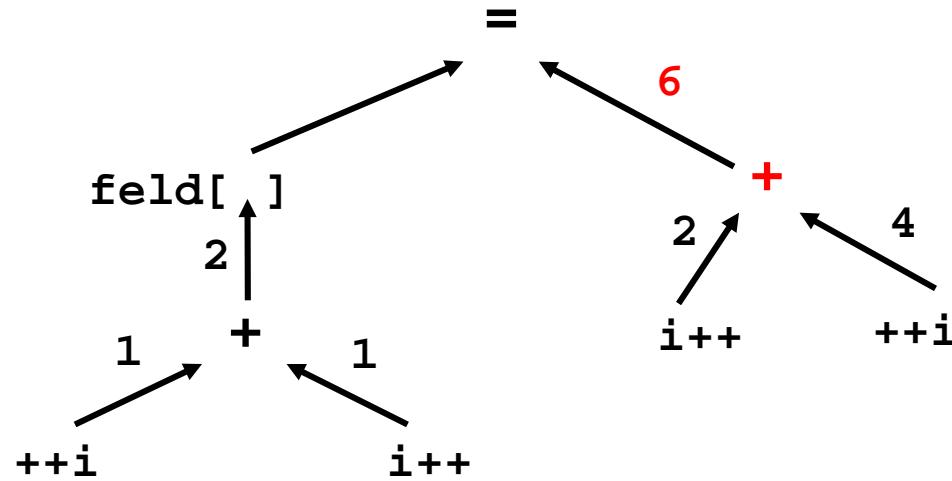
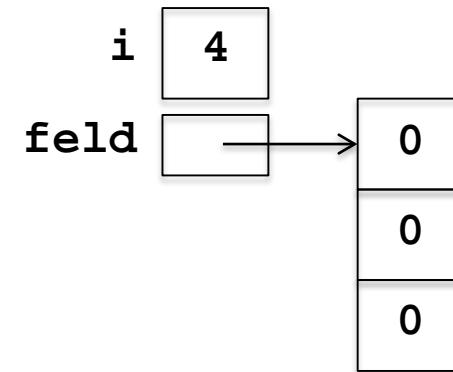
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



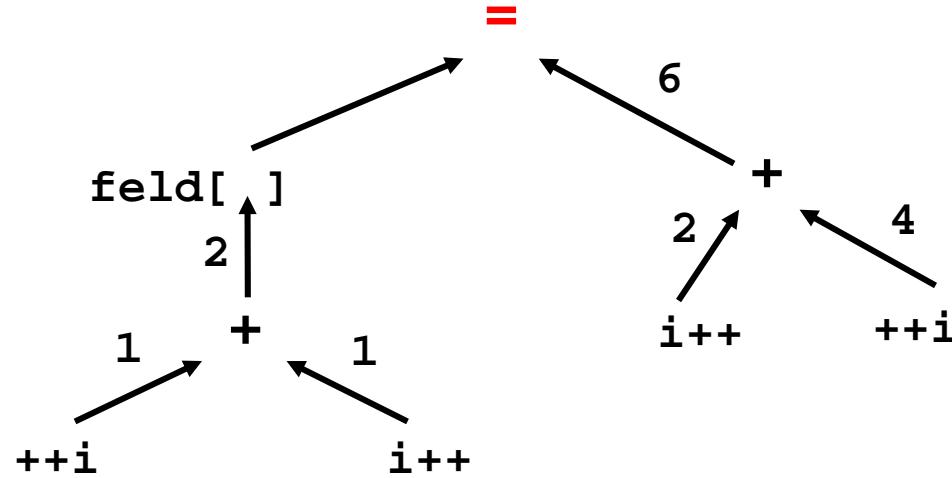
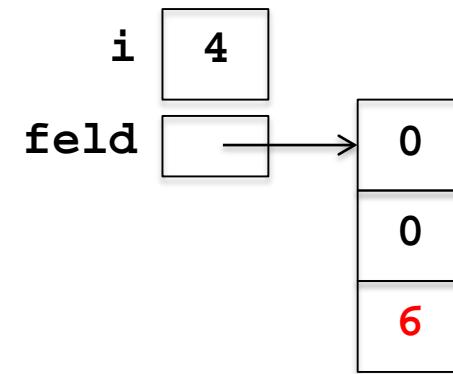
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



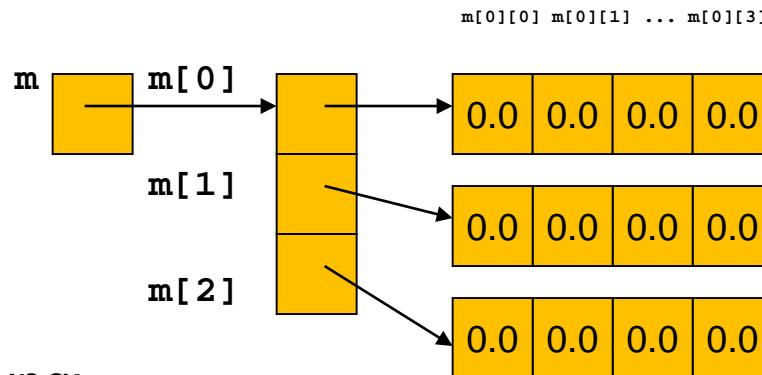
```
int i = 0;  
int[] feld = {0,0,0};  
feld[++i + i++] = i++ + ++i;  
System.out.println(feld[0]);  
System.out.println(feld[1]);  
System.out.println(feld[2]);  
System.out.println(i);
```



- Beispiel "zweidimensionales rechteckiges Feld"

```
double[][] m = new double[3][4];
```

"Feld mit Komponententyp **Feld mit Komponententyp double**"



Alternative Erzeugung:

```
double[][] m;  
m = new double[3][];  
for (int i=0; i<3; i++)  
    m[i] = new double[4];
```

Wichtig: In beiden Fällen gilt

m.length ist 3

m[i].length ist 4

Initialisierung bei der Deklaration

```
int[][][] dreieck =  
    {{1}, {1,1}, {1,2,1}, {1,3,3,1}, {1,4,6,4,1}};
```

Allgemeine Form:

```
<Typ>[ ][ ]...[ ] <Feldname> = new <Typ>[a1]...[ak][ ]...[ ];
```

<Typ>

- beliebiger Typ

<Feldname>

- Bezeichner

a₁,...,a_k

- ganzzahlige Ausdrücke, k > 0

links und rechts muss die **gleiche** Anzahl eckiger Klammern stehen

Zweite Variante:

```
<Typ>[ ][ ]...[ ] <Feldname>;  
<Feldname> = new <Typ>[a1]...[ak][ ]...[ ];
```

Beachten Sie:

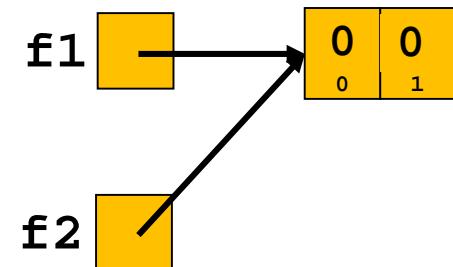
Bei einem mehrdimensionalen Feld müssen immer zuerst die ersten Dimensionen initialisiert werden.

```
int[][][] big = new int[][][2]; // Fehler!
```

→ Vorsicht beim Kopieren von Referenztypen!

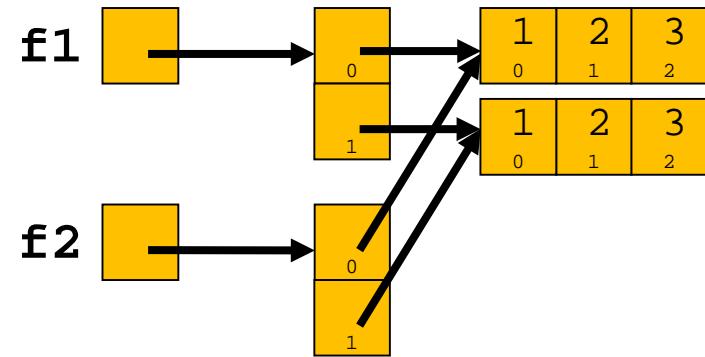
- Es werden stets nur die Referenzen kopiert. Werden andere Kopien erwünscht, z.B. **vollständige (tiefe)** Kopien (siehe nächste Folie), so muss **Cloning** benutzt (und eventuell / je nach Anwendungsfall selbstständig implementiert) werden!

- Man unterscheidet
 - **Referenzkopie**: Nur die Referenz wird kopiert (Kopieren des „Werts“ einer Referenzvariable in eine andere)
 - **Flache Kopie**: Lediglich die erste Ebene einer Struktur wird dupliziert
 - **Tiefe Kopie**: Alle Ebenen einer Struktur werden dupliziert. Dies nennt man auch **Clonen** (später)
- Anwendung auf Felder
 - **Referenzkopie**: durch die Anweisung `f2 = f1` wird lediglich eine Kopie des „Wertes“ (Pfeil) von `f1` erzeugt; eine Referenz auf dieselbe Struktur
 - Eine Veränderung von `f2` bewirkt eine Veränderung von `f1`



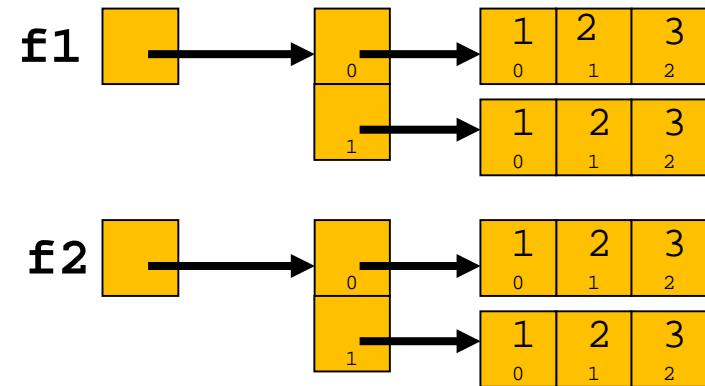
Flache Kopie

- Es wird lediglich ein Duplikat für die erste Ebene erstellt.
Alle weiteren Ebenen sind identisch
- Änderungen von der ersten Ebene von **f2** wirken sich nicht auf **f1** aus
- **Übung:** Wie muss der zugehörige Quelltext dazu aussehen?



Tiefe Kopie

- Die komplette Struktur wird dupliziert.
f2 ist völlig unabhängig von **f1**
- **Übung:** Wie muss der zugehörige Quelltext dazu aussehen?



- Der Zugriff von außen auf Klassen bzw. Objekte und deren Komponenten (Variablen und Methoden) wird geregelt durch sogenannte **Zugriffs-Modifizierer** (mehr später)
- Es existieren vier verschiedene Zugriffsberechtigungen
 - **public** (Klassendiagramm-Notation +)
in allen Klassen kann zugegriffen werden
 - **private** (Klassendiagramm-Notation -)
nur innerhalb der Klasse kann zugegriffen werden
 - **protected** (Klassendiagramm-Notation #)
in der Klasse selbst, in *allen* abgeleiteten Klassen und in anderen Klassen desselben Pakets kann zugegriffen werden (mehr später)
 - kein Modifizierer (default „*package-private*“)
in der Klasse selbst und in anderen Klassen desselben Pakets kann zugegriffen werden (Klassendiagramm-Notation ~)

- **Bisher:** Komponenten einer Klasse werden mit jedem neuen Objekt (jeder neuen Instanz) neu erzeugt: **Instanzvariablen**
- **Alternative:** Verwendung des Modifizierers **static** zur Kennzeichnung von Komponenten, die genau einmal erzeugt werden (beim Laden der Klasse) und für jedes Objekt der Klasse denselben Wert haben: **Klassenvariablen**
- Beispiel:

```
public class Mitspieler {  
    public static int gesamtzahl = 0;  
    public int nummer;  
    public String name;  
}
```

- **Klassenvariable**, d.h. als **static** deklarierte Variable gehören zur Klasse und nicht zu deren Instanzen
- Sie werden genau einmal pro Klasse angelegt und erhalten, wenn kein anderer Wert festgelegt wird, den entsprechenden **Default Wert** des ihnen zugeordneten Typs
- Auf Klassenvariable greift man über den Klassennamen (möglich ist auch der Zugriff über eine Objektreferenz) zu:

<KLASSENNAME> . <KLASSENVARIABLE>

- Die übrigen Variablen nennt man **Instanzvariablen**
Instanzvariablen werden für jedes Objekt neu erzeugt

```
public class TestSpieler {  
    public static void main (String[] args) {  
        Mitspieler ich, du, er;  
        ich = new Mitspieler();  
        ich.nummer = ++Mitspieler.gesamtzahl;  
        ich.name = "Hugo";  
        du = new Mitspieler();  
        du.nummer = ++Mitspieler.gesamtzahl;  
        du.name = "Hilde";  
        er = new Mitspieler();  
        er.nummer = ++Mitspieler.gesamtzahl;  
        er.name = "Otto";  
        System.out.println("Du bist Spieler Nr. "  
                           + du.nummer + " von insgesamt "  
                           + Mitspieler.gesamtzahl);  
    }  
}
```

Ausgabe:

Du bist der Spieler Nr. 2 von insgesamt 3

Danach geben die Anweisungen

```
System.out.println(er.gesamtzahl)  
System.out.println(du.gesamtzahl)  
System.out.println(ich.gesamtzahl)
```

allesamt den Wert 3 aus!

```
public class Punkt { // Punkt in der Ebene
    public double x; // x-Koordinate des Punktes
    public double y; // y-Koordinate des Punktes
}
```

```
public class Rechteck { // Rechteck in der Ebene
    public double breite; // Breite des Rechtecks
    public double hoehe; // Hoehe des Rechtecks
    public Punkt luEcke; // Position der linken unteren
                         // Ecke des Rechtecks
}
```

Übung (2)

```

public class TestRechteck {
    public static void main(String[] args){
        Punkt p = new Punkt();
        p.x = 3;
        p.y = 4;

        Rechteck r1 = new Rechteck();
        r1.breite = 7;
        r1.hoehe = 4;
        r1.luEcke = p;

        Rechteck r2 = new Rechteck();
        r2.breite = 1;
        r2.hoehe = 2;
        r2.luEcke = new Punkt();
        r2.luEcke.x = 5;
        r2.luEcke.y = 0;

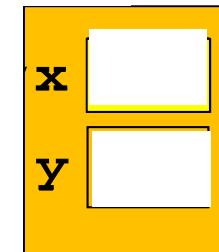
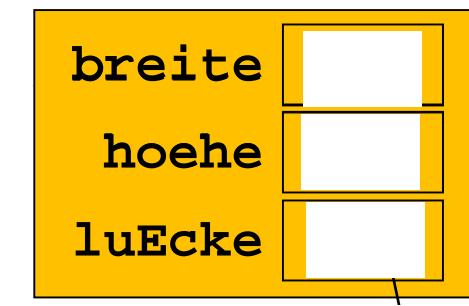
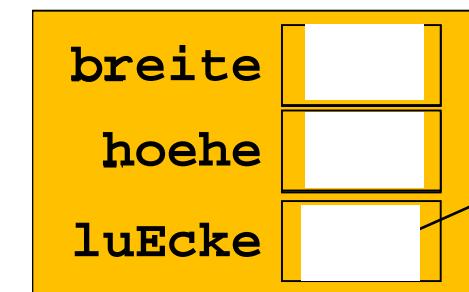
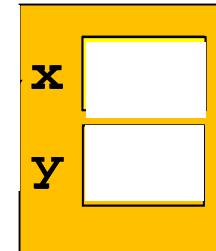
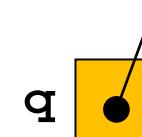
        Punkt q = p;
        q.x = 10;

        System.out.println(p.x);
        System.out.println(r1.luEcke.x);

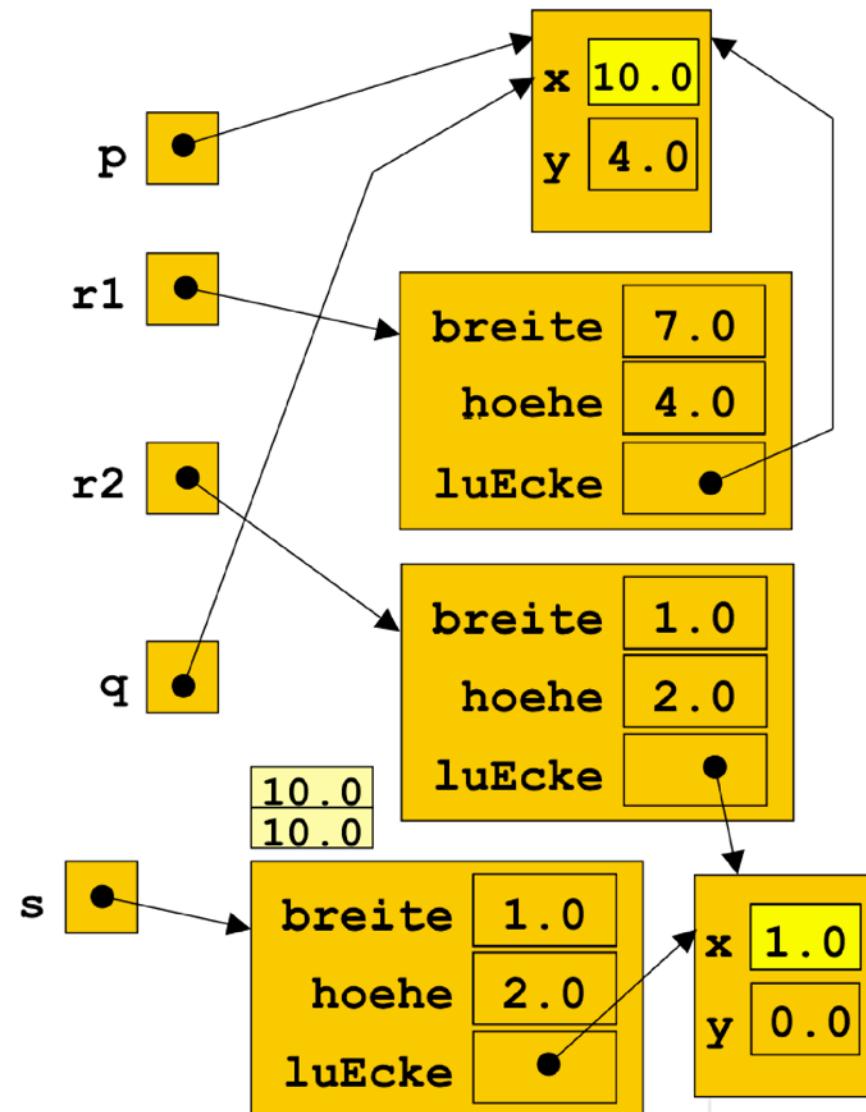
        Rechteck s = new Rechteck();
        s.breite = r2.breite;
        s.hoehe = r2.hoehe;
        s.luEcke = r2.luEcke;
        r2.luEcke.x = 1;

        System.out.println(s.luEcke.x);
    }
}

```



```
public class TestRechteck {  
    public static void main(String[] args){  
        Punkt p = new Punkt();  
        p.x = 3;  
        p.y = 4;  
  
        Rechteck r1 = new Rechteck();  
        r1.breite = 7;  
        r1.hoehe = 4;  
        r1.luEcke = p;  
  
        Rechteck r2 = new Rechteck();  
        r2.breite = 1;  
        r2.hoehe = 2;  
        r2.luEcke = new Punkt();  
        r2.luEcke.x = 5;  
        r2.luEcke.y = 0;  
  
        Punkt q = p;  
        q.x = 10;  
  
        System.out.println(p.x);  
        System.out.println(r1.luEcke.x);  
  
        Rechteck s = new Rechteck();  
        s.breite = r2.breite;  
        s.hoehe = r2.hoehe;  
        s.luEcke = r2.luEcke;  
        r2.luEcke.x = 1;  
  
        System.out.println(s.luEcke.x);  
    }  
}
```

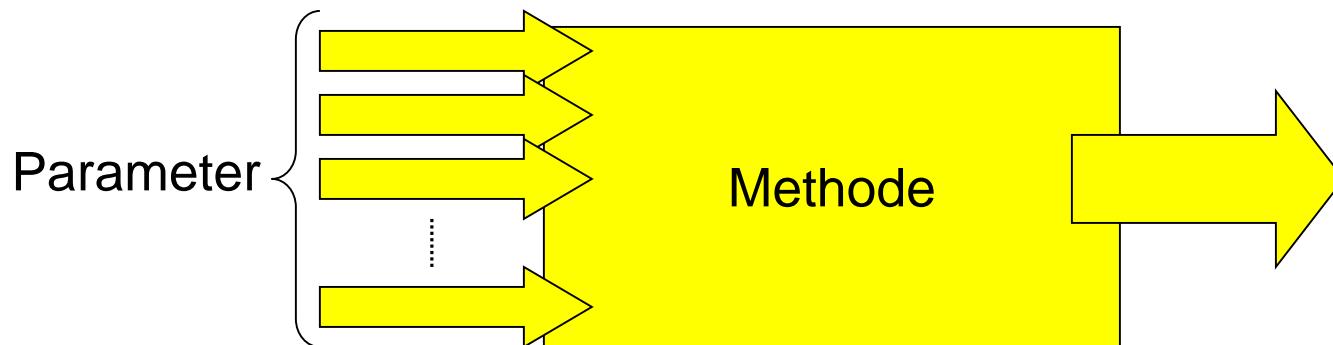


- Instanzvariablen werden bei der Anlage eines Objektes mit ihren Standardwerten (Default-Werten) belegt

byte	(byte) 0
short	(short) 0
int	0
long	0L
char	(char) 0
boolean	false
float	0.0F
double	0.0D
Referenztyp	null

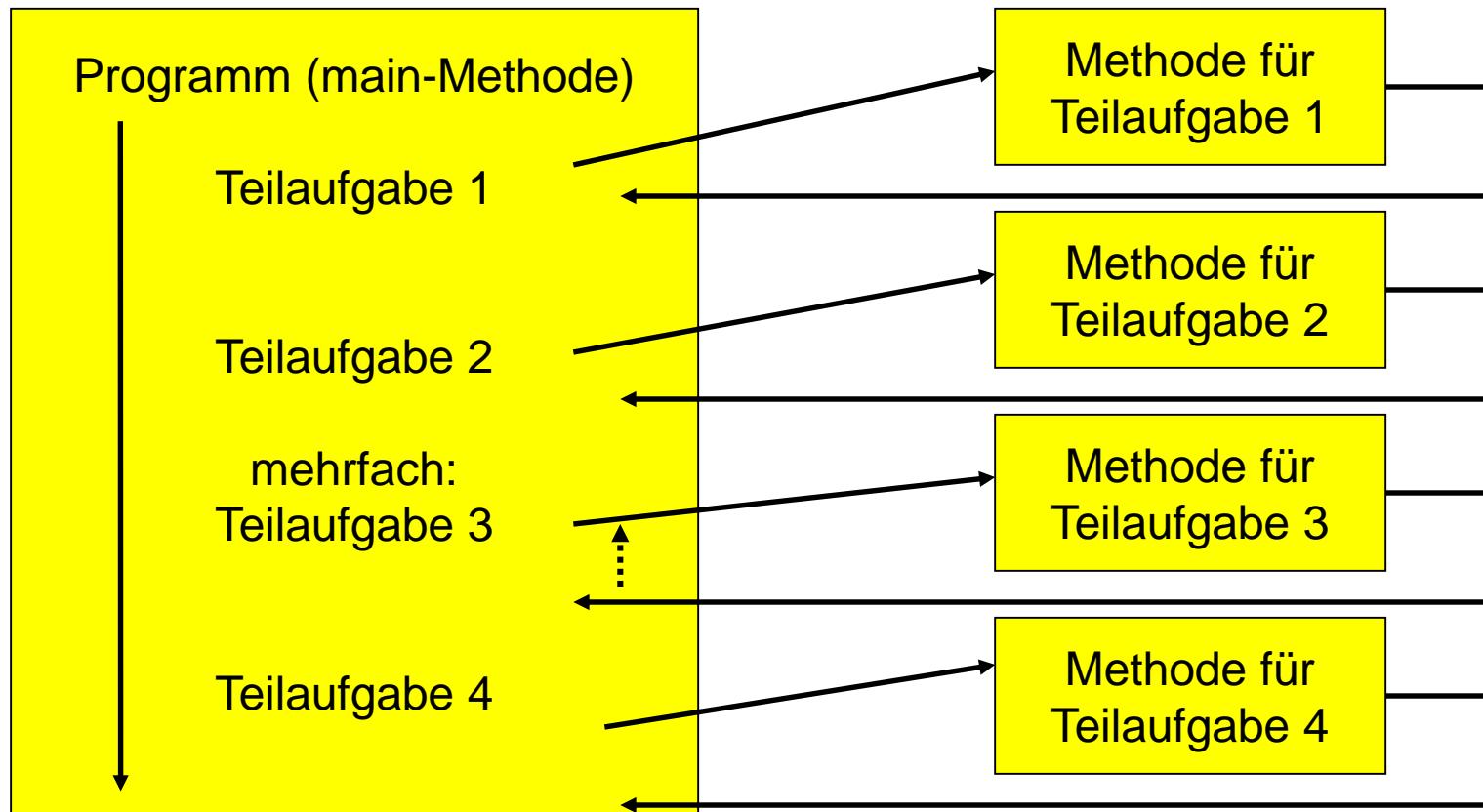
Methoden

- fassen Teile des ausführbaren Programmcodes zu **Unterprogrammen** zusammen
- können durch einen **Methodenaufruf** ausgeführt werden
- können **Parameter (Argumente)** übergeben bekommen
- können ein Ergebnis (**Rückgabewert**) zurückliefern



Bedeutung von Methoden

- Strukturierung bzw. Modularisierung des Programms
- Reduzierung des Programmieraufwands
- Leichtere Anpassbarkeit von Programmen



Am Beispiel

Modifizierer Ergebnistyp Bezeichner Formale Parameter

```
public static int spezialSumme(int m, int[] feld) {
```

Lokale Variable

```
{ int temp = 0;  
  
    for (int i=0; i<feld.length; i++) {  
        temp = temp + m*feld[i];  
        m = m + 1;  
    }
```

```
}  
  
return temp; }
```

Formale Parameter

Methoden-Rumpf

Ausführungsende,
Rückgabe des Wertes

- Steht in einem Ausdruck ein Methodenaufruf, so wird bei der Auswertung des Ausdrucks an der entsprechenden Stelle diese Methode ausgeführt und mit dem Rückgabewert weitergerechnet
- Am Beispiel

```
...
int k, ergebnis;
int[] x = new int[20];
Scanner input = new Scanner(System.in);
System.out.println("k = ");
k = input.nextInt();
for (int i=0; i<x.length; i++) {
    System.out.println("x[" + i + "] = ");
    x[i] = input.nextInt();
}
ergebnis = 5 - 3*spezialSumme(3*k,x);
...
```

Methoden-Name

aktuelle Parameter

```
public class MyMethods { // Eine Methoden-Sammlung

    // (1) Quadrat-Funktion
    public static double sqr(double x) {
        return x*x;
    }

    // (2) Quersummen-Funktion
    public static int quersumme (int k) {
        int qs = 0;
        while (k != 0) {
            qs = qs + k % 10;
            k   = k / 10;
        }
        return qs;
    }
}
```

```
// (3) Fakultäts-Funktion

public static long fakultaet(long k) {
    if (k <= 1)
        return 1;
    else {
        long f = 1;
        while (k > 1) {
            f = f * k;
            k--;
        }
        return f;
    }
}
```

```
// (4) Mittelwert-Funktion für Felder
public static double mittelwert(double[] feld) {
    double mw = 0;
    for (int i=0; i < feld.length; i++)
        mw = mw + feld[i];
    return mw / feld.length;
}
```

```
// (5) Kopier-Funktion für Felder (flache Kopie)
public static double[] kopie(double[] x) {
    double[] y = new double[x.length];
    for (int i=0; i < x.length; i++)
        y[i] = x[i];
    return y;
}
```

```
// (6) Vereinfachte println-Methode
public static void println(String s) {
    System.out.println(s);
}

// (7) Wurzel-2-Funktion
public static double wurzel2() {
    return Math.sqrt(2);
}

// (8) Nichstun-Methode :-)
public static void tuNix() {
    return; // kann auch entfallen!
}

// (9) Testfunktion für eine gerade Zahl
public static boolean istGerade(long k) {
    return (k % 2 == 0);
}
```

```
// (10) main-Methode testet alle obigen Methoden
public static void main(String[] args) {
    double x = 1.7, y;
    long k;
    double[] feldchen = {1.0, 2.0, 3.0, 4.0};
    double[] nochEins;
    y = sqr(x);
    System.out.println(quersumme(12345));
    k = fakultaet(12/3);
    x = mittelwert(feldchen);
    nochEins = kopie(feldchen);
    println("I say hello " + 2 + " u");
    println("Wurzel aus 2 ist " + wurzel2());
    tuNix();
    if (istGerade(k))
        println(k + " ist eine gerade Zahl");
    // k = fakultaet(x);          //unzulässig
    // println(y);                //unzulässig
}
} // Ende der Klasse MyMethods
```



- Erzeuge die formalen Parameter-Variablen der Methode
- Initialisiere diese mit den *Werten* der aktuellen Parameter (Argumente) wie sie beim Aufruf angegeben sind
- Führe den Rumpf der Methode aus
- Gib den Rückgabewert (Ergebniswert) zurück

Call-by-Value / Pass-by-Value Semantik

Die Methode arbeitet mit *Kopien* der aktuellen Parameter, so dass die aktuellen Parameter nicht verändert werden!

Aber Vorsicht

Ist der aktuelle Parameter eine **Referenz**, so können auch von der Methode aus über diese Referenz auf Objekte der aufrufenden Umgebung zugegriffen und dort Veränderungen verursacht werden!

Wird also die Methode

Definition siehe Folie 5

```
public static int spezialSumme(int m, int[] feld)
```

aufgerufen in der Form

```
ergebnis = 5 - 3*spezialSumme(3*k,x);
```

so wird folgendes passieren:

In der Methode **spezialSumme** erhalten

m den Wert von **3*k**

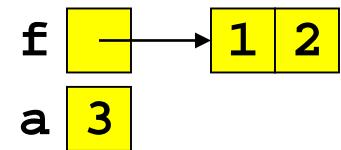
feld den Wert von **x** (Referenzkopie!)

Die Anweisungen der Methode werden ausgeführt.

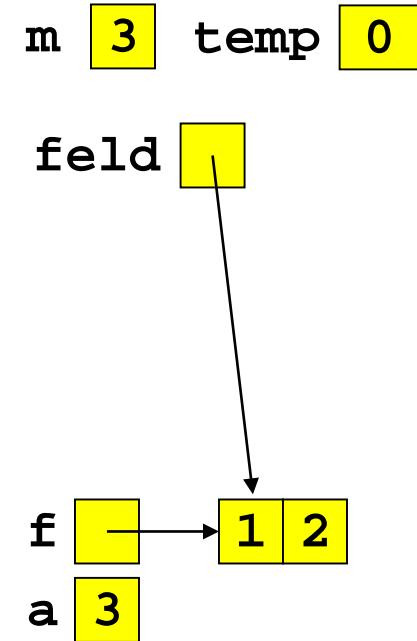
Der Rückgabewert (**rgw**) wird erzeugt. Der Ausdruck wird unter Verwendung von **rgw** weiter ausgewertet gemäß

```
ergebnis = 5 - 3*rgw;
```

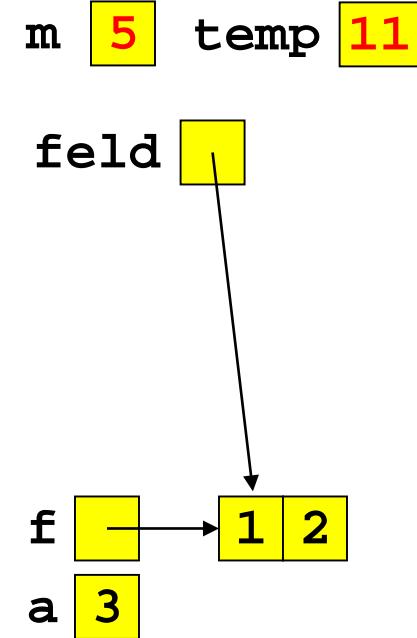
```
public class MethodenAblaufTest {  
    public static int  
        spezialSumme(int m, int[] feld) {  
            int temp = 0;  
            for (int i=0; i<feld.length; i++) {  
                temp = temp + m*feld[i];  
                m = m + 1;  
            }  
            feld[1] = 5;  
            return temp;  
        }  
    public static void main (String[] args) {  
        int[] f = {1, 2};  
        int a = 3;  
        int s = spezialSumme(a, f);  
        System.out.println("f[1] = " + f[1]);  
        System.out.println("a = " + a);  
        System.out.println("s = " + s);  
    }  
}
```



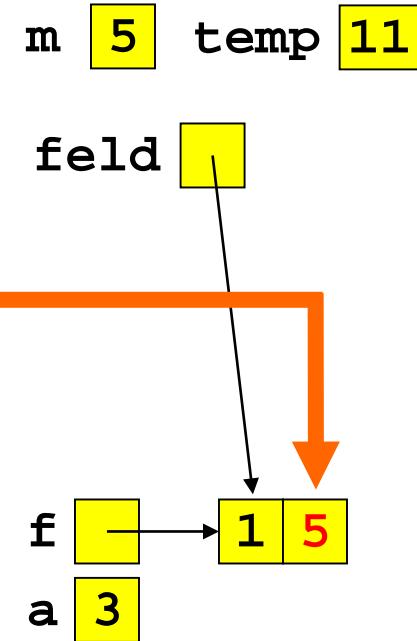
```
public class MethodenAblaufTest {  
    public static int  
        spezialSumme(int m, int[] feld) {  
            int temp = 0;  
            for (int i=0; i<feld.length; i++) {  
                temp = temp + m*feld[i];  
                m = m + 1;  
            }  
            feld[1] = 5;  
            return temp;  
        }  
    public static void main (String[] args) {  
        int[] f = {1, 2};  
        int a = 3;  
        int s = spezialSumme(a, f);  
        System.out.println("f[1] = " + f[1]);  
        System.out.println("a = " + a);  
        System.out.println("s = " + s);  
    }  
}
```



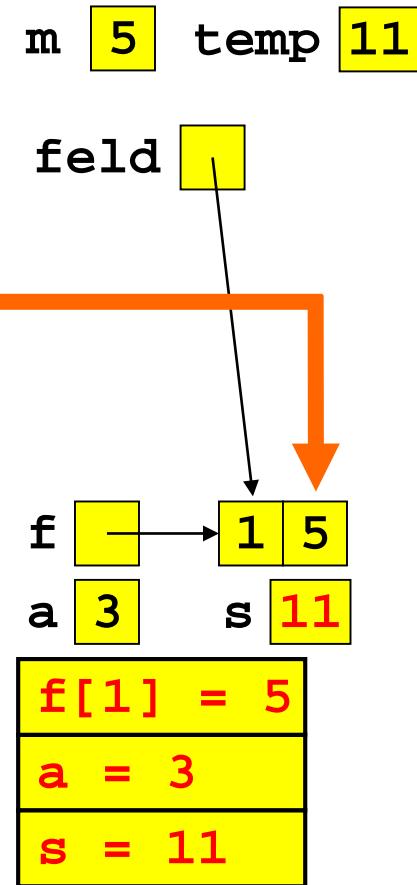
```
public class MethodenAblaufTest {  
    public static int  
        spezialSumme(int m, int[] feld) {  
            int temp = 0;  
            for (int i=0; i<feld.length; i++) {  
                temp = temp + m*feld[i];  
                m = m + 1;  
            }  
            feld[1] = 5;  
            return temp;  
        }  
    public static void main (String[] args) {  
        int[] f = {1, 2};  
        int a = 3;  
        int s = spezialSumme(a, f);  
        System.out.println("f[1] = " + f[1]);  
        System.out.println("a = " + a);  
        System.out.println("s = " + s);  
    }  
}
```



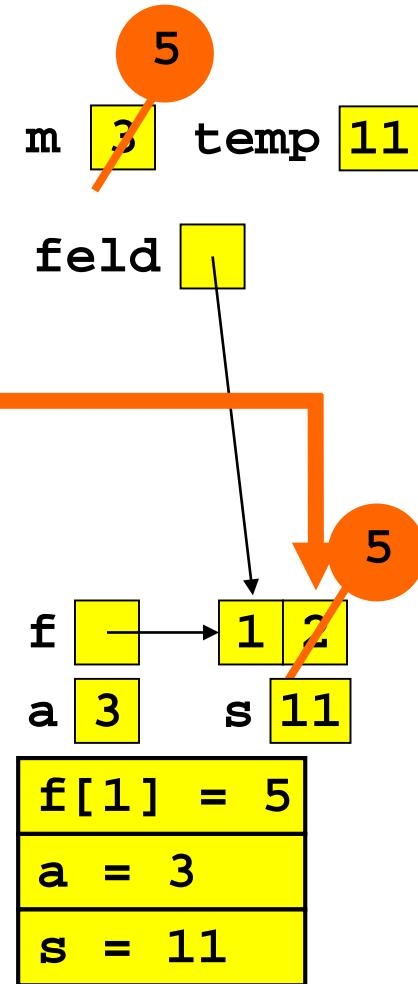
```
public class MethodenAblaufTest {  
    public static int  
        spezialSumme(int m, int[] feld) {  
            int temp = 0;  
            for (int i=0; i<feld.length; i++) {  
                temp = temp + m*feld[i];  
                m = m + 1;  
            }  
            feld[1] = 5; —————  
            return temp;  
        }  
    public static void main (String[] args) {  
        int[] f = {1, 2};  
        int a = 3;  
        int s = spezialSumme(a, f);  
        System.out.println("f[1] = " + f[1]);  
        System.out.println("a = " + a);  
        System.out.println("s = " + s);  
    }  
}
```



```
public class MethodenAblaufTest {  
    public static int  
        spezialSumme(int m, int[] feld) {  
            int temp = 0;  
            for (int i=0; i<feld.length; i++) {  
                temp = temp + m*feld[i];  
                m = m + 1;  
            }  
            feld[1] = 5; —————  
            return temp;  
        }  
    public static void main (String[] args) {  
        int[] f = {1, 2};  
        int a = 3;  
        int s = spezialSumme(a, f);  
        System.out.println("f[1] = " + f[1]);  
        System.out.println("a = " + a);  
        System.out.println("s = " + s);  
    }  
}
```



```
public class MethodenAblaufTest {  
    public static int  
        spezialSumme(int m, int[] feld) {  
            int temp = 0;  
            for (int i=0; i<feld.length; i++) {  
                temp = temp + m*feld[i];  
                m = m + 1;  
            }  
            feld[1] = 5; —————  
            return temp;  
        }  
    public static void main (String[] args) {  
        int[] f = {1, 2};  
        int a = 3;  
        int s = spezialSumme(a, f);  
        System.out.println("f[1] = " + f[1]);  
        System.out.println("a = " + a);  
        System.out.println("s = " + s);  
    }  
}
```



```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```

```
a = 5, b = 100  
help = 5  
a = 100  
b = 5
```

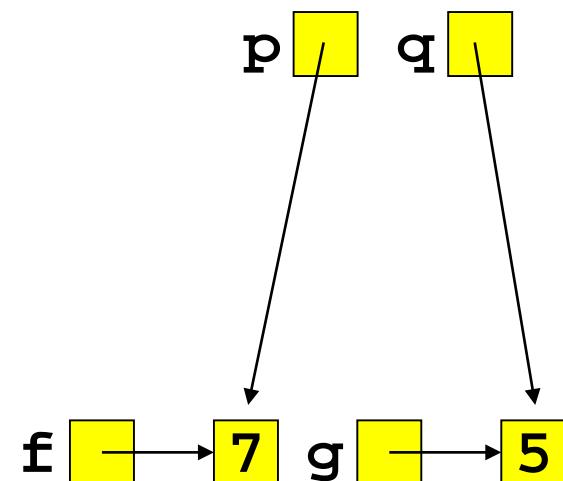


```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```

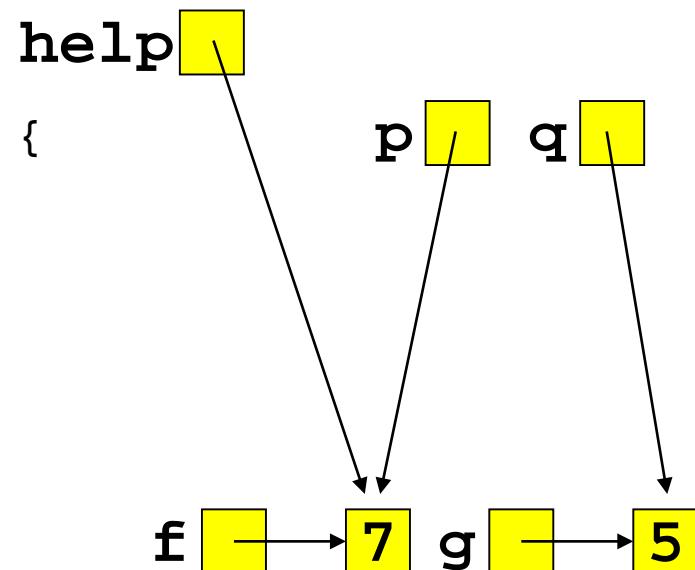
x = 5, y = 100



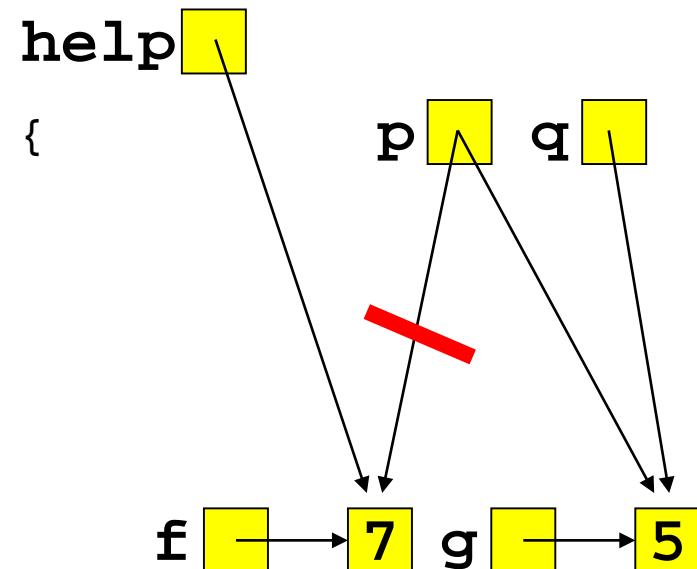
```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



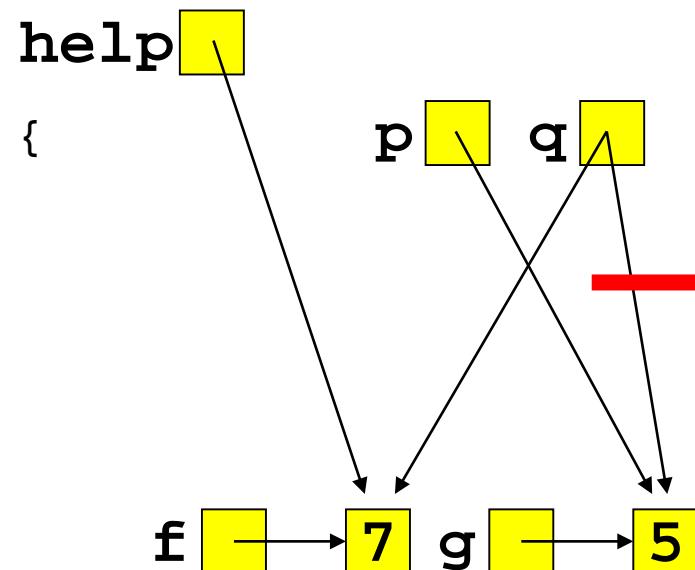
```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]="+ g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]="+ g[0]);  
    }  
}
```

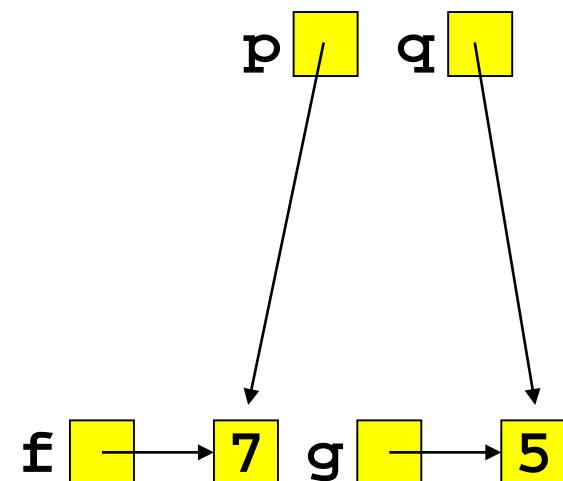


```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



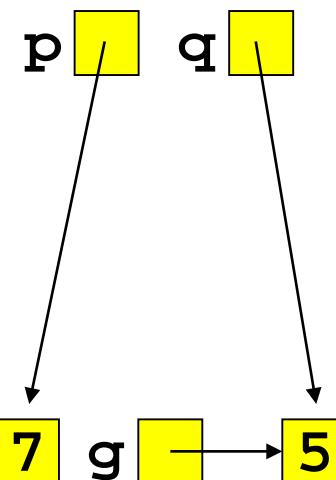
f[0]=7 g[0]=5

```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



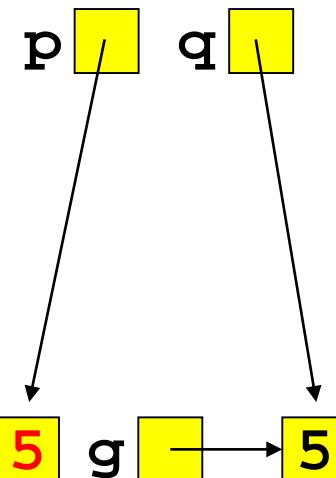
```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=\"" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=\"" + f[0] + " g[0]=\"" + g[0]);  
    }  
}
```

help 7



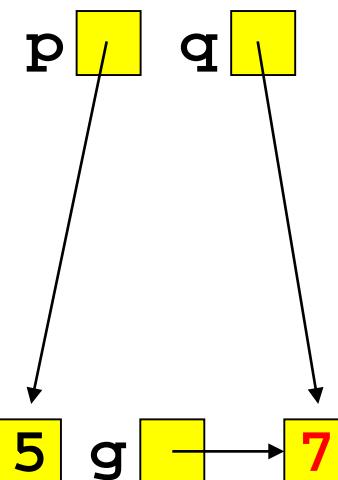
```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=\"" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=\"" + f[0] + " g[0]=\"" + g[0]);  
    }  
}
```

help 7



```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]="+ g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]="+ g[0]);  
    }  
}
```

help 7



```
public class TauschenTest {  
    public static void tauschel(int a, int b) {  
        int help = a;  
        a = b; b = help;  
    }  
    public static void tausche2(int[] p, int[] q) {  
        int[] help = p;  
        p = q; q = help;  
    }  
    public static void tausche3(int[] p, int[] q) {  
        int help = p[0];  
        p[0] = q[0]; q[0] = help;  
    }  
    public static void main(String[] args) {  
        int x = 5, y = 100;  
        int[] f = {7}, g = {5};  
        tauschel(x, y);  
        System.out.println("x=" + x + " y=" + y);  
        tausche2(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
        tausche3(f, g);  
        System.out.println("f[0]=" + f[0] + " g[0]=" + g[0]);  
    }  
}
```



f[0]=5 g[0]=7



Ausgaben

x=5 y=100

f[0]=7 g[0]=5

f[0]=5 g[0]=7

Begründung

Alle drei Methoden vertauschen etwas:

tausche1 vertauscht nur ihre lokalen Kopien a und b der Werte x und y aus der Methode main. x und y bleiben unverändert!

tausche2 vertauscht nur ihre lokalen Kopien p und q der Referenzen f und g aus der Methode main. f und g bleiben unverändert!

tausche3 vertauscht durch Referenzzugriff über die lokalen Kopien p und q der Referenzen f und g aus der Methode main die Feldinhalte der durch f und g referenzierten Felder!

```
public class VerdeckenTest {  
    static int a = 1, b = 2, c = 3;  
    static int m(int a) {  
        int b = 20;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        return 100;  
    }  
    public static void main (String[] args) {  
        int a = 1000;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("m(c)= " + m(c));  
    }  
}
```

Sichtbarkeit und Verdecken

In der Klasse `VerdeckenTest` werden verwendet

 Klassenvariablen

 formale Argument-Variablen (formale Parameter)

 lokale Variablen

Innerhalb von Methoden verdecken lokale Variablen und formale Variablen die Klassenvariablen

a = 3 formaler Parameter a (Kopie von Klassen-c)



```
public class VerdeckenTest {  
    static int a = 1, b = 2, c = 3;  
    static int m(int a) {  
        int b = 20;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("c = " + c);  
        return 100;  
    }  
    public static void main (String[] args) {  
        int a = 1000;  
        System.out.println("a = " + a);  
        System.out.println("b = " + b);  
        System.out.println("m(c)= " + m(c));  
    }  
}
```

Sichtbarkeit und Verdecken

In der Klasse VerdeckenTest werden verwendet

Klassenvariablen

formale Argument-Variablen (formale Parameter)

lokale Variablen

a = 3 Parameter a (Kopie von Klassen-c)

b = 20 lokales b (verdeckt Klassen-b)

c = 3 Klassen-c

a = 3 formaler Parameter a (Kopie von Klassen-c)

a = 1000 lokales a (verdeckt Klassen-a)

b = 2 Klassen-b

m(c) = 100

Ergebniswert des m-Aufrufs

- In der Klasse **VerdeckenTest** werden verwendet
 - Klassenvariablen
 - formale Argument-Variablen (formale Parameter)
 - lokale Variablen
- Innerhalb von Methoden verdecken lokale Variablen und formale Variablen die Klassenvariablen
- Beim Start von **VerdeckenTest** ergeben sich folgende Ausgaben:

Ausgaben	Begründung
a = 1000	lokales a (verdeckt Klassen- a)
b = 2	Klassen- b
a = 3	formaler Parameter a (Kopie von Klassen- c)
b = 20	lokales b (verdeckt Klassen- b)
c = 3	Klassen- c
m(c) = 100	Ergebniswert des m -Aufrufs

- Als Elemente einer Klasse deklarierte Variablen müssen verschiedene Namen erhalten!

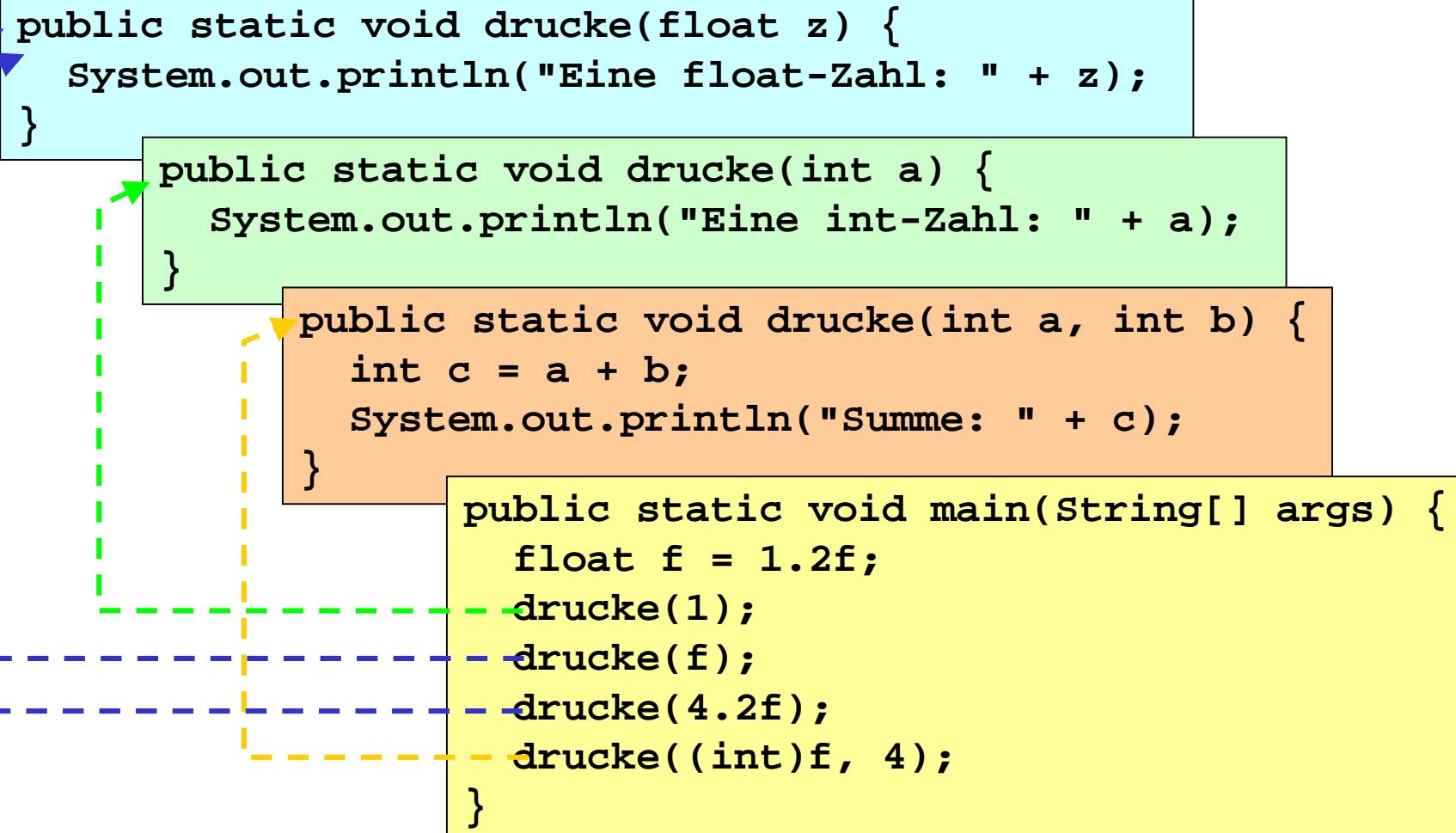
Beispiel:

```
class A {  
    double y = 18.48;  
    int     y = 12;      // Fehler : Doppeldeklaration  
    ...  
}
```

- Methoden können dagegen mit gleichem Namen vorkommen, wenn sie sich in Ihrer *Signatur* (siehe nachfolgende Definition) unterscheiden. Man spricht dann vom **Überladen** von Methoden (method overloading).

- Deklaration von mehreren Methoden mit dem gleichen Namen aber unterschiedlicher Parameter-Liste
- Unterscheidung der überladenen Methoden anhand
 - der Anzahl,
 - der Datentypen und
 - der Reihenfolgeder Parameter (bilden zusammen mit dem Methoden-Namen die so genannte **Signatur** der Methode)
- **Achtung:** Die Namen der Parameter oder die Ergebnistypen der Methoden werden **nicht** zur Unterscheidung herangezogen!
- Bekanntes Beispiel für Überladung: Math.abs

```
public static double abs (double x)
public static float abs (float x)
public static long abs (long x)
public static int abs (int x)
```



Die **Signatur** einer Methode besteht aus Ihrem **Namen** gefolgt von der **Folge der Typen der formalen Parameter** aus Ihrer Deklaration (in der originalen Reihenfolge unter Beachtung der originalen Anzahlen)

Beispiele:

```
int funktion1() { ... }
```

Signatur: **funktion1()**

```
double funktion1() { ... }
```

Signatur: **funktion1()**

Problem:
Signatur für beide gleich!

```
char buchstaben(char a, int i, String s) { ... }
```

Signatur: **buchstaben(char, int, String)**

```
char buchstaben(char a, int i, int j, String s) {
```

...

}

Signatur: **buchstaben (char, int, int, String)**

```
int berechne(double x, double y, double z) {
```

...

}

```
double berechne(double u, double v, double w) {
```

...

}

Problem:

Signatur für beide gleich!

berechne (double, double, double)

Überladen von Methoden (innerhalb einer Klasse)

- Innerhalb einer Klasse dürfen Methoden gleichen Namens vorkommen, sofern sich deren Signaturen unterscheiden
- Eine Klassendeklaration darf nicht mehr als eine Methode mit einer bestimmten Signatur enthalten
- Ergebnistyp und andere Charakteristika einer Methode sind dabei nicht von Bedeutung

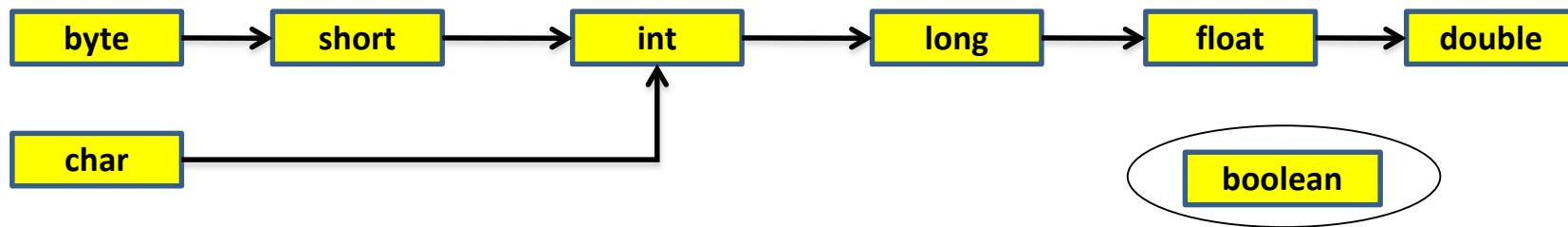
```
class Ueberladen {  
    int g() { ... }  
    int g( int i ) { ... }  
    int g( long i ) { ... }  
    int g( int i, int j ) { ... }  
    int g( double x, int j ) { ... }  
    int g( double y, int j, char a ) { ... }  
    duoble g( double x, double y ) { ... }  
    double g( int j, double x ) { ... }  
    double g( long x ) { ... }  
    // Fehler int g( long i )  
    // existiert bereits  
    ...  
}
```

Überladen von Methoden (innerhalb einer Klasse)

- Kommen mehrere Methoden mit **demselben Namen**, aber **verschiedenen Signaturen** innerhalb einer Klasse vor (es können sowohl deklarierte, als auch geerbte^[1] Methoden sein), so ist der betreffende Name **überladen**.

[1] Vererbung von Methoden wird später behandelt - siehe Abschnitt Objektorientierung

- Beim Aufruf einer überladenen Methode untersucht Java alle Methoden (auch geerbte – wird später behandelt)
 - welche den entsprechenden überladenen Methodennamen tragen,
 - deren Parameteranzahl mit der Argumentanzahl übereinstimmt,
 - für welche die Argumenttypen des Aufrufs durch elementare Typvergrößerungen oder Vergrößerungen von Referenztypen in die Parametertypen umwandelbar sind oder mit diesen übereinstimmen,
 - ob sie zugreifbar sind (abhängig von Modifikatoren `public`).



- Obige Methoden gelten als zum gegebenen aufrufenden Ausdruck **aufrufbare** Methoden. Kann keine aufrufbare Methode gefunden werden, so erfolgt eine **Fehlermeldung** seitens des Compilers.

- Ist für einen Methodenaufruf mehr als eine Methode aufrufbar, so wird zur Laufzeit die **spezifischste** ausgewählt
- Sei $m(S_1, \dots, S_n)$ und $m(T_1, \dots, T_n)$ ein Paar aufrufbarer Methoden. Dabei sei m der Name der Methode und S_i bzw. T_i (mit $1 \leq i \leq n$) seien die Typen der formalen Parameter

$m(S_1, \dots, S_n)$ heißt **spezifischer** als $m(T_1, \dots, T_n)$, falls für alle i (mit $1 \leq i \leq n$):

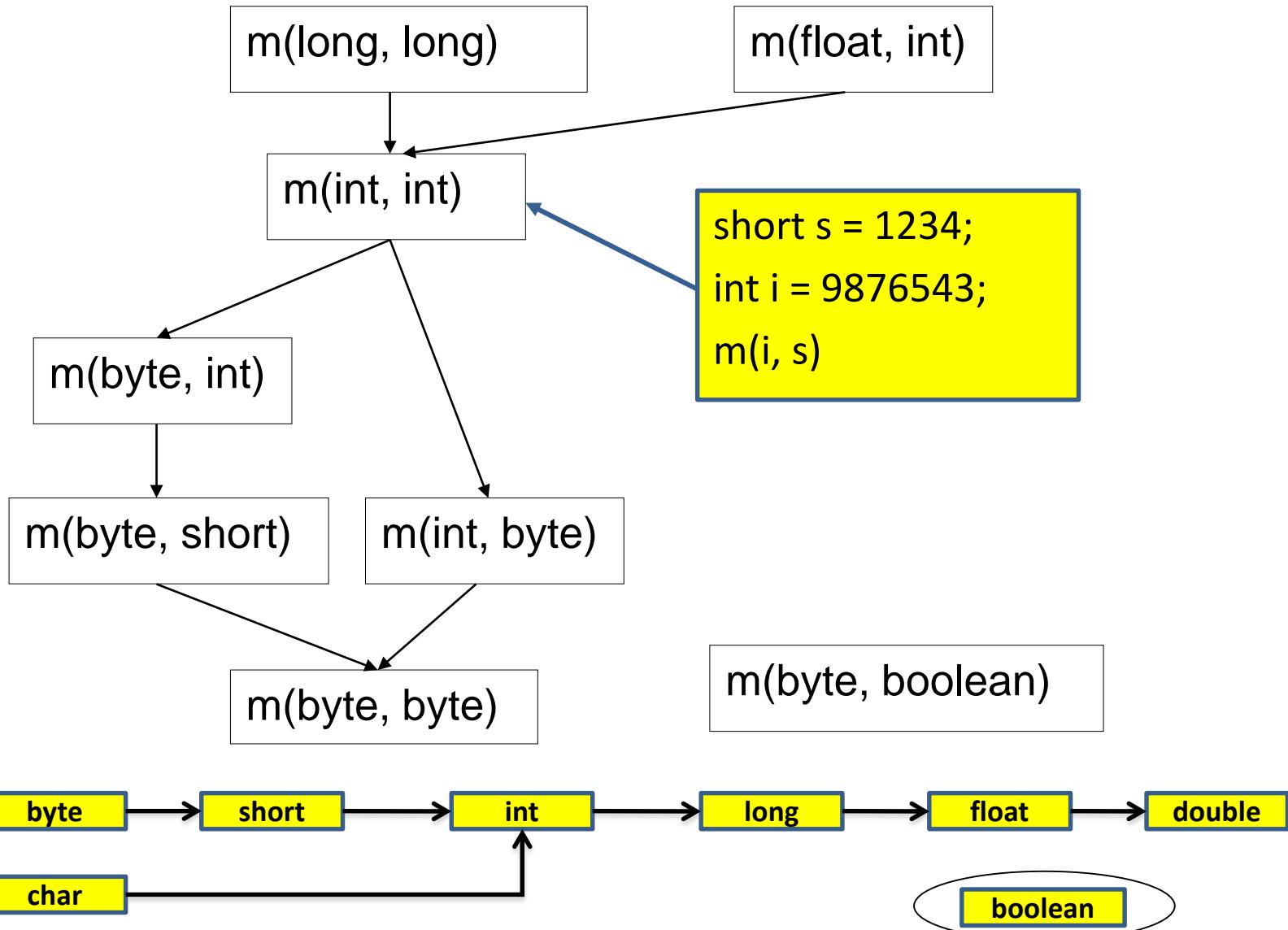
T_i durch **elementare Typvergrößerung** oder

durch **Vergrößerung von Referenztypen** (später) aus S_i entstehen kann, oder

$T_i = S_i$ gilt und

für mindestens ein i_0 (mit $1 \leq i_0 \leq n$) $T_{i_0} \neq S_{i_0}$ gilt.

- Eine Methodendeklaration ist für einen Methodenaufruf **maximal spezifisch**, falls sie aufrufbar ist und keine andere aufrufbare Methode spezifischer als sie ist
- Existiert unter den aufrufbaren Methoden genau eine maximal spezifische Methode, so ist sie die **spezifischste**
- Existieren **zwei** oder **mehr** maximal spezifische Methodendeklarationen, so heißt der Methodenaufruf **mehrdeutig** und veranlasst eine entsprechende **Fehlermeldung**
- Existiert die spezifischste Methode, so wird diese aufgerufen und für die aktuellen Parameter ausgeführt



- Die Methode `main` ist nach dem gleichen Schema wie jede andere Methode aufgebaut
- Ihr Rückgabetyp ist `void`, sie liefert also kein Ergebnis zurück
- Der einzige Parameter ist ein eindimensionales Feld vom Typ String, das bisher fast immer den Namen `args` trug

```
public class Anrede {  
    public static void main(String[] args) {  
        System.out.println("Hallo, " + args[0] + "!");  
        System.out.println("Der Name " + args[1] + " gefaellt mir gut!");  
        int i=0;  
        for (String p : args) {  
            System.out.println("Die " + i + "-te Eingabe lautet: " + p);  
            i++;  
        }  
    }  
}
```

- In Programmen werden oft Methoden aus anderen Klassen benutzt
- Beispiel: die Methoden der Klasse `Math` aus V03
- Alle diese Methoden waren durch das Schlüsselwort `static` gekennzeichnet
- Dieses Schlüsselwort macht die jeweilige Methode zur **Klassenmethode**

Damit wird die Methode für die Klasse selbst und mit der Klasse auch für andere Klassen verfügbar

```
public class MeineMethoden {  
  
    public static void mal5nehmen(int n) {  
        n = n*5;  
        System.out.println("n = " + n);  
    }  
  
    public static int fakultaet(int n) {  
        if (n == 0)  
            return 1;  
        for (int i = n-1; i>0; i--)  
            n = n*i;  
        return n;  
    }  
    public static void main(String[] args){  
        int n = 7;  
        mal5nehmen(n);  
        System.out.println("n! = " + fakultaet(n));  
    }  
}
```

- Die Klasse **MeineMethoden** enthält die Klassenmethoden:
mal5nehmen, **fakultaet** und **main**
- Sollen in einem weiteren Programm die Klassenmethoden **mal5nehmen** oder **fakultaet** aufgerufen werden, so genügt es diese mit dem vorangestellten Klassennamen **MeineMethoden** aufzurufen
- Der Compiler findet diese dann im bereits compilierten Code der Klasse **meineMethoden**, und zur Laufzeit können sie dann von dort eingebunden und ausgeführt werden

```
public class TesteMethoden {  
    public static void main(String[] args) {  
        int x = 5;  
        MeineMethoden.mal5nehmen(x);  
        System.out.println(x + " ! = " + MeineMethoden.fakultaet(x));  
    }  
}
```

- Wird das Schlüsselwort **static** in der Methodendeklaration weggelassen, so wird die betreffende Methode zur **Instanzmethode**
- Sie kann dann nur mit einem Objekt der betreffenden Klasse aufgerufen werden
- Einzelheiten und Erklärungen folgen später im Teil Objektorientierung

Beispiel:

```
public class Multiplizierer {  
    public int faktor = 0;  
    public int mul(int n) {  
        return faktor * n;  
    }  
}
```

```
public class TesteMultiplizierer {  
    public static void main(String[] args) {  
        Multiplizierer m7 = new Multiplizierer();  
        Multiplizierer m8 = new Multiplizierer();  
        m7.faktor = 7;  
        m8.faktor = 8;  
        System.out.println("7*5 = " + m7.mul(5));  
        System.out.println("8*5 = " + m8.mul(5));  
    }  
}
```

- Weitere **Einzelheiten und Erklärungen** folgen **später** in den Teilen zur Objektorientierung
- Ein Beispiel sind die Methoden der Klasse **java.lang.String**, die ebenfalls **später** erklärt werden

- **Iteration**

- Verfahren, das ein Problem schrittweise durch wiederholte Anwendung desselben Verfahrens-Schritts löst. Jeder dieser Iterationsschritte liefert Ergebnisse, die im nächsten Iterationsschritt weiter verwendet werden.
- Abbruch der Iteration, wenn die Ergebnisse eines Iterationsschrittes eine Abbruchbedingung erfüllen.
- Im Programm: **Schleife**

- **Rekursion**

- Verfahren, das ein Problem durch Zurückführen auf ein einfacheres Problem löst. Für die Lösung des einfacheren Problems greift das Verfahren auf sich selbst zurück.
- Abbruch der Rekursion, wenn das einfachere Problem gelöst werden kann, ohne dass erneut auf das Verfahren selbst zurückgegriffen werden muss.
- Im Programm: **Rekursive Methode**

Iterativ

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ 1 \cdot 2 \cdot \dots \cdot n & \text{für } n > 0 \end{cases}$$

```
static long fakul(long n) {  
    long f = 1;  
    for (long i=1; i<=n; i++)  
        f = f * i;  
    return f;  
}
```

Rekursiv

$$n! = \begin{cases} 1 & \text{für } n = 0 \\ n \cdot (n - 1)! & \text{für } n > 0 \end{cases}$$

```
static long fakul(long n) {  
    if (n == 0)  
        return 1;  
    else  
        return n * fakul(n-1);  
}
```

- **Iterativer Algorithmus**
 - Bei der Ausführung werden bestimmte Teile des Algorithmus mehrfach ausgeführt
- **Rekursiver Algorithmus**
 - Bei der Ausführung wird in bestimmten Teilen des Algorithmus der Algorithmus selbst direkt oder indirekt aufgerufen
 - **Vorteil** von rekursiven Algorithmen: Elegant und kurz für viele Problemstellungen
 - **Nachteil** von rekursiven Algorithmen: Manchmal ineffizient
- *Die rekursive Formulierung kann immer in eine äquivalente iterative Formulierung umgewandelt werden*
 - Explizite Auswertung und Verwaltung des Aufruf-Stacks (siehe Folgefolien)

- Bei der Ausführung der Methode wird die Methode selbst direkt oder indirekt wieder aufgerufen.

direkt: Im Rumpf der Methode **m** steht ein Aufruf der Methode **m**

Indirekt: Im Rumpf der Methode **m** steht ein Aufruf der Methode **a**

 Im Rumpf der Methode **a** steht der Aufruf der Methode **b**

 Im Rumpf der Methode **b** steht der Aufruf der Methode ...

...

 Im Rumpf der Methode .. steht der Aufruf der Methode **m**

- Weiteres Beispiel: Einfache Potenzfunktion (vgl. **Math.pow**)

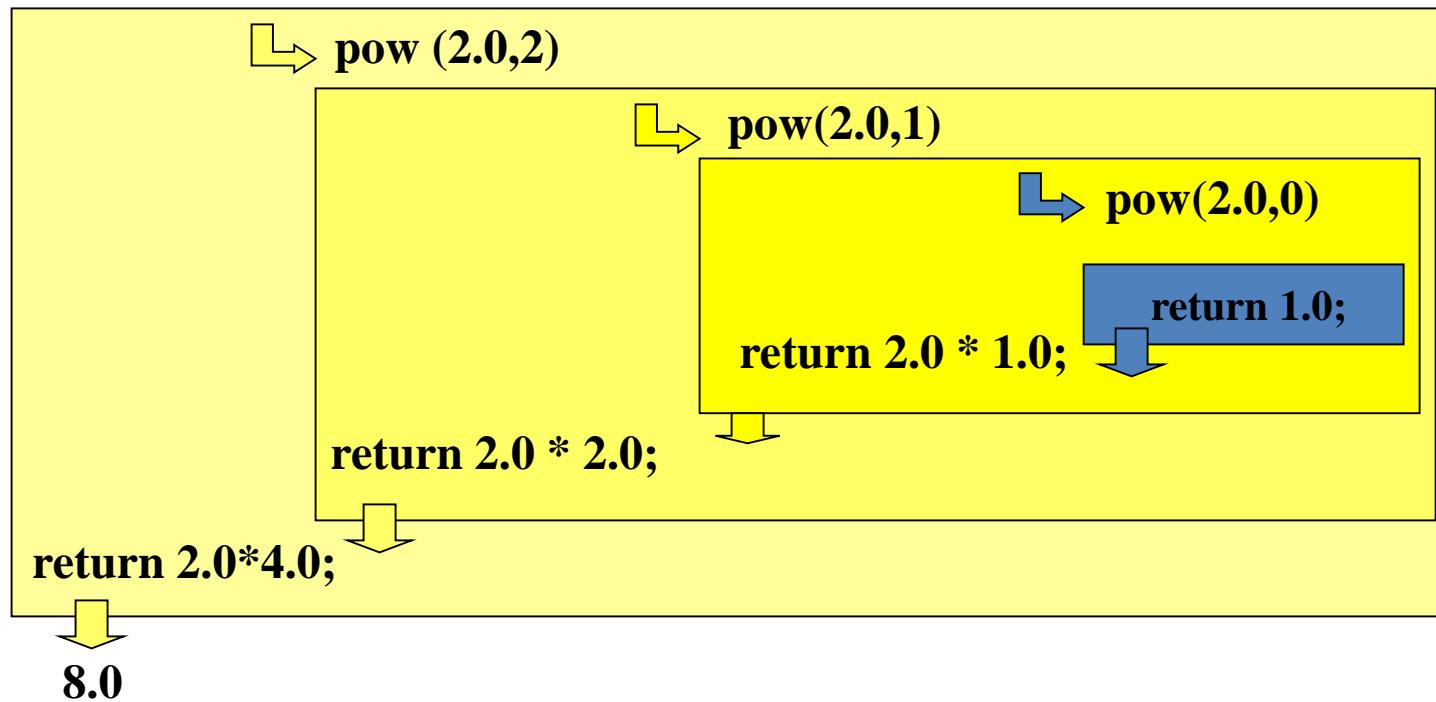
$$x^k = \begin{cases} 1 & \text{für } k = 0 \\ x \cdot x^{k-1} & \text{für } k > 0 \\ 1 / x^{-k} & \text{für } k < 0 \end{cases}$$

```
static double pow(double x, int k) {  
    if (k == 0)  
        return 1;  
    else if (k > 0)  
        return x * pow(x, k-1);  
    else  
        return 1 / pow(x, -k);  
}
```

Ablauf des Methodenaufrufs

```
static double pow(double x, int k) {  
    if (k == 0)  
        return 1;  
    else if (k > 0)  
        return x * pow(x,k-1);  
    else  
        return 1 / pow(x,-k);  
}
```

pow(2.0,3)



```
public static int summeVon1bis(int n) {  
    return n + summeVon1bis(n-1);  
}
```

Laufzeitfehler: Endlos-Rekursion!

- Wichtig!
 - Rekursionen müssen abbrechen
 - Ein oder mehrere einfache Fälle müssen gesondert behandelt werden
- Grundregel bei der Implementierung
 - Erst die nichtrekursiven einfachen Fälle implementieren
 - Danach erst den Rekursionsschritt programmieren

- Fibonacci-Zahlen und Kaninchenvermehrung

- Leonardo Pisano **Fibonacci** formulierte 1190:

Wenn ein neugeborenes Kaninchenpaar nach 2 Monaten ein neues Kaninchenpaar wirft und dann monatlich jeweils ein weiteres Paar und außerdem jedes neugeborene Paar sich auf die gleiche Art vermehrt, wie viele Kaninchenpaare gibt es dann nach n Monaten, wenn keines der Kaninchen vorher stirbt?

- Anzahl Kaninchenpaare nach n Monaten: **fib(n)**

```
fib(0) = 1
```

```
fib(1) = 1
```

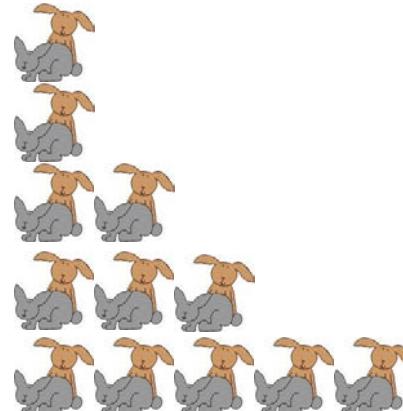
```
fib(2) = 1 + 1 = 2
```

```
fib(3) = 2 + 1 = 3
```

```
fib(4) = 3 + 2 = 5
```

```
...
```

```
fib(n) = fib(n - 1) + fib(n - 2)
```



- Rekursive Fibonacci-Methode (sehr ineffizient!)

```
static long fib(long n) {  
    if(n == 0)  
        return 1;  
    else if(n == 1)  
        return 1;  
    else  
        return fib(n-1) + fib(n-2);  
}
```

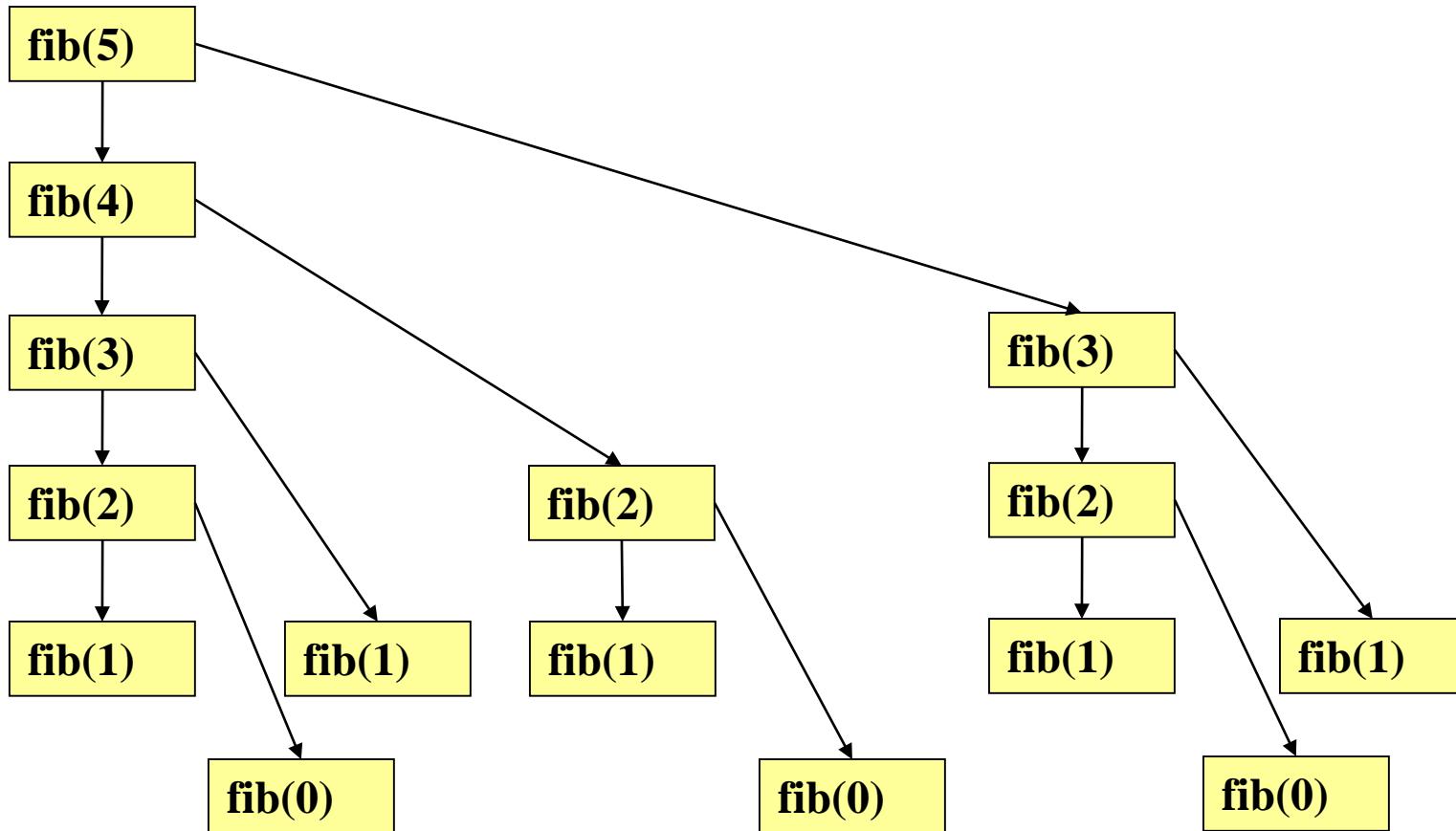
n = 40
rekursiv fib(40) = 165580141
Laufzeit: 3157 ms

- Iterative Fibonacci-Methode

```
static long fib(long n) {  
    long fN = 1, fNminus1 = 0, fNminus2;  
    for(long i=0; i<n; i++) {  
        fNminus2 = fNminus1;  
        fNminus1 = fN;  
        fN = fNminus1 + fNminus2;  
    }  
    return fN;  
}
```

n = 40
iterativ fib(40) = 165580141
Laufzeit: 0 ms

Ablauf des rekursiven Methodenaufrufs am Beispiel `fib(5)`



```
löse rekursiv ( Problem ) {  
    if (Lösung des Problems trivial) { return Lösung }  
    else {  
        zerlege Problem in Teilprobleme;  
        löse rekursiv ( Teilprobleme );  
        setze die Lösungen der Teilprobleme  
        zu einer Lösung des Gesamtproblems zusammen;  
    }  
}
```

Übung: Was berechnet die folgende Methode?

```
// b muss positiv sein
// sonst unendliche Rekursion

public static int problem(int a, int b) {
    if (b == 1) return a;
    else return a + problem(a, b-1);
}
```

- Durch Methoden wird ausführbarer Code unter einem Namen zusammengefasst
- Dieser Code kann unter Verwendung von Parametern formuliert sein, denen später beim Aufruf der Methode Werte übergeben werden
- Die Übergabe der Parameter ist **Pass-by-Value / Call-by-Value**
- Methoden erlauben die Modularisierung des Programms durch die Abtrennung von Teilaufgaben, die in eigenständigen Einheiten zusammengefasst werden
- Um die Wiederverwendbarkeit (**reusability**) zu unterstützen sollte jede Methode möglichst nur eine einzelne, wohldefinierte Aufgabe haben (Vermeidung von Nebeneffekten) und einen Namen tragen, der die Aufgabe zum Ausdruck bringt
- Wenn dies nicht geht, so sollte man überlegen, ob man die Methode nicht in kleinere, überschaubarere zerlegt

- Man unterscheidet Prozeduren und Funktionen:
 - Prozeduren liefern **keinen** Rückgabewert
 - Funktionen liefern einen Rückgabewert

An jeder Stelle, an der ein bestimmter Wert verwendet werden kann, kann auch ein Methodenaufruf verwendet werden, der diesen Wert zurückliefert!

Regel 8 (der Goldenen Regeln der Code-Formatierung): Eine Methode mit einem Rückgabewert heißt Funktion. Eine Funktion besitzt stets nur **eine einzige return-Anweisung**. Diese steht in der letzten Zeile des Methodenblocks und lautet **return result;**

Methoden: Vorteile

- Unterstützung der Wiederverwendung bekannter Programmteile
- Erleichtern das Testen des Programms
- Jede Methode kann und sollte vor dem Gebrauch getestet werden
- Auch das Hauptprogramm ist testbar durch Verwendung von Testcode für die verwendeten Methoden
- Methoden können später leichter geändert werden, z.B. Ersetzung des Quelltextes im Methodenkörper durch ein effizienteres Programm
- Der definierte Zugriff auf gekapselte Daten (als **private** deklarierte Instanzvariablen) ist durch Methoden auf geregelte Weise möglich
 - **Getter-Methoden**: zum Auslesen der Werte solcher Variablen
 - **Setter-Methoden**: zum Setzen der Werte solcher Variablen
- Methoden unterstützen: **Generalisierung**, **Vererbung**, **Kapselung** und **Polymorphie** (siehe Teil Objektorientierung)

Methoden: Vorteile

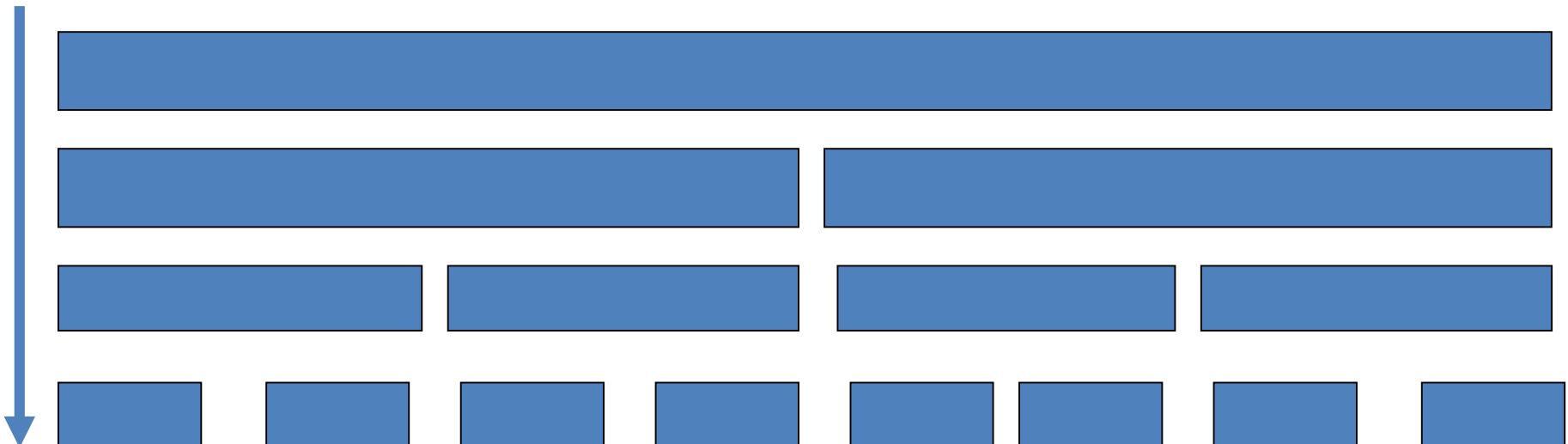
- Methoden erleichtern Rekursion und damit **Divide-and-Conquer-Verfahren** (**Teile-und-herrsche-Verfahren**) und das **Top-Down-Prinzip** (**schrittweise Verfeinerung**)
- Hierdurch wird das Paradigma der **modularen Programmierung** unterstützt

Divide-and-Conquer-Verfahren

- Bezeichnung für ein algorithmisches Lösungsverfahren, das sich in die beiden folgende Grundschritte gliedert:
 - **Divide-Schritt:** Das Problem wird in zwei oder mehr möglichst gleichgroße Teilprobleme derselben Art wie das Originalproblem aufgespalten, die unabhängig voneinander gelöst werden
 - **Conquer-Schritt:** Die Lösungen der Teilprobleme werden zu einer Lösung des Gesamtproblems zusammengefügt
- In der Praxis werden diese Schritte iteriert, was zu einem rekursiven Lösungsansatz führt.
- Typische Beispiele sind **Mergesort** und **Quicksort** (wie in Vorlesung ADS behandelt)

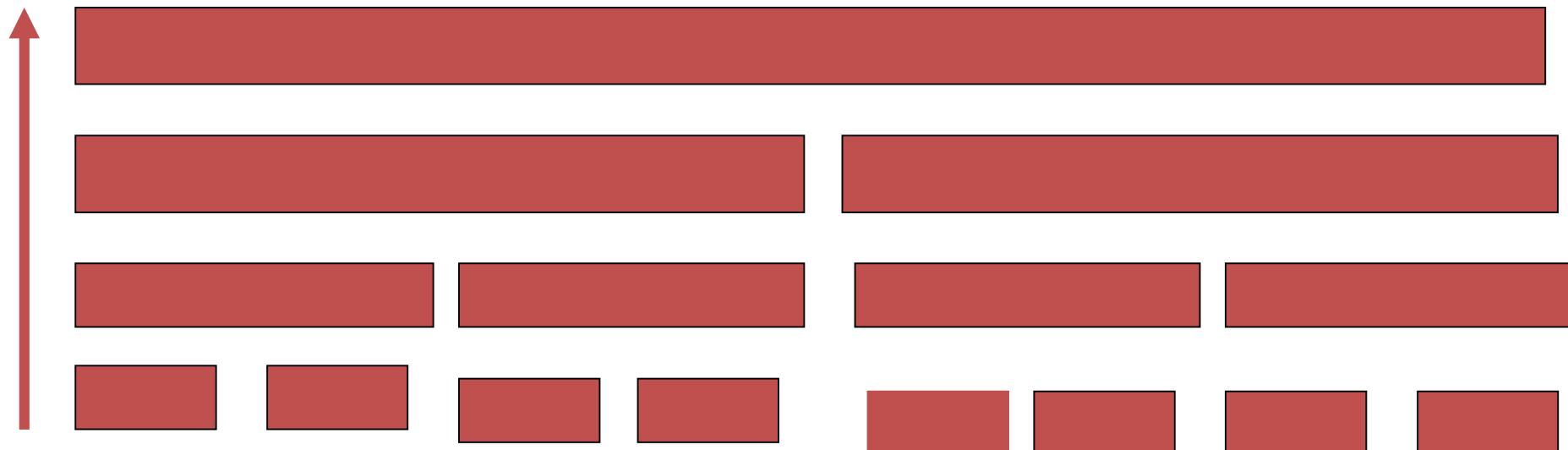
Divide-and-Conquer-Verfahren

Beispiel: Divide-Schritt



Divide-and-Conquer-Verfahren

Beispiel: Conquer-Schritt



Top-Down-Prinzip (schrittweise Verfeinerung)

■ Modulare Programmierung

- Dabei wird bei der Programmentwicklung nach dem Top-Down-Prinzip vorgegangen, indem man mit einem umfassenden abstrakten Modell beginnt und sich schrittweise verfeinernd hin zu einer konkreten Detaillierung und programmiersprachlichen Realisierung hinarbeitet

■ Top-Down-Prinzip (schrittweise Verfeinerung)

- Bei jedem Entwurfsschritt wird festgelegt, was die Untermoduln leisten sollen, nicht jedoch wie sie es leisten sollen
- Die Funktionen der Moduln jeder Ebene werden nur durch die Funktionen der Moduln der unmittelbar darunter liegenden Ebenen realisiert
- Es entsteht eine Hierarchie von Verfeinerungsstufen

Modulare Programmierung

- Dabei wird ein **Modul** in der Praxis als ein in sich zusammenhängender Baustein aufgefasst, der stets folgende Eigenschaften besitzen sollte:
 - Er ist logisch oder funktional abgeschlossen
 - Wie er arbeitet oder implementiert ist, braucht außen nicht bekannt zu sein (**information hiding**)
 - Er besitzt klar definierte Schnittstellen nach außen hin
 - Er ist überschaubar und somit leicht testbar
 - Er sollte nur möglichst wenige andere Module verwenden
- Ein **Softwaremodul** ist eine geschlossene Programmeinheit, bestehend aus Konstanten, Variablen, Datentypen und Operationen, die genau definierte Funktionen ausführen und bestimmte Eigenschaften besitzen
- Ein **System ist modular** aufgebaut, wenn es aus abgrenzbaren Einheiten zusammengesetzt ist und wenn diese Einheiten einzeln ausgetauscht, verändert oder hinzugefügt werden können, ohne dass andere Teile des Systems hierdurch beeinflusst werden oder das System arbeitsunfähig wird

Historische Anmerkungen

- 1950 – 1970 wurde Programmieren innerhalb der Informatik als „Kunst“ bezeichnet
- Übergreifende Theorien speziell zur Entwicklung großer Programmsysteme fehlten weitgehend → **Softwarekrise**
- 1970 – 1990 entstand ein systematischeres strukturiertes Vorgehen
- Prozedurale und modularen Ansätze in der Programmierung waren erfolgreiche erste Schritte zu einem stärker strukturierten Vorgehen
- 1990 entstand das **objektorientierte Paradigma** (kommt bald in VL10)

„Hello World“

Java

Klassenrumpf

In Java ist alles in Klassen organisiert

Main-Methode

Rückgabetyp: **void**
 Parameter: `String[]`
static-Methode in einer Klasse

```
/* HelloWorld.java */
public class HelloWorld {

    private static int zahl = 42;

    // Hauptprogramm
    public static void main(String[] args) {
        System.out.println("Hello " + zahl);
    }
}
```

Attribut

An die Klasse gebundene Variable

Ausgabe

Mit `System.out` und String-Konkatenation

#include-Anweisung

Vergleichbar mit Java-import

```
/* helloworld.c */
#include <stdio.h>
```

Importierte header-Datei

Header behandeln wir später im C++-Kapitel

Main-Funktion

Rückgabetyp: **int**
 Parameter: leer
 An keine Klasse gebunden!

```
/* Hauptprogramm */
int main(void) {
    printf("Hello %d\n", zahl);
}
```

Globale Variable

Von überall aus zugreifbar.
 Nicht an eine Klasse gebunden.

Ausgabe

Mit `printf` aus der importierten `stdio.h`.
 String-Konkatenation ist nicht trivial.
 Wird in der Übung besprochen.

C

- **Datentypen:** int, float, double, char, enum
- **Modifizierer, z.B.:**
 - signed: mit Vorzeichen (z.B. signed int)
 - unsigned: ohne Vorzeichen

→ *Tatsächliche Größen und Bereiche sind Plattform-abhängig!*

- **Arrays:** angezeigt mit []
- **Boolean Werte**
 - C: int = 0 für false; int ≠ 0 für true
 - C++: bool
- **Strings**
 - C: char[], endet mit '\0'
 - C++: std::string (unterstützt durch die Klasse string in der Standardbibliothek)

- Arrays fester Größe werden in C mit den Klammern nach dem Namen des Arrays definiert und initialisiert
 - Beispiel: int Array der Größe 5
 - `int foo[5];`
 - Müssen nicht extra initialisiert werden
 - Solche Arrays können nur lokal verwendet werden

Java

Deklaration

Initialisierung

```
/* Java */  
int max = 4;  
int[] feld;  
feld = new int[max];  
  
for (int i = 0; i < max; i++) {  
    feld[i] = i;  
}
```

C/C++

```
/* C */  
int max = 4;  
int feld[max];  
  
int i;  
for (i = 0; i < max; i++) {  
    feld[i] = i;  
}
```

Deklaration und Initialisierung

Laufvariable

Kann in C99 und neuer auch innerhalb der **for**-Schleife definiert werden.
In C++ auch möglich.

- Nutzer-definierte Datentypen (**derived types**)
- Ähnlich wie Klassen in Java wenn sie zur Definition von neuen Datentypen verwendet werden
- Gruppieren Variablen miteinander → Mitglieder (**members**)
- Mitglieder werden über den Punktoperator (.) angesprochen
- union ist ähnlich wie struct aber nutzt gemeinsamen Speicherbereich für alle Mitglieder

```
struct myPoint {int dimX; int dimY; int dimZ;};  
  
int main() {  
    struct myPoint p;  
    p.dimX = 10;  
    p.dimY = 20;  
    p.dimZ = 30;  
    printf("%d\n", p.dimX);  
  
    return 0;  
}
```

- Nutzer-definierte Datentypen (*derived types*)
- Ähnlich wie Klassen in Java wenn sie zur Definition von neuen Datentypen verwendet werden
- Gruppieren Variablen miteinander → Mitglieder (*members*)
- Mitglieder werden über den Punktoperator (.) angesprochen
- union ist ähnlich wie struct aber nutzt gemeinsamen Speicherbereich für alle Mitglieder

```
struct myPoint {int dimX; int dimY; int dimZ;};  
  
int main() {  
    struct myPoint p;  
    p.dimX = 10;  
    p.dimY = 20;  
    p.dimZ = 30;  
    printf("%d %d %d", p.dimX, p.dimY, p.dimZ);  
    return 0;  
}
```

ACHTUNG: Keine Java-Klassen-Syntax
bei der Deklaration der Variable.
“**struct**” muss immer mitgeführt werden!

- Können Gruppen von Daten speichern
 - So wie Java-Klassen in VL 06
- Können **nicht**:
 - Methoden enthalten
 - Erben / vererbt werden (möglich für Klassen – kommt später)
 - Alles andere, was in den folgenden Vorlesungen zu Objektorientierung kommt

```
/* Java */  
public class Date {  
    public int day, month, year;  
}  
public class Main {  
    public static void main(String[] args) {  
        Date xmas;  
        xmas = new Date();  
        xmas.day = 24;  
        xmas.month = 12;  
        xmas.year = 2014;  
    }  
}
```

Deklaration

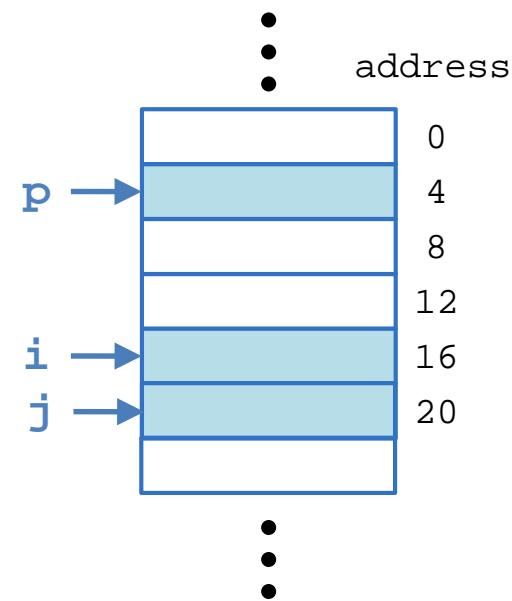
Initialisierung

```
/* C */  
struct date {int day; int month; int year;};  
  
int main(void) {  
    struct date xmas;  
    xmas.day = 24;  
    xmas.month = 12;  
    xmas.year = 2014;  
}
```

Deklaration + Initialisierung

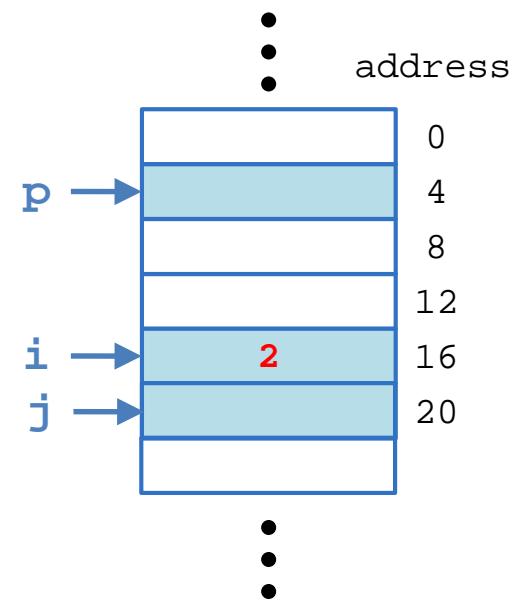
- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
 - Pointer Deklaration: Sternchen (*)
 - Auflösung der Adresse einer Variable: Referenzoperator (&)
 - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (*)

```
int *p;          /* Pointer auf int */  
int i, j;  
  
int main() {  
    i = 2;  
    p = &i;        /* p zeigt auf die Adresse von i */  
    j = *p;        /* j wird der Wert von i zugewiesen */  
    return 0;  
}
```



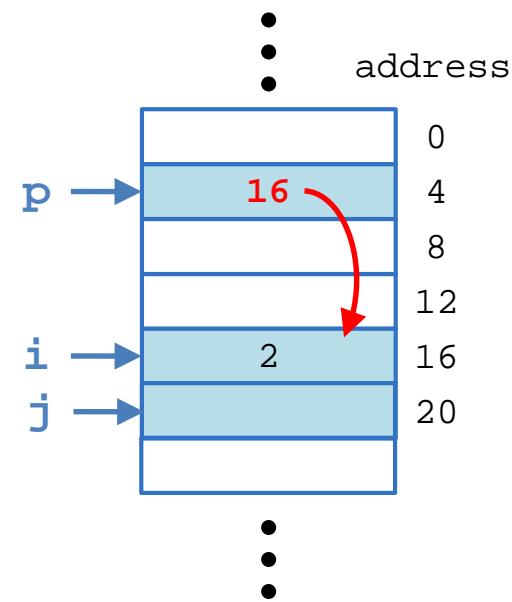
- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
 - Pointer Deklaration: Sternchen (*)
 - Auflösung der Adresse einer Variable: Referenzoperator (&)
 - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (*)

```
int *p;          /* Pointer auf int */  
int i, j;  
  
int main() {  
    i = 2;  
    p = &i;        /* p zeigt auf die Adresse von i */  
    j = *p;        /* j wird der Wert von i zugewiesen */  
    return 0;  
}
```



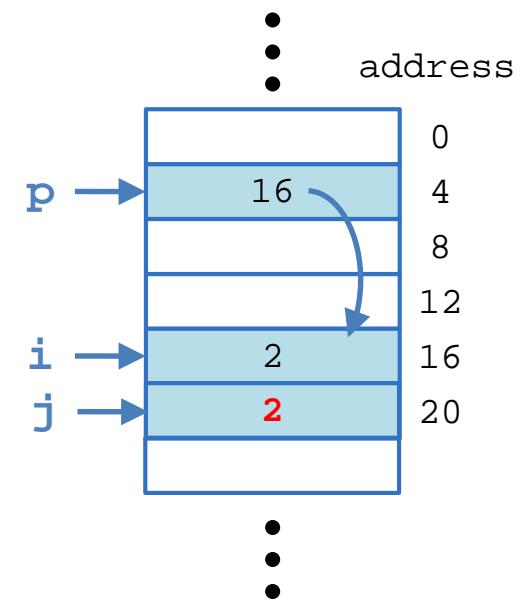
- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
 - Pointer Deklaration: Sternchen (*)
 - Auflösung der Adresse einer Variable: Referenzoperator (&)
 - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (*)

```
int *p;          /* Pointer auf int */  
int i, j;  
  
int main() {  
    i = 2;  
    p = &i;        /* p zeigt auf die Adresse von i */  
    j = *p;        /* j wird der Wert von i zugewiesen */  
    return 0;  
}
```



- C-Variable, die eine *Adresse* einer anderen Variable enthält
- Ähnlich zu Referenzdatentypen in Java, aber viel flexibler
- Pointer-Syntax:
 - Pointer Deklaration: Sternchen (*)
 - Auflösung der Adresse einer Variable: Referenzoperator (&)
 - Auflösung des Zielwertes eines Pointers: Dereferenzoperator (*)

```
int *p;          /* Pointer auf int */  
int i, j;  
  
int main() {  
    i = 2;  
    p = &i;        /* p zeigt auf die Adresse von i */  
    j = *p;        /* j wird der Wert von i zugewiesen */  
    return 0;  
}
```



- Werden gerne benutzt um Parameter an Funktionen zu übergeben
 - Vermeidet das Kopieren von Datenstrukturen, trotz „call by value“
 - Erlaubt Bearbeiten von Daten in einer Funktion, ohne Datenverlust nachdem die Funktion verlassen wird
- Zeigen auf...
 - ...jeden Datentyp, inklusive structs, Klassen (C++), und void
 - ...*Funktionen*
 - ...andere Pointer
- Nützlich, um mit Arrays zu arbeiten
- Können zur Erstellung und Bearbeitung von Datenstrukturen, wie verketteten Listen verwendet werden

- Pointer können *inkrementiert* und *dekrementiert* werden
 - mit der (Plattform-spezifischen) Größe ihrer Datentypen
 - dadurch kann man sich im Hauptspeicher nach vorne und nach hinten bewegen
 - man bewegt sich in Schritten entsprechend der Größe des Datentyps des Pointers
- *Achtung: Mögliche Datenfehler auf Grund von direktem Speicherzugriff!*

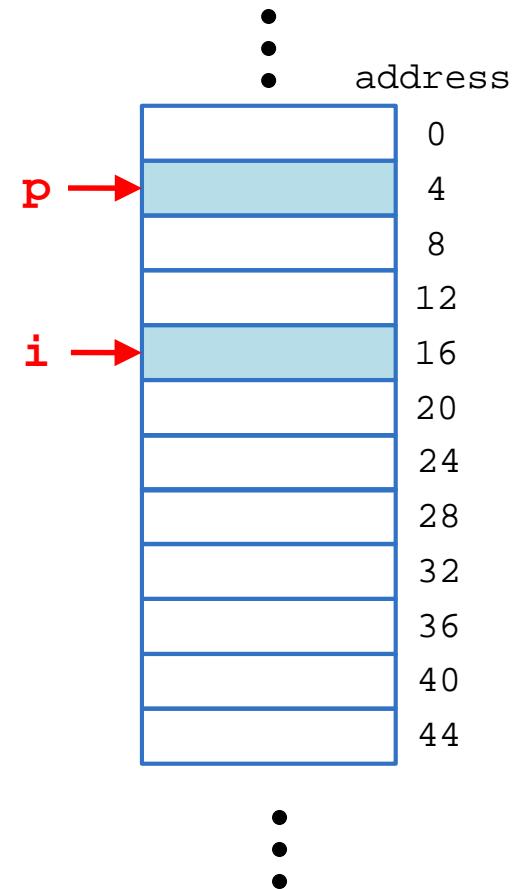
```
char c1;
char *p;      /* pointer auf char */

int main() {
    c1 = 'A';
    p = &c1;      /* p wird die Adresse von Variable c1 zugewiesen */
    p = p + 5;    /* p wird um 5 * sizeof(char) inkrementiert, d.h. 5 chars nach vorne */
    p = p - 5;    /* p wird um 5 * sizeof(char) dekrementiert, d.h. 5 chars nach hinten */
    p++;          /* p wird um 1 * sizeof(char) inkrementiert, d.h. 1 char nach vorne */
    p--;          /* p wird um 1 * sizeof(char) dekrementiert, d.h. 1 char nach hinten */
    p += 10;       /* p wird um 10 * sizeof(char) inkrementiert, d.h. 10 chars nach vorne */
    p -= 10;       /* p wird um 10 * sizeof(char) dekrementiert, d.h. 10 chars nach hinten */

    return 0;
}
```

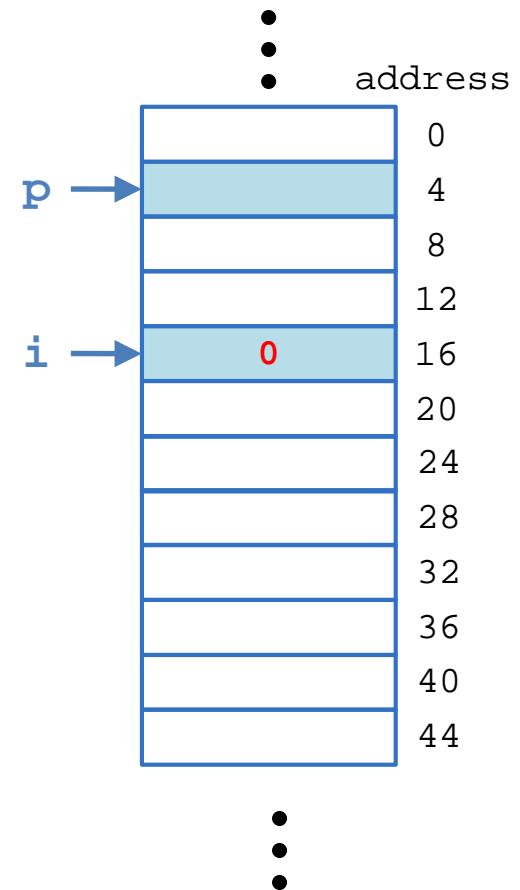
Beispiel: Pointer-Arithmetik

```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i;  
    p = p + 2;  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```

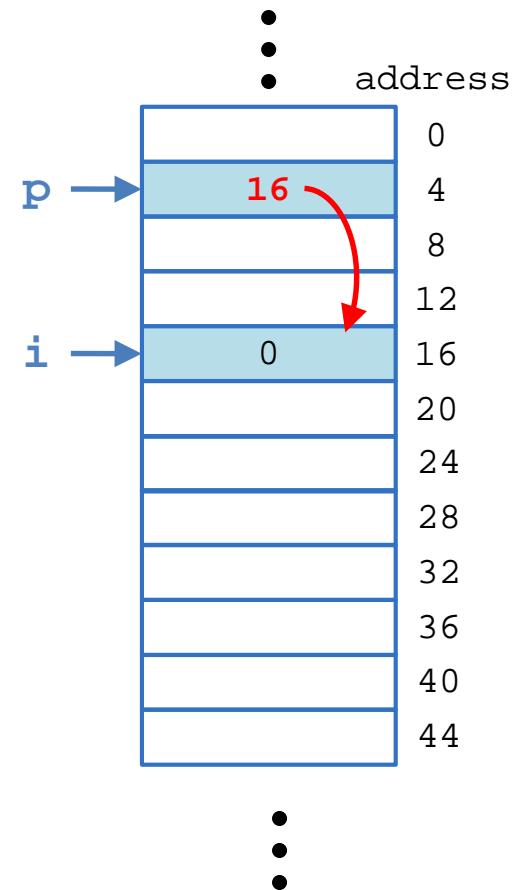


Beispiel: Pointer-Arithmetik (2)

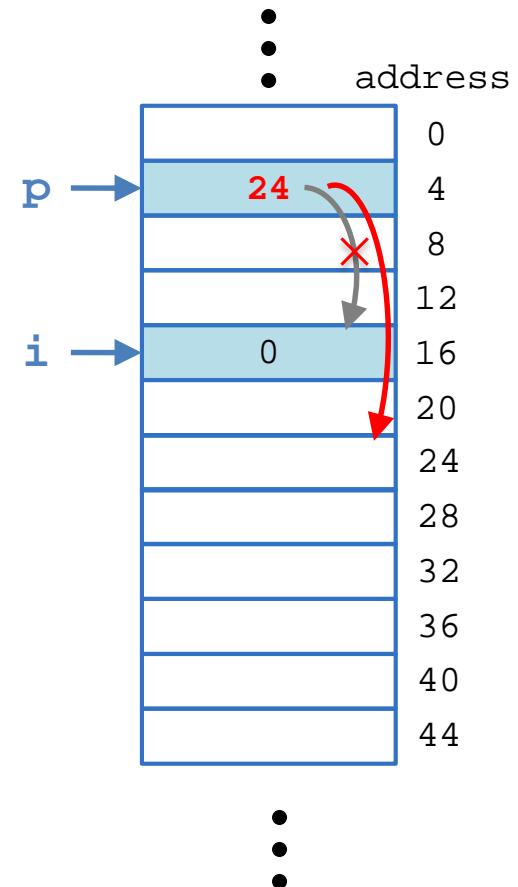
```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i;  
    p = p + 2;  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i; /* p zeigt auf die Adresse von i */  
    p = p + 2;  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



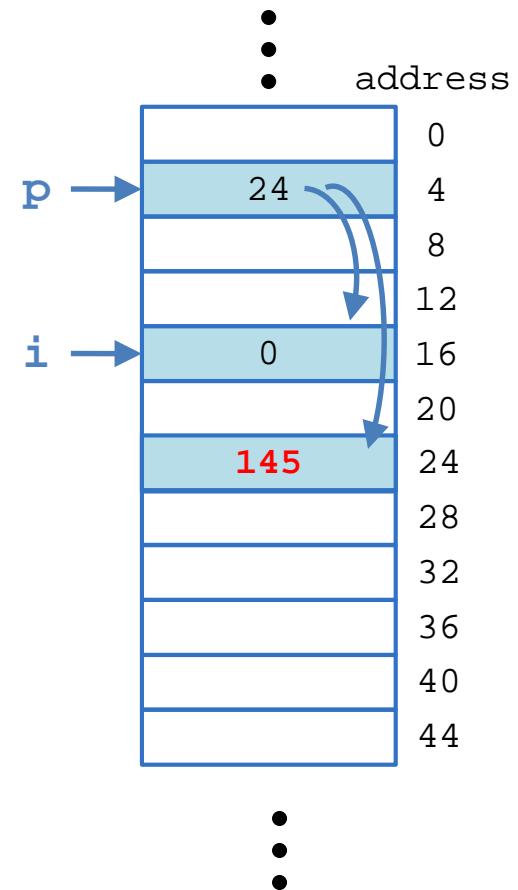
```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i; /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



Anmerkung: Ein **int** ist normalerweise 4 bytes (d.h. `sizeof(int) == 4`) → Zwischen Adresse **p** und Adresse **(p+2)** können genau 2 **int** Werte gespeichert werden!

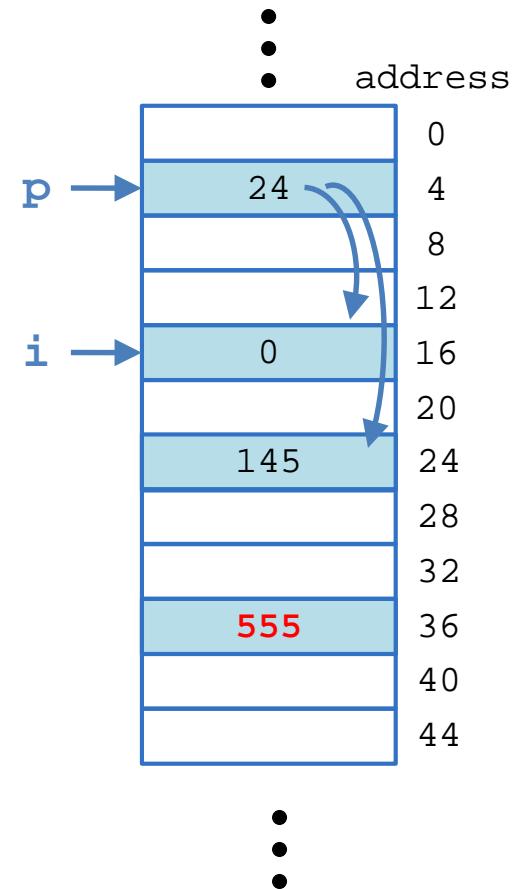
Beispiel: Pointer-Arithmetik (5)

```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i; /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



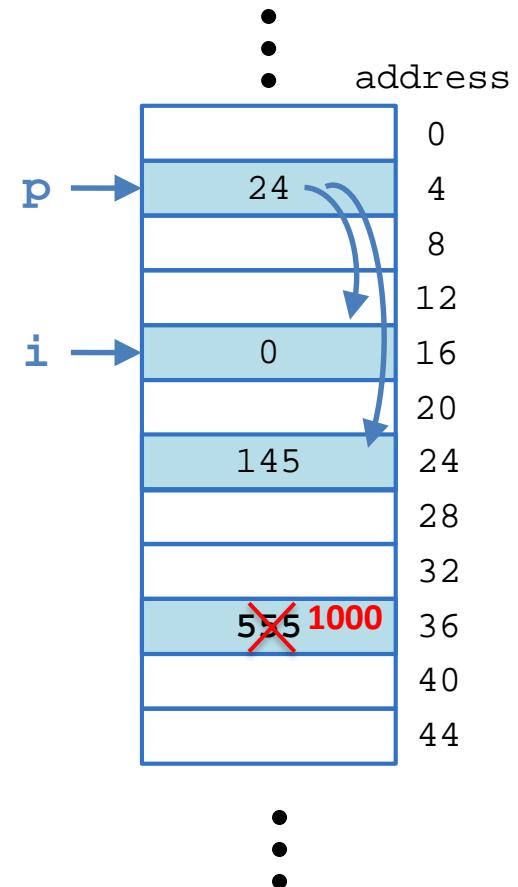
Beispiel: Pointer-Arithmetik (6)

```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i; /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



Anmerkung: Die Benutzung des Dereferenzoperators * mit **(p+3)** bewirkt keine Änderung von **p**, sondern von dem **int-Wert** der auf Adresse **p+3*sizeof(int)** liegt!

```
int i;  
int *p; /* Pointer auf int */  
  
int main() {  
    i = 0;  
    p = &i; /* p zeigt auf die Adresse von i */  
    p = p + 2; /* p wird um 2 * sizeof(int) inkrementiert */  
    *p = 145;  
    *(p+3) = 555;  
    p[3] = 1000;  
    return 0;  
}
```



Anmerkung: **p[3]** ist das gleiche wie ***(p+3)**
d.h. es entspricht der Adresse **p+3*sizeof(int)**

- Man kann auf Pointer zugreifen, als wären sie Arrays

- D.h. `int array[N];` mit `int *p = &array[0];`

gilt: `p[M] == *(p+M) == array[M]` ($M < N$)

```
char c1, c2;
char *p;      /* pointer auf char */

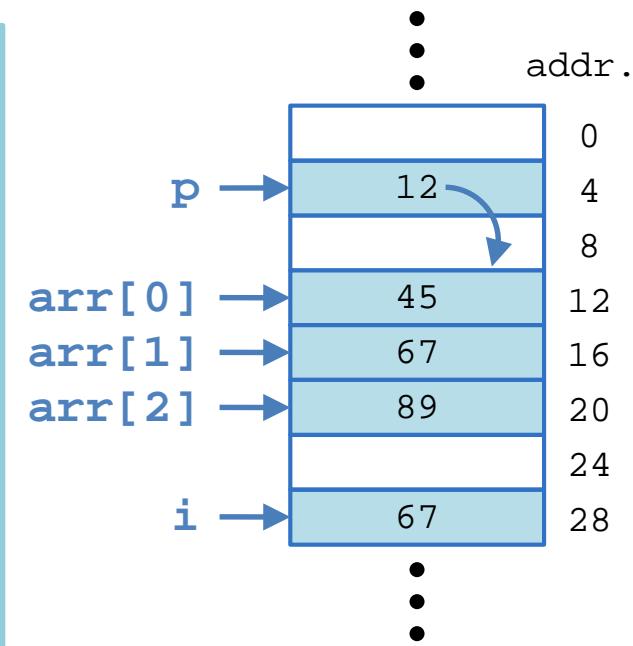
int main() {
    c1 = 'A';
    p = &c1;      /* p wird die Adresse von c1 zugewiesen */
    p += 5;       /* p wird um 5 * sizeof(char) inkrementiert */
    c2 = p[2];    /* c2 wird der Wert zugewiesen, der an der Adresse p + 2 * sizeof(char) steht */
    return 0;
}
```

- Arrays werden als zusammenhängende Speicherbereiche, in denen die Array-Elemente aufeinander folgen, benutzt
- Mit Pointer-Arithmetik können Pointer auf Adressen beliebiger Array-Elemente zeigen
- Arraynamen können auf die Adresse des ersten Array-Elements *heruntergebrochen* werden

```
int arr[] = { 45, 67, 89 };
int *p, i;

int main() {
    p = arr;          /* p is assigned the address of
                        the first element of arr */
    p = &arr[0];     /* same effect as above */

    i = arr[1];      /* i is assigned the value 67 */
    i = p[1];        /* same effect as above */
    i = *(p + 1);   /* same effect as above */
    return 0;
}
```



Default Werte

- In C werden keine Standardwerte gesetzt. Variablen werden im Speicher allokiert und erhalten den Wert, der bereits an dieser Speicherstelle eingetragen ist
 - Dies gilt auch für Pointer-Adressen

```
int main() {  
    int *p;          /* p zeigt irgendwo hin */  
    *p = 5;          /* schreibt 5 an die in p gespeicherte Adresse */  
    int i;           /* enthält den Wert, der an i's Adresse stand */  
    return 0;  
}
```

- Funktionen sind wie Methoden in Java, sie haben
 - Signaturen
 - Einen Rückgabetyp
 - Übergabeparameter
- Aber:
 - Sie existieren außerhalb von Klassen
 - C++: können auch in Klassen existieren
 - Rückgabewert und Parameter sind immer Pass-by-Value
 - C++: auch Pass-by-Reference möglich (siehe C++ Kapitel)

- **Pass-by-Value:** Alle Parameter werden bei der Übergabe kopiert
- Auch Datentypen, die in Java als “Referenzdatentypen” gelten!

C:

Nimmt eine Kopie des Datums an. Änderungen werden nicht zurückgeschrieben

C:

Das ausgegebene Datum ist nicht 24, da das Ergebnis nicht zurückgeschrieben wurde

```
/* falsch */
#include <stdio.h>

struct date {int day; int month; int year;};

void setToXMas(struct date aDate) {
    aDate.day = 24;
    aDate.month = 12;
    aDate.year = 2014;
}

int main() {
    struct date xmas;
    setToXMas(xmas);
    printf("Day: %d\n",xmas.day);
    return 0;
}
```

Java:

Es wird eine Referenz übergeben. Änderungen werden zum Aufrufer propagiert.

Java:

In Java wäre xmas.day == 24

- **Lösung:** Pointer übergeben
 - Pointer werden immer noch kopiert
 - Die Daten werden jetzt aber überschrieben

C: Pointer

Nimmt eine Kopie des Pointers an. Änderungen werden auf die Adresse des Originaldatums geschrieben

C: Pointer

Das ausgegebene Datum ist 24, da das Originaldatum überschrieben wurde

```
/* richtig */
#include <stdio.h>

struct date {int day; int month; int year;};

void setToXMas(struct date *aDate) {
    (*aDate).day = 24;
    (*aDate).month = 12;
    (*aDate).year = 2014;
}

int main() {
    struct date xmas;
    setToXMas(&xmas);
    printf("Day: %d\n",xmas.day);
    return 0;
}
```

Java:

Intern macht Java genau das gleiche, wie C mit Pointern.
“Echtes” **Pass-by-Reference** gibt es nur bei C++

- Jedes Programm wird in drei Speicherbereiche (Segmente) aufgeteilt:
 - *text segment* (oder *code segment*): Programminstruktionen und Konstanten
 - *stack segment*: Lokale Variablen und Parameter bei Funktionen
 - *heap segment*: Globale Variablen und dynamisch allokierte Speicher (im Code)
- Direkte Speichermanipulation mit Pointern ist möglich → **Mögliche Speicherfehler!**
- Explizite **dynamische Allokation** von Speicher mit
 - `void * malloc(int size)`: Allokiert `size` zusammenhängende Bytes. Gibt einen Pointer auf die Adresse des ersten Bytes zurück.
 - Größe von erforderlichem Speicher kann mit `sizeof(<Datentyp>)` ermittelt werden
 - `void free(void * ptr)`: Deallokiert den Speicherblock, der bei `ptr` beginnt.
 - Aufräumen von ungenutztem Speicher nicht vergessen!
 - Keine garbage collection (wie in Java)!

■ Stack

- Umgesetzt als *Last In First Out* (LIFO)
- Automatische Allokation / Speicherfreigabe bei Betreten / Verlassen von Funktionen und Blöcken
- **Achtung:** Rekursive Funktionsaufrufe / „große“ Funktionsparameter können einen Stack-Überlauf (overflow) verursachen!

■ Heap

- Automatische Allokation von globalen Variablen (sowie statischen Variablen → kommt später) ab ihrer Definition bis zum Ende der Programmausführung
- Explizite dynamische Allokation / Freigabe von beliebig großen Speicherblöcken auf Anfrage
- **Achtung:** Sequentielle Speicheranfragen resultieren nicht zwangsläufig in aufeinanderfolgenden Speicherblöcken

```
#include <stdlib.h>

int i;                      /* Globale Variable – Sichtbar bis zum Ende dieser Datei. Auf dem Heap allokiert. */

void incr(int *ptr) {
    int delta = 2;      /* Lokale Variable – In incr sichtbar. Auf dem Stack allokiert. */
    *ptr += delta;
}

int main() {
    double d = 3.7;          /* Lokale Variable – In main sichtbar. Auf dem Stack allokiert. */
    int *ptr;                /* Lokale Pointer-Variable – s.o. */
    ptr = (int *) malloc(sizeof(int));   /* Dynamische Speicher-Allokation auf dem Heap. */
    free(ptr);               /* Freigabe des allokierten Speichers auf dem Heap. */
    return 0;
}
```

- Prinzip der *Vorwärtsdeklaration*
 - Alle Entitäten (**Funktionen**, Variablen, Typen) müssen vor ihrer Nutzung deklariert werden
- Der selbe Bezeichner darf mehrfach deklariert werden
 - z.B., in verschiedenen Dateien
- Die tatsächliche Definition des Bezeichners geschieht an genau einem Ort
 - und kann mit der Deklaration integriert werden

- Prinzip der **Vorwärtsdeklaration**

- Problem: Zyklische Aufrufe sind unmöglich. Methodenordnung ist schwierig.
- Beispiel: Inkrementiert und verdoppelt Wert n -Mal.
Enthält zyklische Abhängigkeit.

```
int doubleValue(int a, int i) {  
    return increment(2*a, i);  
}  
int increment(int a, int i) {  
    if (i == 0) { return a; }  
    return doubleValue(a+1, i-1);  
}  
int main() {  
    int i = 5;  
    return increment(1, i);  
}
```

- Compilerausgabe:

```
main.cpp: In function 'int doubleValue(int, int)':  
main.cpp:2:27: error: 'increment' was not declared in this scope  
        return increment(2*a,i);  
                           ^
```

- Lösung: Separate Vorwärtsdeklaration

Vorwärtsdeklaration

Deklariert increment(int,int)
vor der Implementierung.

Deklarationen werden
normalerweise in
separaten **header-**
Dateien (.h) abgelegt die
mit **#include**
eingebunden werden
(zum Beispiel
#include <string.h>
bindet die string.h
header-Datei aus der C
Standard Library ein).

```
int increment(int, int);  
  
int doubleValue(int a, int i) {  
    return increment(2*a, i);  
}  
  
int increment(int a, int i) {  
    if (i == 0) { return a; }  
    return doubleValue(a+1, i-1);  
}  
  
int main() {  
    int n = 5;  
    return increment(1, i);  
}
```

**Referenzierung von
deklarerter Funktion**
doubleValue(int,int) kann
increment(int,int) referenzieren,
da es bereits deklariert wurde.

**Auflösung der separaten
Implementierung**
Als Teil des Kompiliervorgangs
ordnet der Linker die
Implementierung von
increment(int,int) der Deklaration
zu.

**ACHTUNG: Das geht auch Datei-
übergreifend!**

- Typ-Spezifizierer: void, char, short, int, long, signed, unsigned, float, double
- const
 - read-only nach Definition, ähnlich wie *final* in Java
 - Achtung: Mit Pointern und direktem Speicherzugriff können Speicherdaten auch mit Hilfe einer read-only Pointervariable überschrieben werden!

```
int i;                                /* i: normal integer variable */

const double pi = 3.14159; /* pi: read-only double */

int * const j = &i;                /* j: read-only pointer to int */
```

- Andere Modifizierer: extern, volatile, static, ...

- C Deklarationen können schwer zu lesen sein:

- Nicht einfach von links nach rechts

```
int * arr[ ]; /* arr is an array of  
               pointers to int */
```

- Möglicherweise verschachtelt

```
int *(*p)(); /* p is a pointer to a  
               function returning a  
               pointer to an int */
```

- Modifizierer wie const und volatile

```
volatile int * const i; /* i is a  
                         read-only pointer to  
                         a volatile int */
```

- What does the following declaration mean?

```
static unsigned int * const * (*next)();
```

- A Bezeichner "name" (von links nach rechts) „[name] ist ein...“
- B Präzedenzreihenfolge:
 - B.1 Klammer () „...“ gruppiert Teile einer Deklaration
 - B.2 Postfixoperatoren:
 - B.2.1 () „...Funktion mit Rückgabewert...“
 - B.2.2 [] „...Array von...“
 - B.3 Prefixoperator: * „...Pointer auf...“
 - B.4 Prefixoperator * und **const / volatile** Modifizierer: „...[modifizierter] Pointer auf...“
 - B.5 **const / volatile** Modifizierer neben Typ-Spezifikator: „...[modifizierter] [Spezifikator]“
 - B.6 Typ-Spezifikator: „...[Spezifikator]“

```
static unsigned int * const * (*next)();
```

```
static unsigned int * const * (*next)();
```

1. A

next

„next is ein ...“

```
static unsigned int * const *(*next)();
```

- | | | | |
|----|-----|------|---------------------|
| 1. | A | next | „next is ein ...“ |
| 2. | B.3 | * | „...Pointer auf...“ |
| | | | |
| | | | |
| | | | |
| | | | |
| | | | |

```
static unsigned int * const * (*next)();
```

- | | | | |
|----|-----|------|-------------------------------------|
| 1. | A | next | „next is ein ...“ |
| 2. | B.3 | * | „...Pointer auf...“ |
| 3. | B.1 | () | „...“ (Gruppierung der Deklaration) |

```
static unsigned int * const *(*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	()	„...“ (Gruppierung der Deklaration)
4.	B.2.1	()	„...eine Funktion mit Rückgabewert...“

```
static unsigned int * const *(*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	()	„...“ (Gruppierung der Deklaration)
4.	B.2.1	()	„...eine Funktion mit Rückgabewert...“
5.	B.3	*	„...Pointer auf...“

```
static unsigned int * const *(*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	()	„...“ (Gruppierung der Deklaration)
4.	B.2.1	()	„...eine Funktion mit Rückgabewert...“
5.	B.3	*	„...Pointer auf...“
6.	B.4	* const	„...einen read-only Pointer auf...“

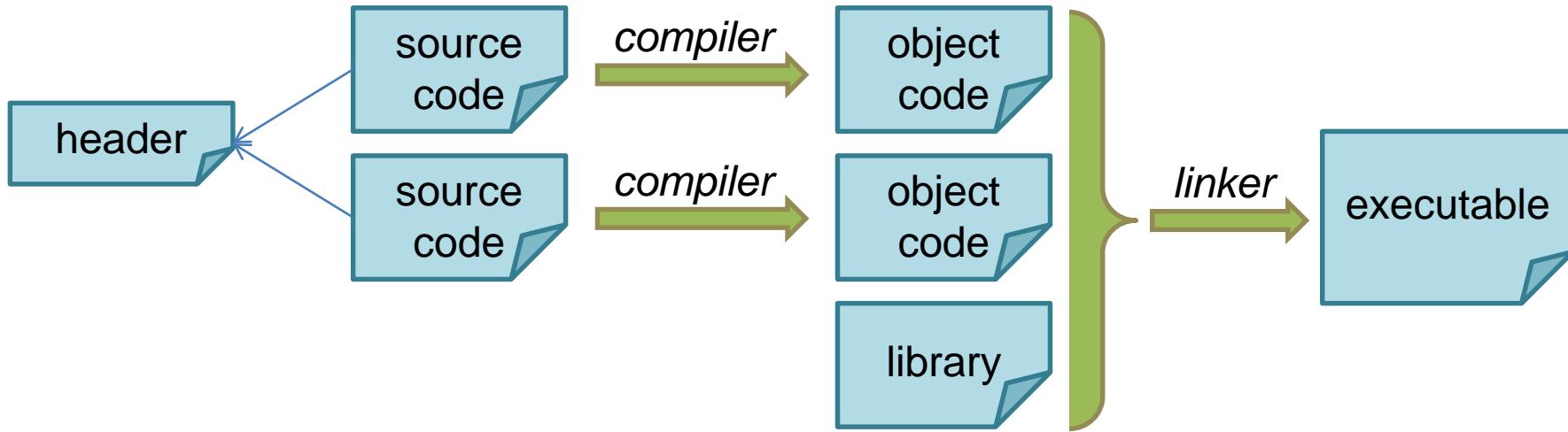
static unsigned int * const *(*next)();

1.	A	next	„next is ein ...“
2.	B.3	*	„...Pointer auf...“
3.	B.1	()	„...“ (Gruppierung der Deklaration)
4.	B.2.1	()	„...eine Funktion mit Rückgabewert...“
5.	B.3	*	„...Pointer auf...“
6.	B.4	* const	„...einen read-only Pointer auf...“
7.	B.6	static unsigned int	„...static unsigned int.“

```
static unsigned int * const *(*next)();
```

1.	A	next	„next is ein ...“
2.	B.3	*	„....Pointer auf...“
3.	B.1	()	„....“ (Gruppierung der Deklaration)
4.	B.2.1	()	„....eine Funktion mit Rückgabewert...“
5.	B.3	*	„....Pointer auf...“
6.	B.4	* const	„....einen read-only Pointer auf...“
7.	B.6	static unsigned int	„....static unsigned int.“

- Artefakte
 - Source-Code Dateien (inklusive header-Dateien)
 - Object-Code Dateien (inclusive Bibliotheken)
 - Executable
- Werkzeuge
 - Compiler
 - Linker



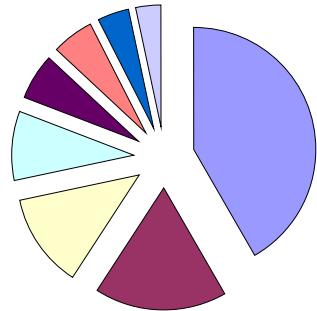
- C
 - Keine Objektorientierung, keine Klassen, nur structs
 - Programmablauf wird von einer Funktionsmenge vorgegeben
 - Ausführung beginnt in Funktion `main()`
- C++
 - Zusätzliche Objektorientierung
 - Kann immer noch Funktionen und Variablen außerhalb von Klassen verwenden
 - Kann immer noch prozedurale Programmierung ohne Objektorientierung darstellen
- Java
 - Strikte Objektorientierung (**erlaubt trotzdem primitive Datentypen**)
 - Alles muss in einer Klasse enthalten sein (auch `static void main()`)
 - Source-Code Dateien werden nach Klassengrenzen organisiert

- C immer noch eine weit verbreitete Sprache
 - Besonders für eingebettete Systeme (embedded systems) und Hochleistungsrechnen (high-performance computing, HPC)
 - Aufgrund von mehr Freiheitsgraden ([manuelle Speicherverwaltung,...](#))
- Obwohl es syntaktisch oft ähnlich zu Java ist, gibt es wichtige Unterschiede:
 - Pointer
 - Keine festen Größen für mitgelieferte Datentypen
 - keine Default-Werte

Die Treiber für die Objektorientierung (Ende der 1980er Jahre)



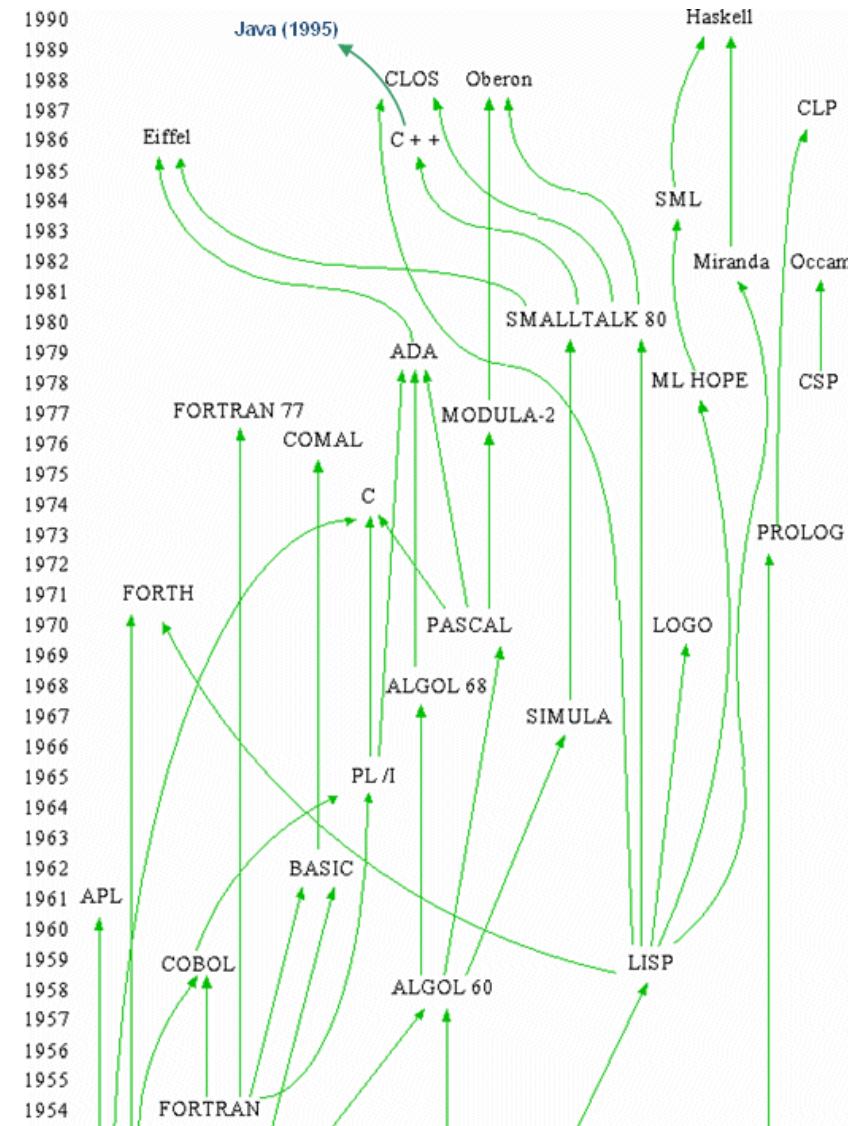
Grafische Benutzen-Oberflächen für den neuen PC-Massen-Markt (Programm-Komplexität vervielfacht sich)!



Die Kosten für die Wartung der Software sind zu hoch, Anpassungen dauern zu lange und kosten zu viel!

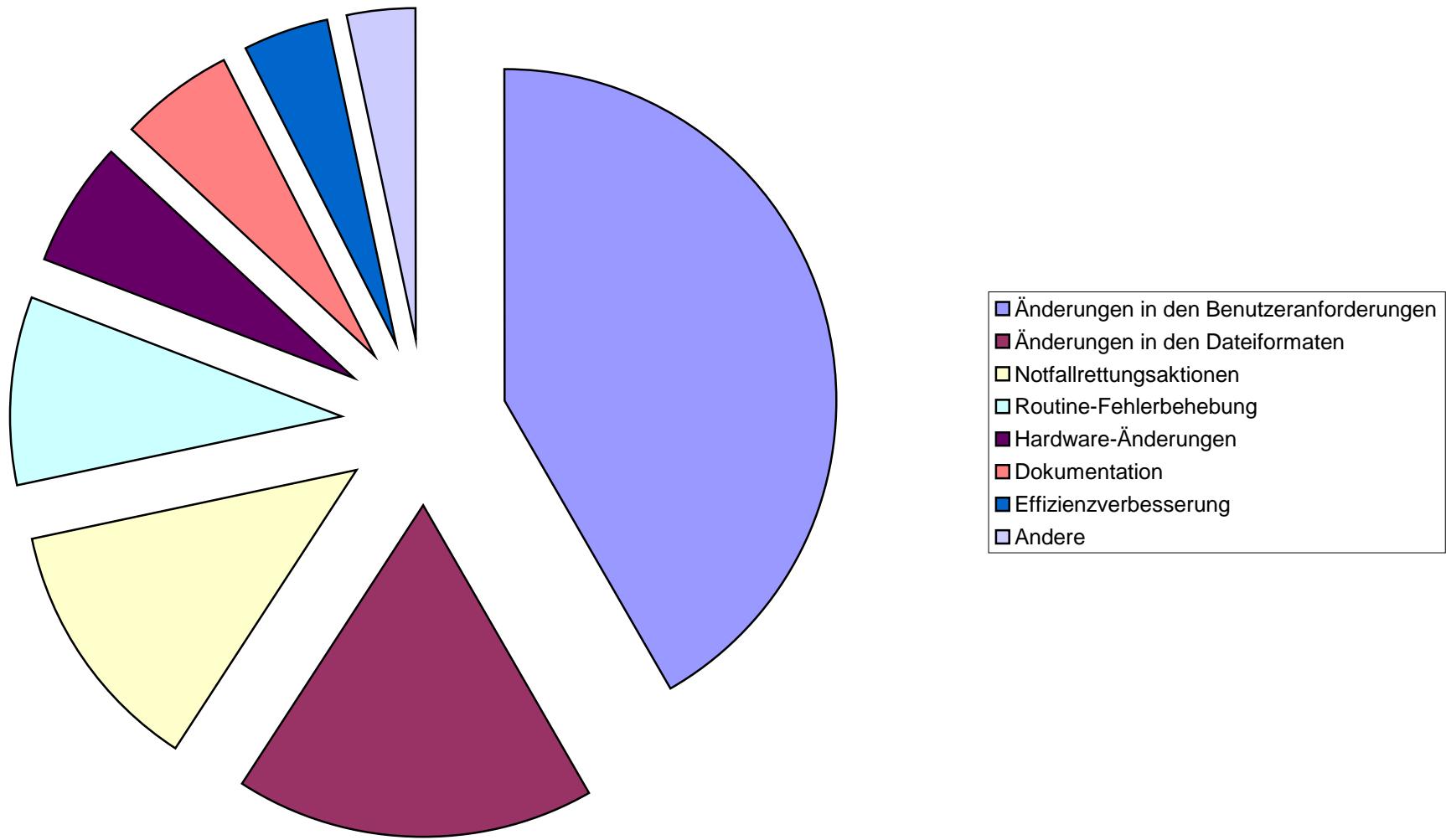


Softwarekrise führt zu immer größeren und imageschädigerenden Unglücken!



Softwarewartung

Softwarewartung machen Ende der 1980er ca. 70% der Gesamtkosten eines SW-Projekt aus



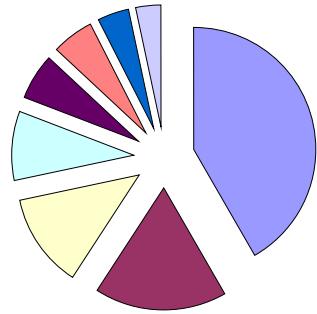
Softwarekrise

- Der Begriff der „**Softwarekrise**“ entsteht bereits in den 60gern
- Durch besonders spektakuläre Fälle erreichte dieser Begriff Ende der 70er eine breite Öffentlichkeit:
 - 1979 : **Studie zu Softwareprojekten** (USA), inges. 7 Mio USD
 - 75% der Ergebnisse nie eingesetzt
 - 19% der Ergebnisse stark überarbeitet
 - 6% benutzbar
 - 1981 : **US Air Force** Command & Control Software überschreitet Kostenvoranschlag fast um den Faktor 10 (3,2 Mio USD)
 - 1984 : **Überschwemmungskatastrophe** im französischen Tarntal, weil Steuercomputer nach Überlaufwarnung Schleusen öffnet
 - 1990 : Ausfall des **Telefonnetzes** in den USA
 - 1996 : Absturz der **Ariane 5** wegen eines Software-Fehlers

Die Treiber für die Objektorientierung (Ende der 1980er Jahre)



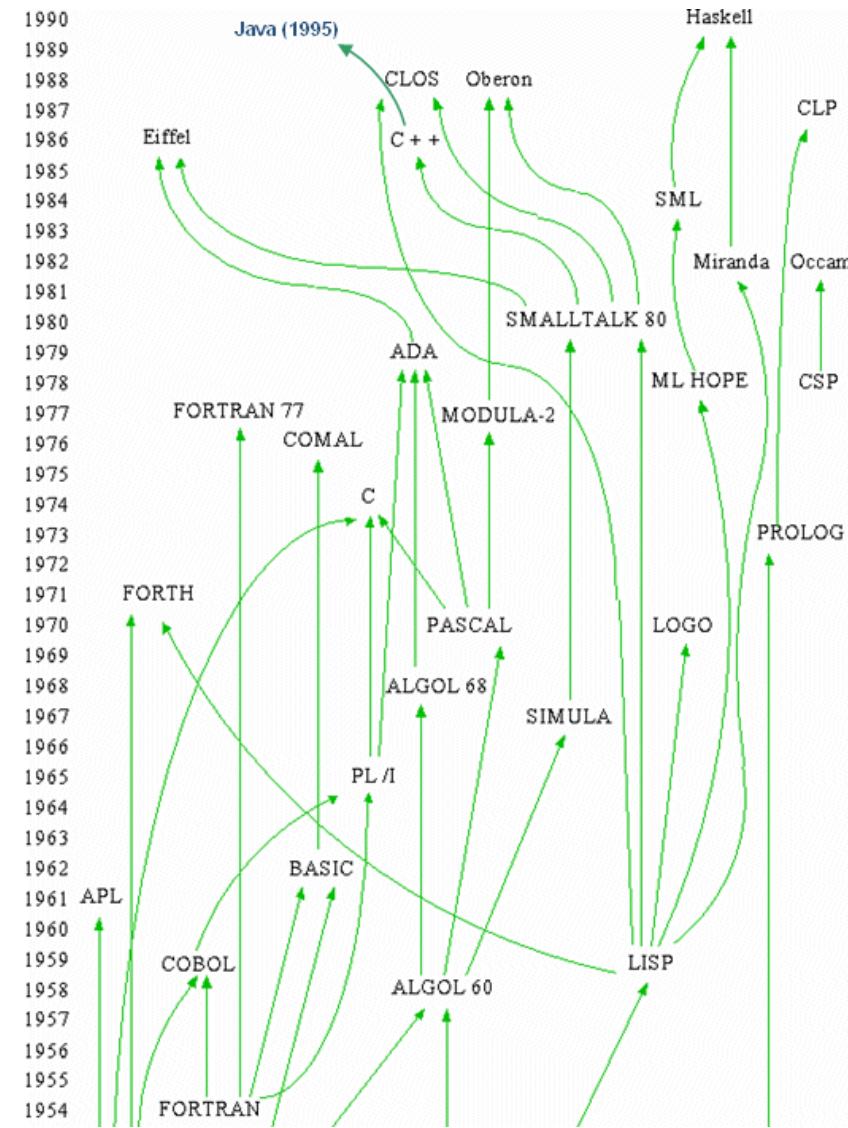
Grafische Benutzen-Oberflächen für den neuen PC-Massen-Markt (Mac OS, IBM OS/2, MS Windows)!



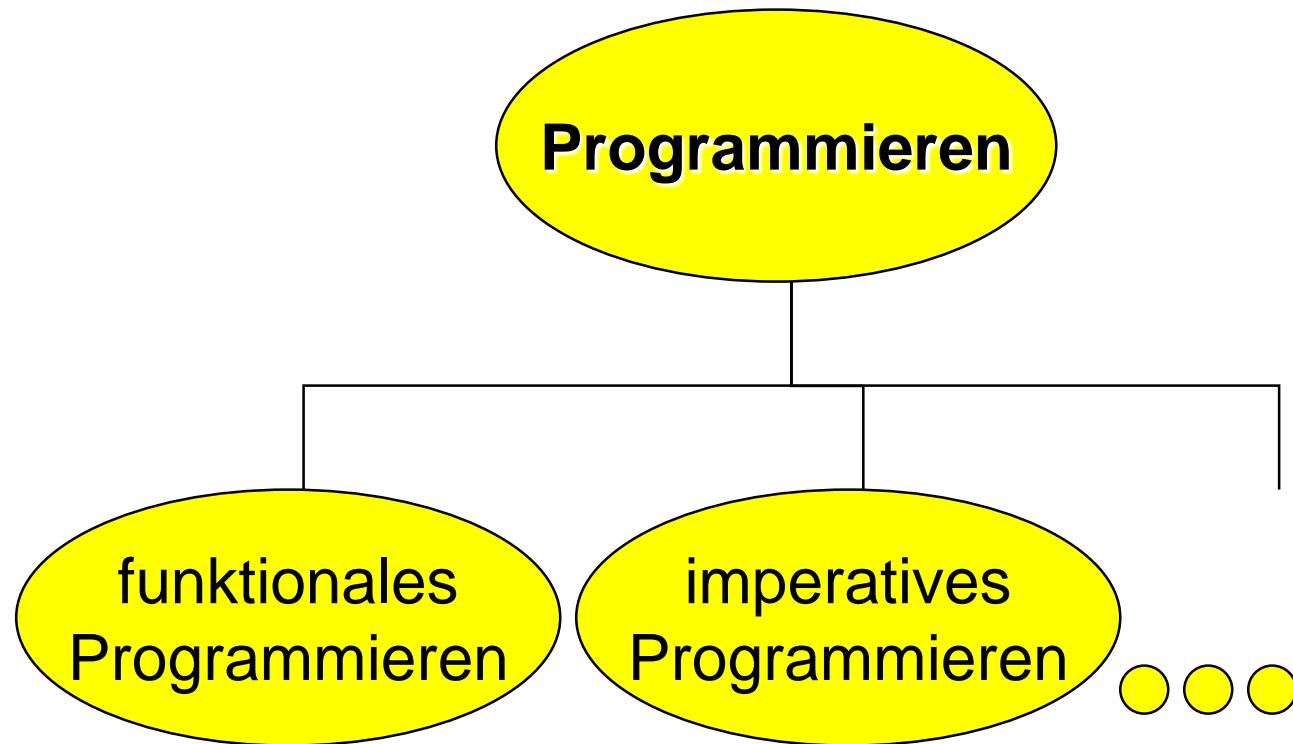
Die Kosten für die Wartung der Software sind zu hoch, Anpassungen dauern zu lange und kosten zu viel!



Softwarekrise führt zu immer größeren und imageschädigerenden Unglücken!

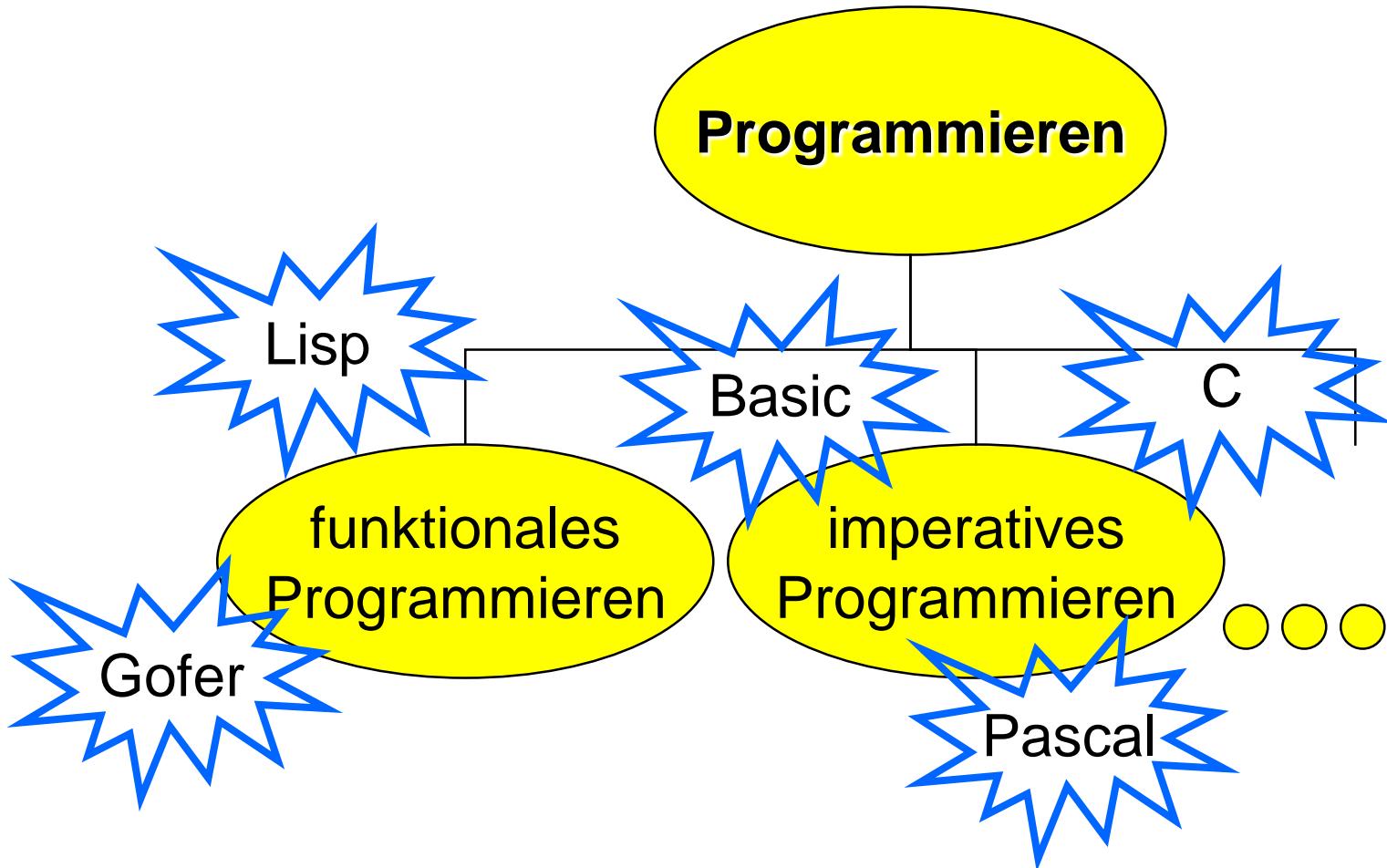


Es gibt verschiedene Philosophien, wie Programme auf dem Rechner verfasst werden sollten, sogenannte **Paradigmen**

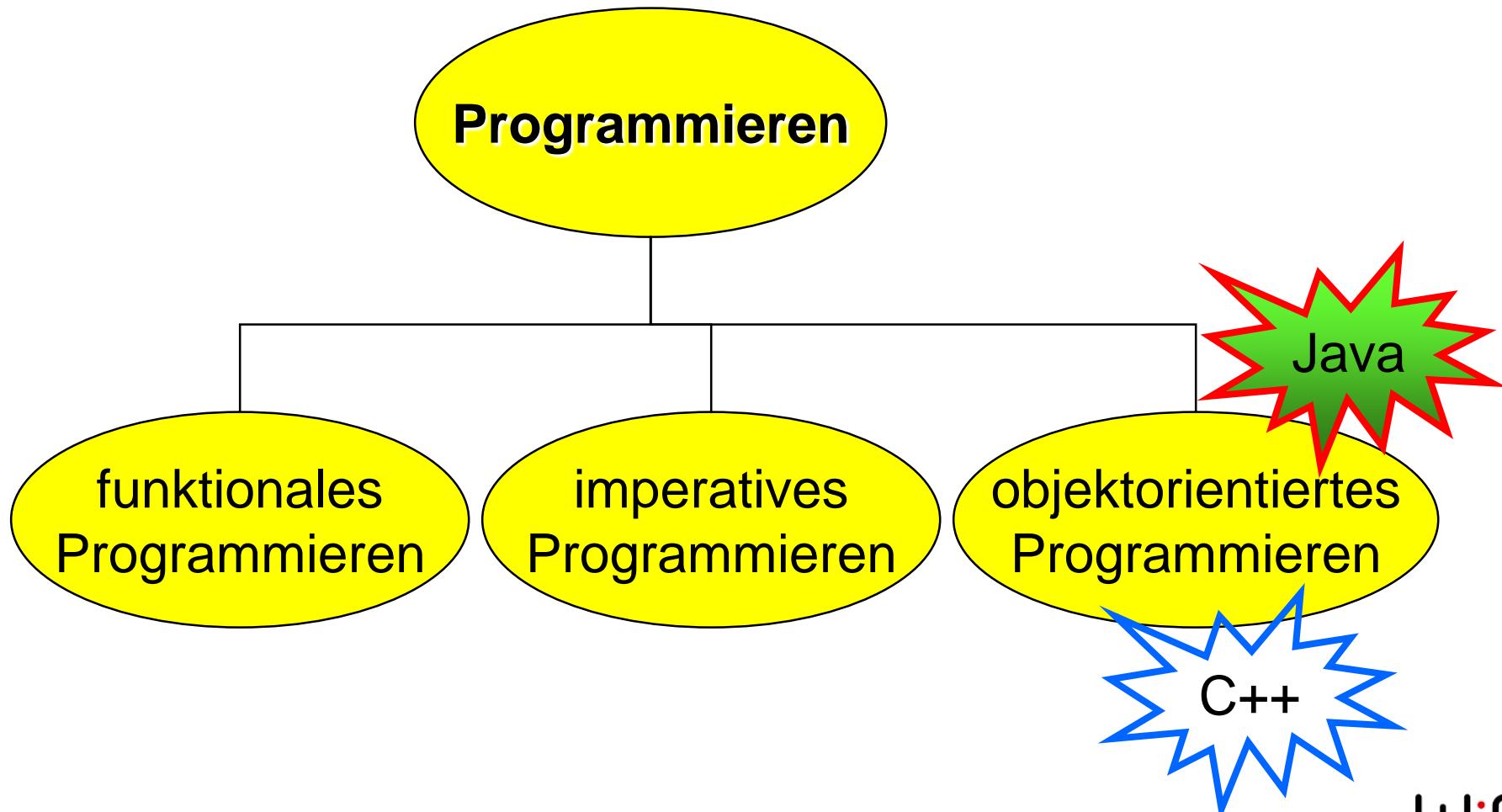


Was bedeutet „Objektorientierung“?

In der Entwicklungsgeschichte der Programmiersprachen haben sich zu den verschiedenen Paradigmen unterschiedliche Sprachen entwickelt...



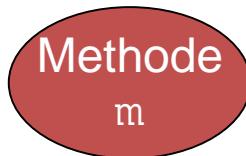
Der Programmiersprache Java liegt ein noch relativ junger Ansatz zugrunde: das **objektorientierte Programmieren**



Paradigmenwechsel

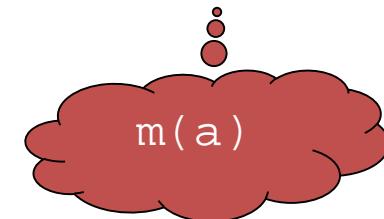
Paradigmenwechsel beim Übergang von der Dominanz der Methoden hin zu der Dominanz der Datenstrukturen

Prozedurales Programmieren



Wir realisieren
Funktionalität

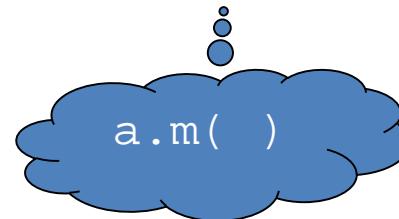
Verwendung



Datenstruktur
 a

Objektorientiertes Programmieren

Verwendung



Wir realisieren ein
Modell des Systems

Datenstruktur
 a



- Objektorientierte Programmierung geht von dem Grundsatz aus, dass ein wie auch immer geartetes Programm meist einen Teil der Welt widerspiegelt, in der wir leben
- Wenn wir also ein Modell dieser Welt auf dem Computer erzeugen wollen, sollten wir die einzelnen **Objekte**, mit denen wir im Leben konfrontiert werden, auf dem Computer abbilden können

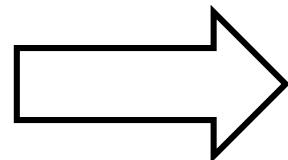


Beispiel

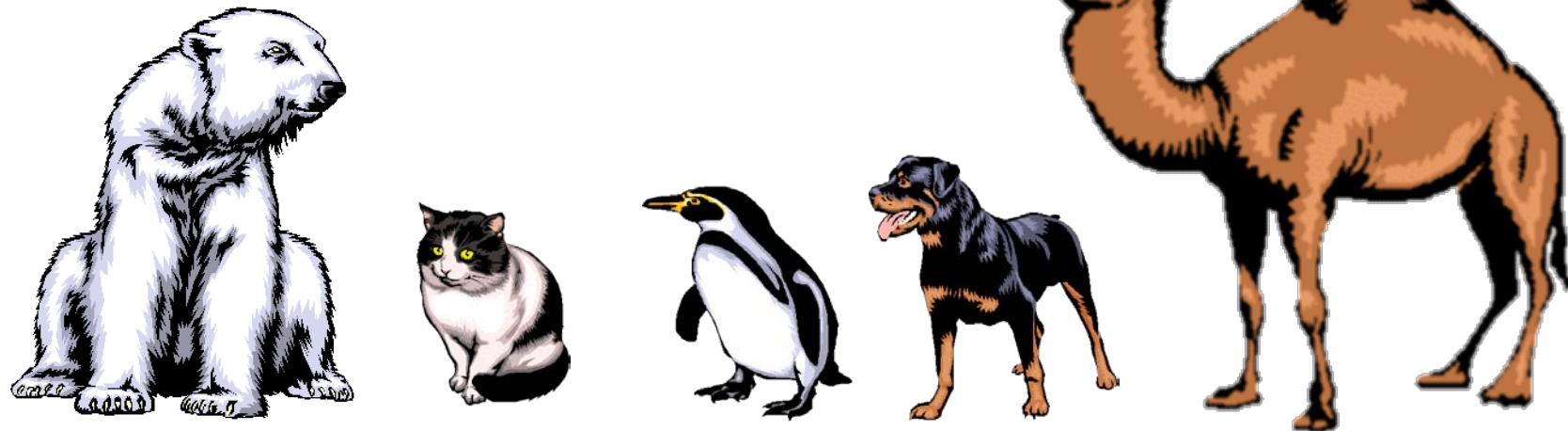
Wir wollen auf unserem Computer verschiedene Tiere „simulieren“



- Würden wir (wie bisher) imperativ programmieren, so müssten wir in Java eine Unmenge von Daten und Methoden erstellen, um das Verhalten der einzelnen Tiere abzubilden



Unser Programm würde
unübersichtlich und kompliziert!



- Mit Hilfe des objektorientierten Ansatzes umgehen wir dieses Problem, indem wir die auftretenden Objekttypen durch verschiedene **Klassen** (abstrakte Datentypen, Schlüsselwort **class**) modellieren



- Mit Hilfe des objektorientierten Ansatzes umgehen wir dieses Problem, indem wir die auftretenden Objekttypen durch verschiedene **Klassen** (abstrakte Datentypen, Schlüsselwort **class**) modellieren

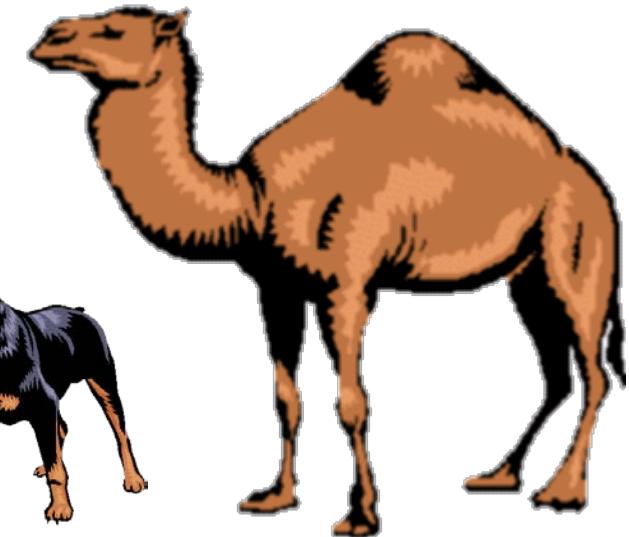
Eisbaer

Katze

Pinguin

Hund

Dromedar



- Jede dieser Klassen repräsentiert die gesammelten Eigenschaften eines Objekttyps und stellt somit sozusagen dessen „**Bauplan**“ (abstrakter Datentyp) dar

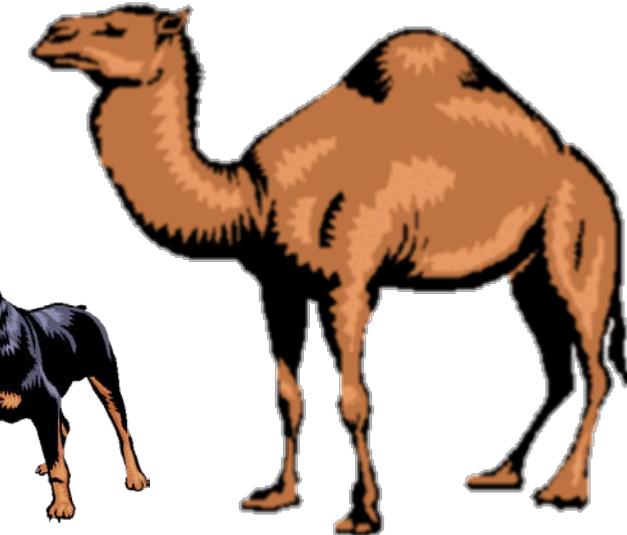
Eisbaer

Katze

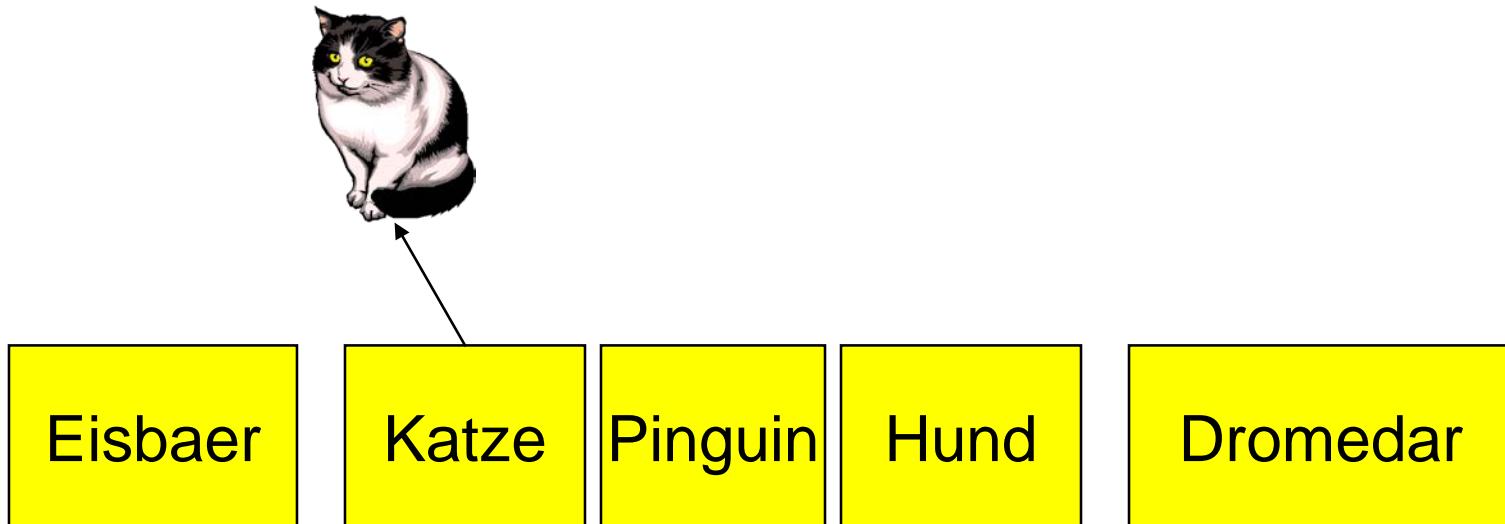
Pinguin

Hund

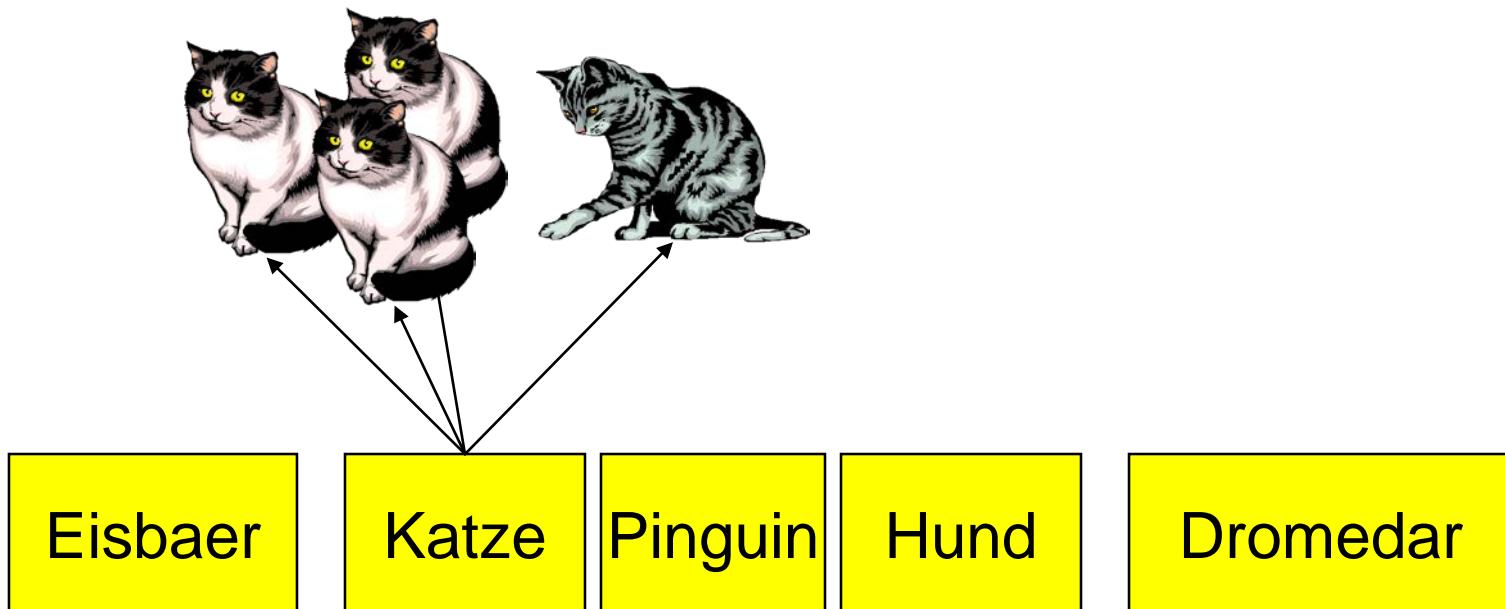
Dromedar



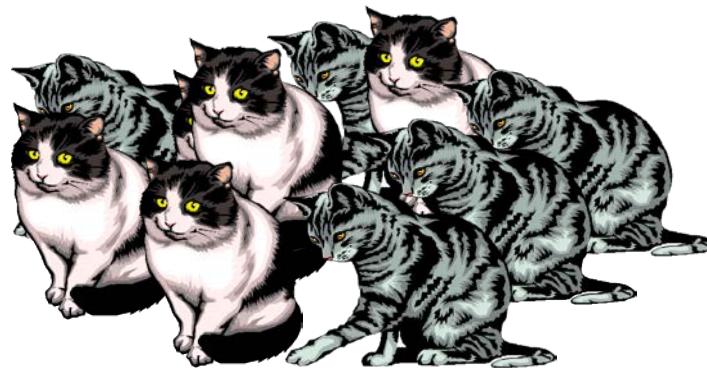
- Wollen wir nun in unserem Programm etwa ein Exemplar der Gattung „Katze“ verwenden, so können wir unsere Klassenbeschreibung benutzen, um aus ihr das gewünschte **Objekt** zu **erzeugen**
- Man bezeichnet diesen Vorgang als **Instantiierung**



- Natürlich ist es auch möglich, anhand **einer** Klassenbeschreibung **mehrere** Objekte zu erzeugen
- Wie bei verschiedenen Häusern, die nach dem selben Bauplan gebaut wurden, ist jedes Objekt individuell verschieden und kann seine eigenen Besonderheiten haben



- Egal jedoch, wie viele **Instanzen** unserer Klasse wir bilden
 - es handelt sich stets um ein und denselben Bauplan, welchen wir beliebig oft als Vorlage verwenden können



Eisbaer

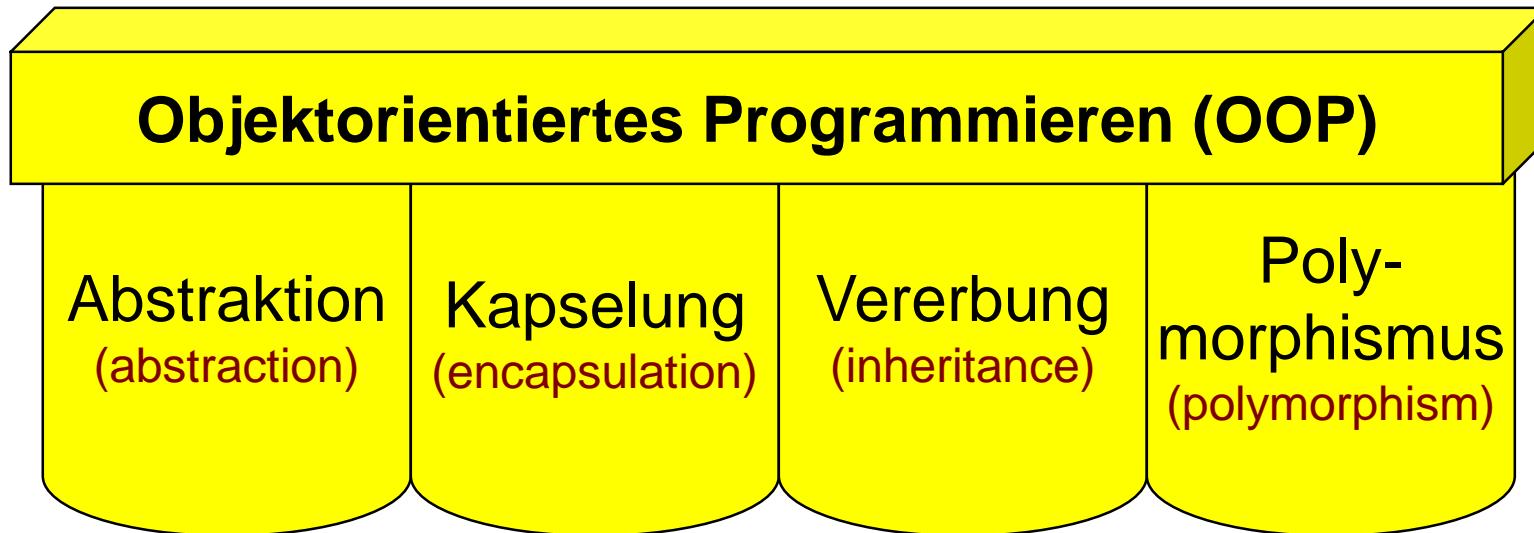
Katze

Pinguin

Hund

Dromedar

- Die objektorientierte Philosophie stützt sich allgemein auf vier grundlegende Prinzipien:





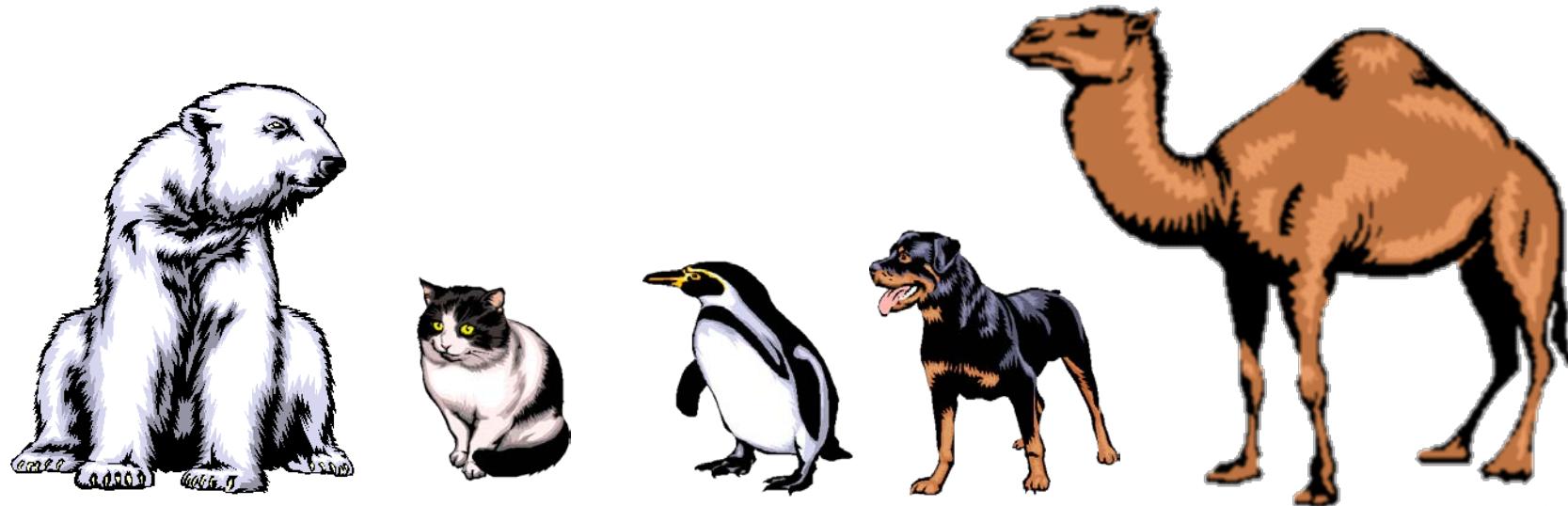
ist das Zusammenfassen mehrerer Klassen mit gemeinsamen Merkmalen zu einer allgemeineren Oberklasse

Ziel der **Abstraktion** ist es,

- eine **Hierarchie** zu erzeugen, in der Objekte wie in der wahren Welt miteinander in Beziehung stehen,
- gemeinsame **Eigenschaften** verschiedener Klassen **einheitlich** zu modellieren und somit
- **Entwicklungszeit** zu sparen und
- eine Vielzahl von **Fehlerquellen** auszuschalten

Beispiel

Wir wollen unsere fünf Tierarten in verschiedene Kategorien unterteilen



Beispiel

Wir wollen unsere fünf Tierarten in verschiedene Kategorien unterteilen

Wir wollen als erstes Kriterium unterscheiden, ob es sich bei den Tieren um **Säugetiere** oder **Vögel** handelt

Eisbaer

Katze

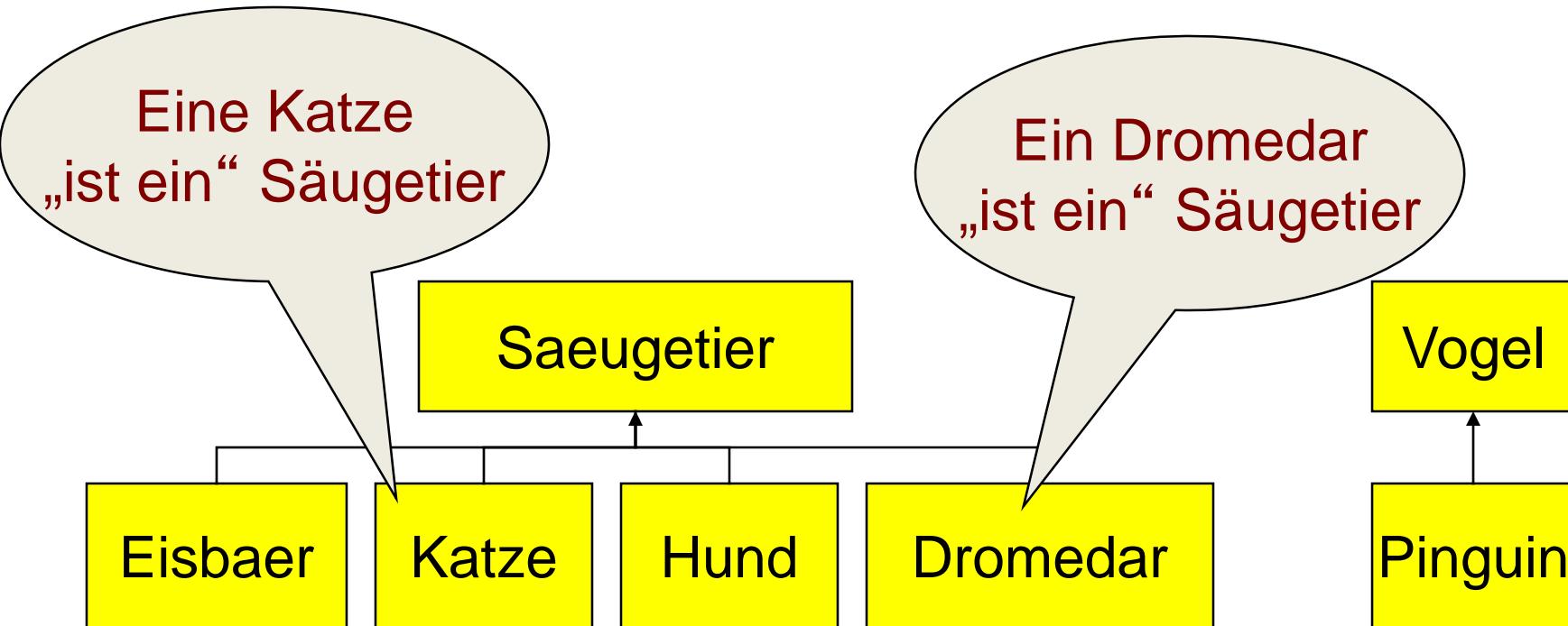
Pinguin

Hund

Dromedar

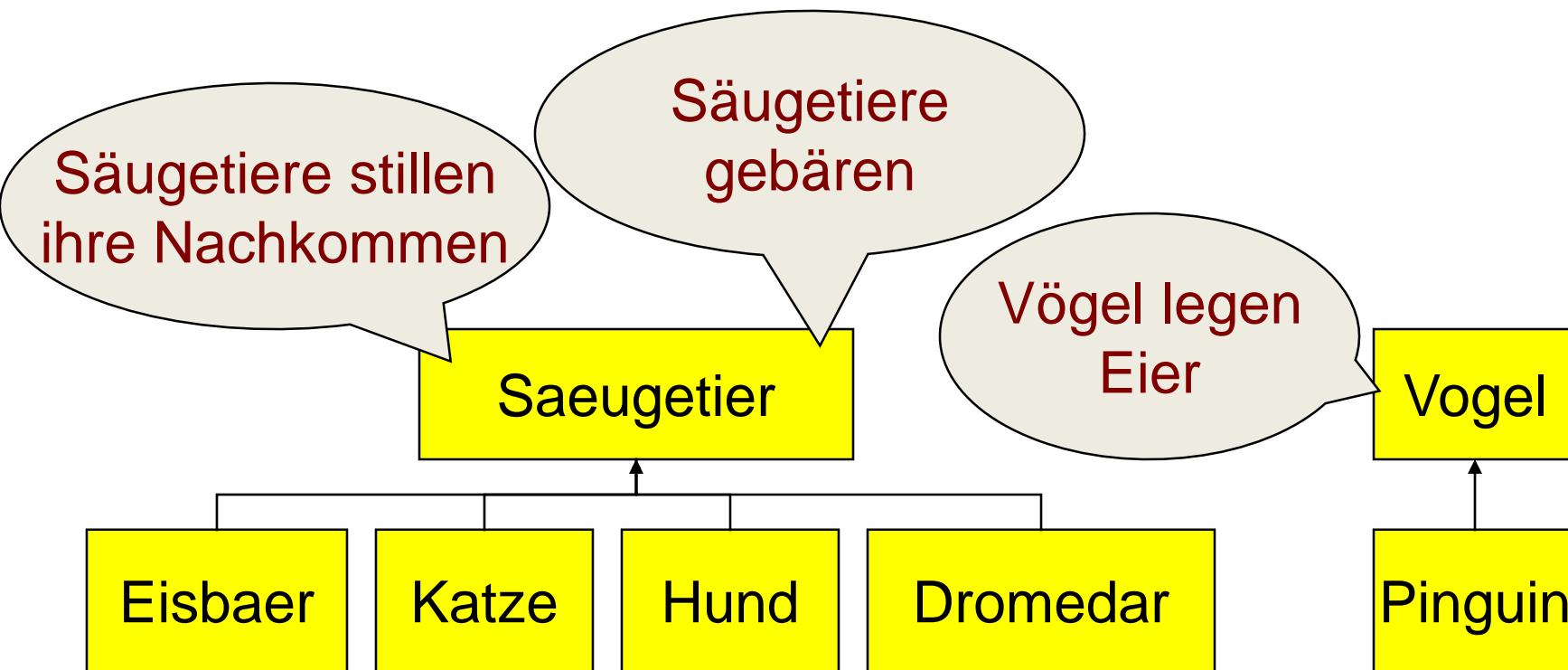
Beispiel

Bei der hier angegebenen Abstraktion (sie wird auch als **Generalisierung** bezeichnet), handelt es sich um eine „ist-ein“-Beziehung



Beispiel

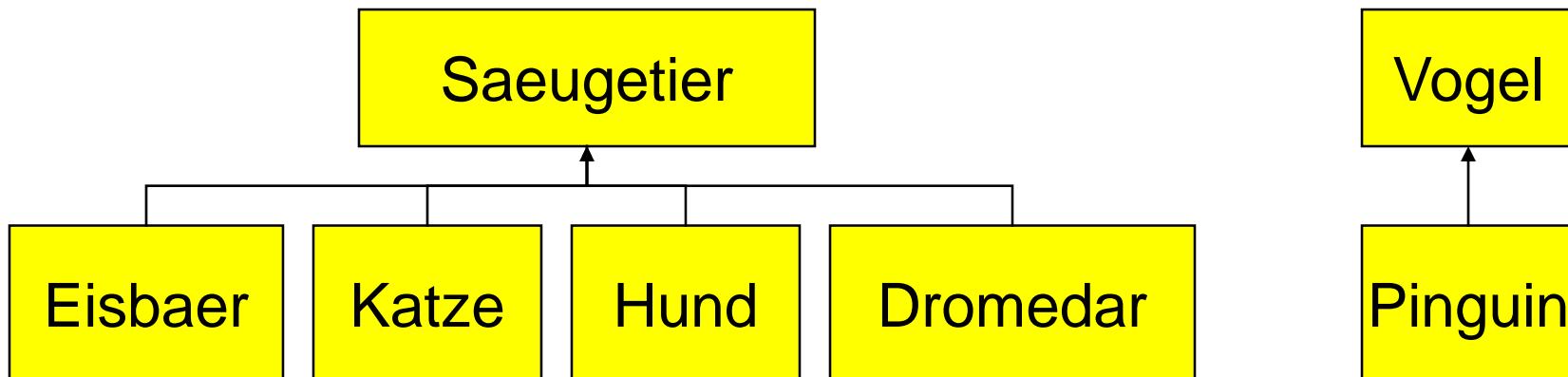
Sämtliche Tiere, für die diese „ist-ein“-Beziehung gilt, haben gewisse Eigenschaften gemeinsam



Beispiel

Sämtliche Tiere, für die diese „ist-ein“-Beziehung gilt, haben gewisse Eigenschaften gemeinsam

Anstatt also etwa den Vorgang des „Milch gebens“ für Eisbär, Katze, Hund und Dromedar einzeln zu modellieren, wird der geschickte Programmierer dies lediglich für die **Superklasse** „Säugetier“ tun

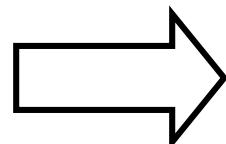


Beispiel

Sämtliche Tiere, für die diese „ist-ein“-Beziehung gilt, haben gewisse Eigenschaften gemeinsam

Anstatt also etwa den Vorgang des „Milch gebens“ für Eisbär, Katze, Hund und Dromedar einzeln zu modellieren, wird der geschickte Programmierer dies lediglich für die **Superklasse** „Säugetier“ tun

Hierdurch erhalten auch alle **Subklassen** (das sind die Klassen, die zur Superklasse in einer „ist-ein“-Beziehung stehen) automatisch die Eigenschaft, Milch geben zu können



Wir bezeichnen diesen Vorgang als **Vererbung**



ist das Weiterreichen von Eigenschaften zwischen Klassen, die in einer „ist-ein“-Beziehung stehen

Ziel der **Vererbung** ist es

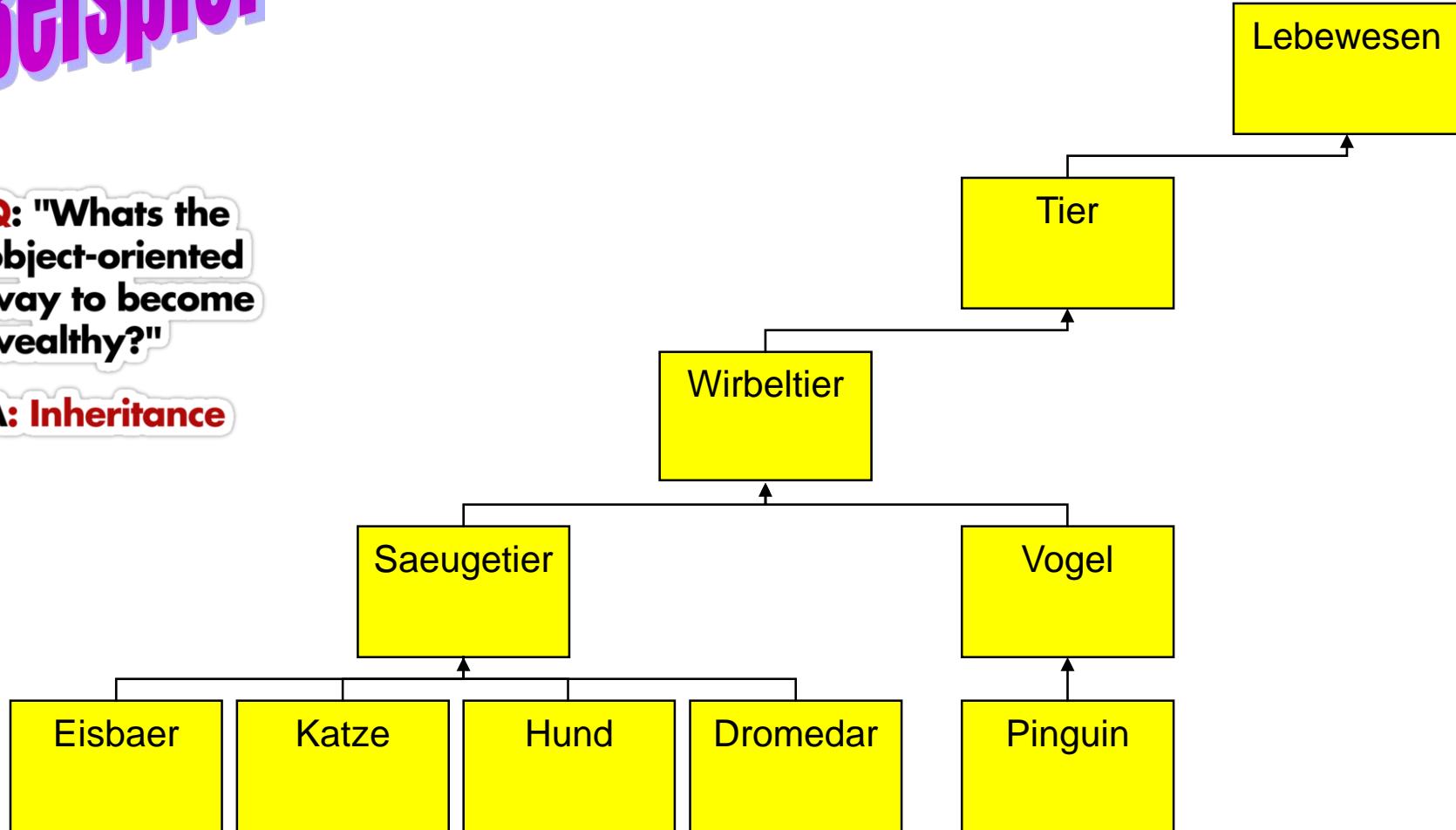
- gemeinsame **Eigenschaften** verwandter Klassen **einheitlich** zu modellieren,
- möglichst viel **Programmcode wiederverwenden** zu können und somit
- **Entwicklungszeit** zu sparen und
- eine Vielzahl von **Fehlerquellen** auszuschalten

Beispiel

Betrachten wir eine „erweiterte Version“ unserer Tierhierarchie:

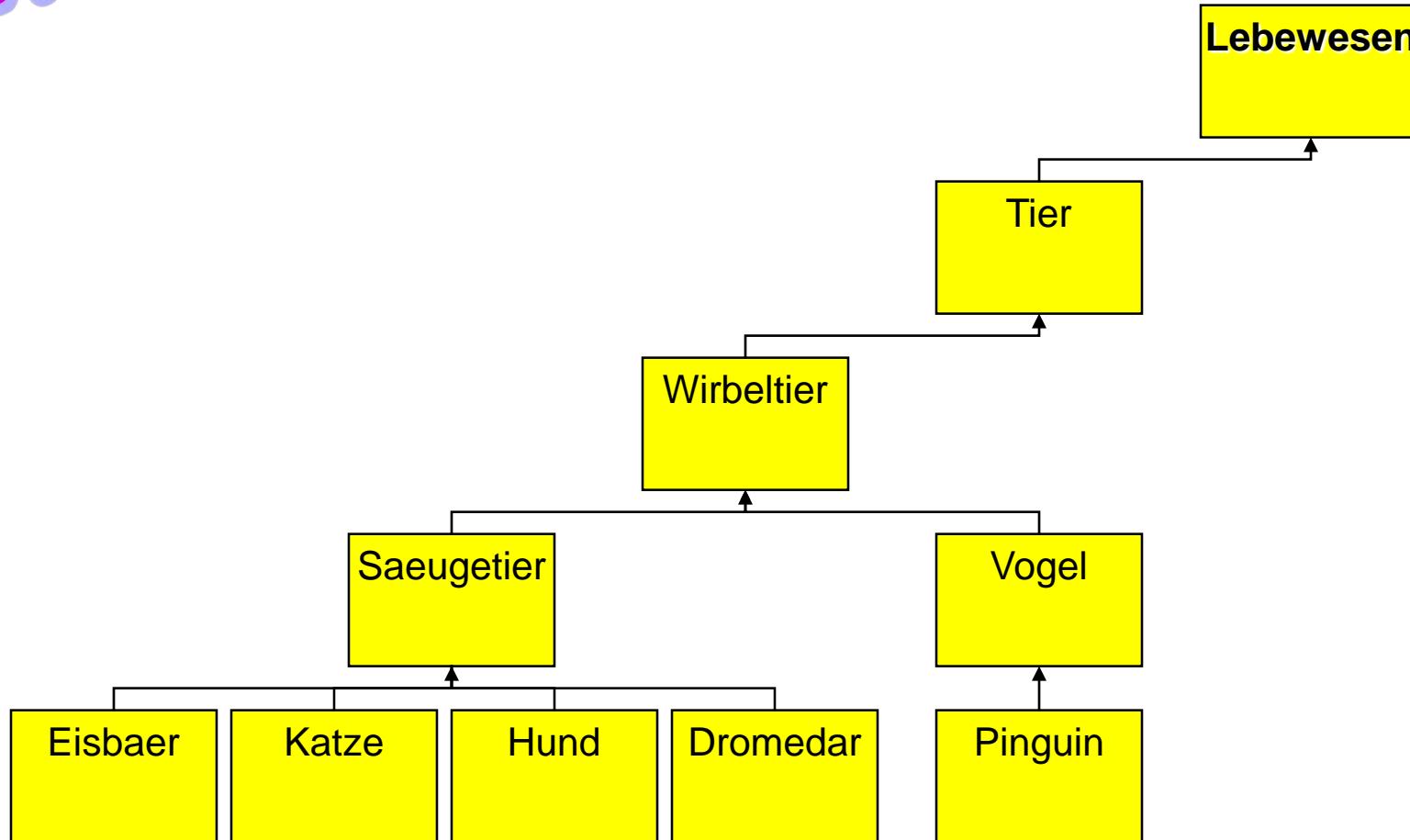
Q: "Whats the object-oriented way to become wealthy?"

A: Inheritance



Beispiel

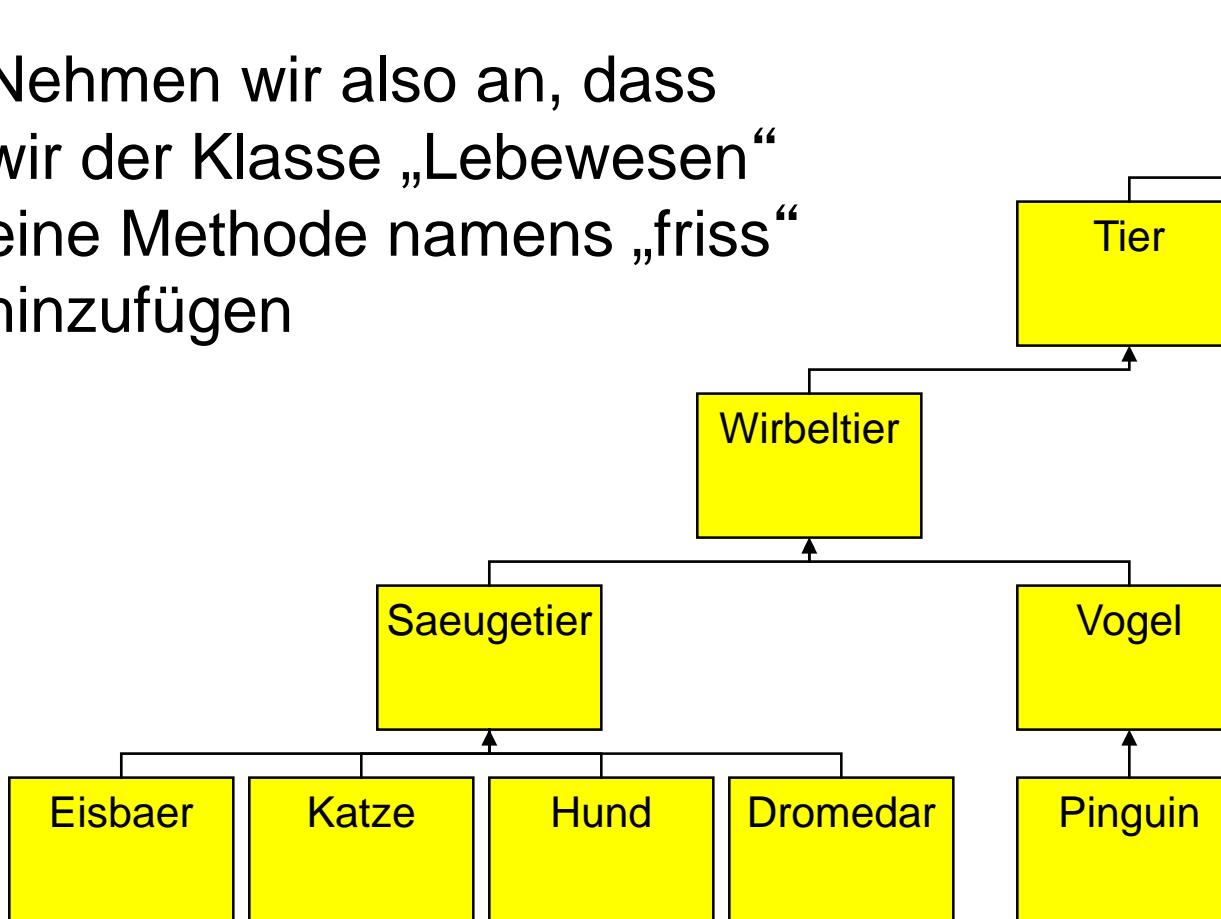
Jede Klasse in diesem Schaubild „ist ein“ Lebewesen und soll deshalb mit diesem gewisse Eigenschaften teilen



Beispiel

Jedes Lebewesen muss beispielsweise essen

Nehmen wir also an, dass wir der Klasse „Lebewesen“ eine Methode namens „friss“ hinzufügen

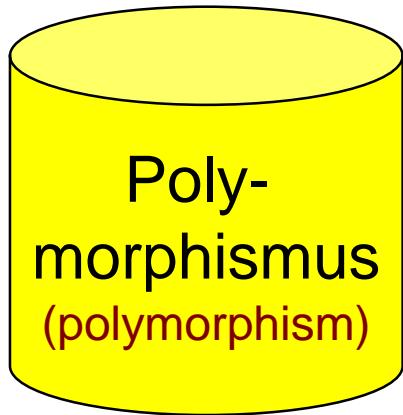


Beispiel

Dann **vererbt** die Klasse diese Methode **automatisch** an alle ihre Subklassen

Jede der hier angegebenen Klassen besitzt somit automatisch die Methode, obwohl wir sie nur für die Klasse „Lebewesen“ programmiert haben!





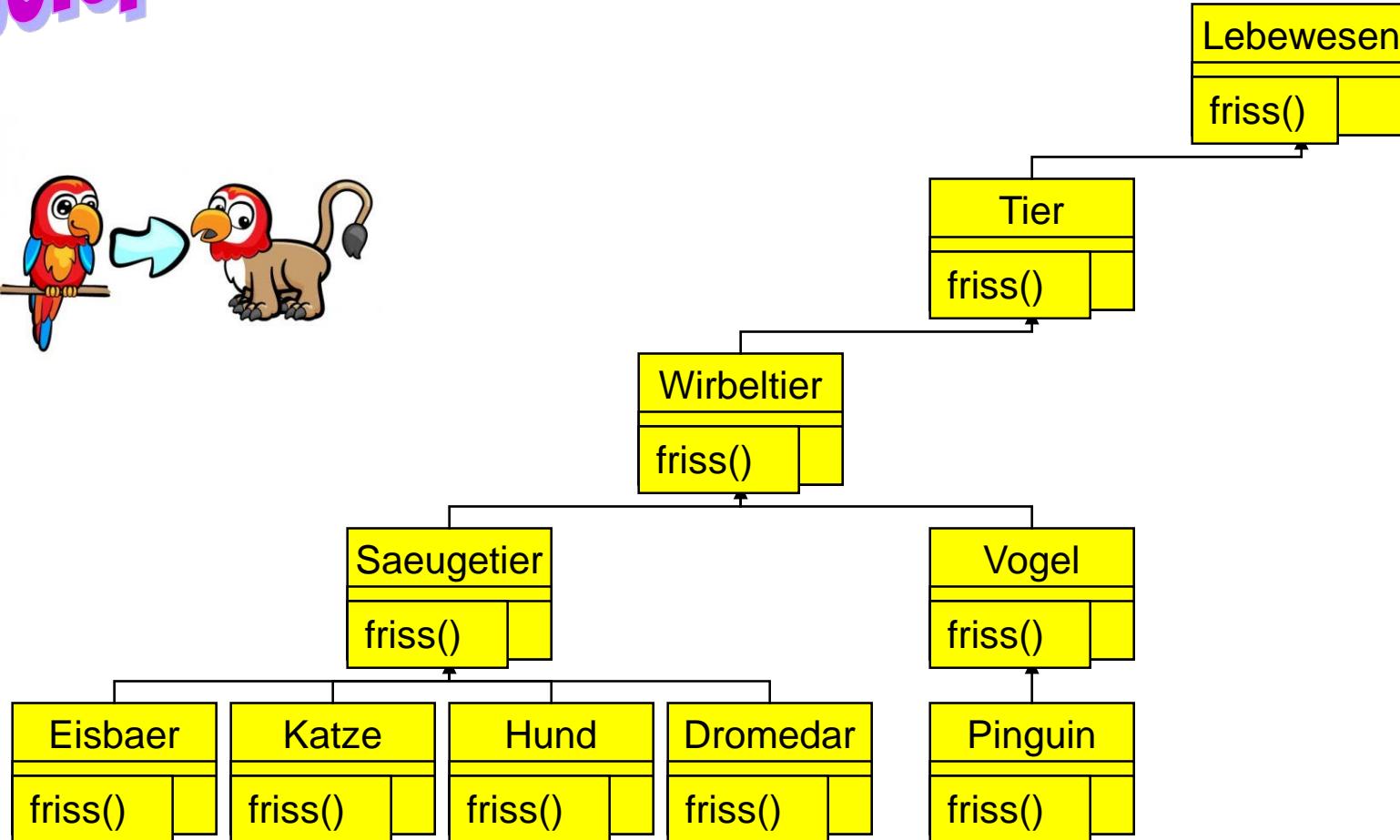
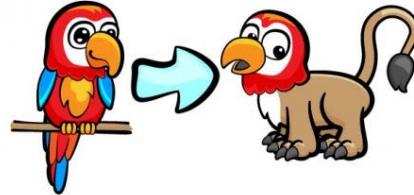
ist die Möglichkeit, Methoden innerhalb der Objekthierarchie zu **überschreiben** (override) und somit individuelles Verhalten verschiedener Klassen auf dieselbe Anweisung zu modellieren

Mit Hilfe des **Polymorphismus** kann der Entwickler

- ohne Probleme **Besonderheiten** bei den verschiedenen Subklassen **berücksichtigen**,
- **einheitliche Schnittstellen** erzeugen und somit
- **Entwicklungszeit** sparen und
- eine Vielzahl von **Fehlerquellen** ausschalten

Beispiel

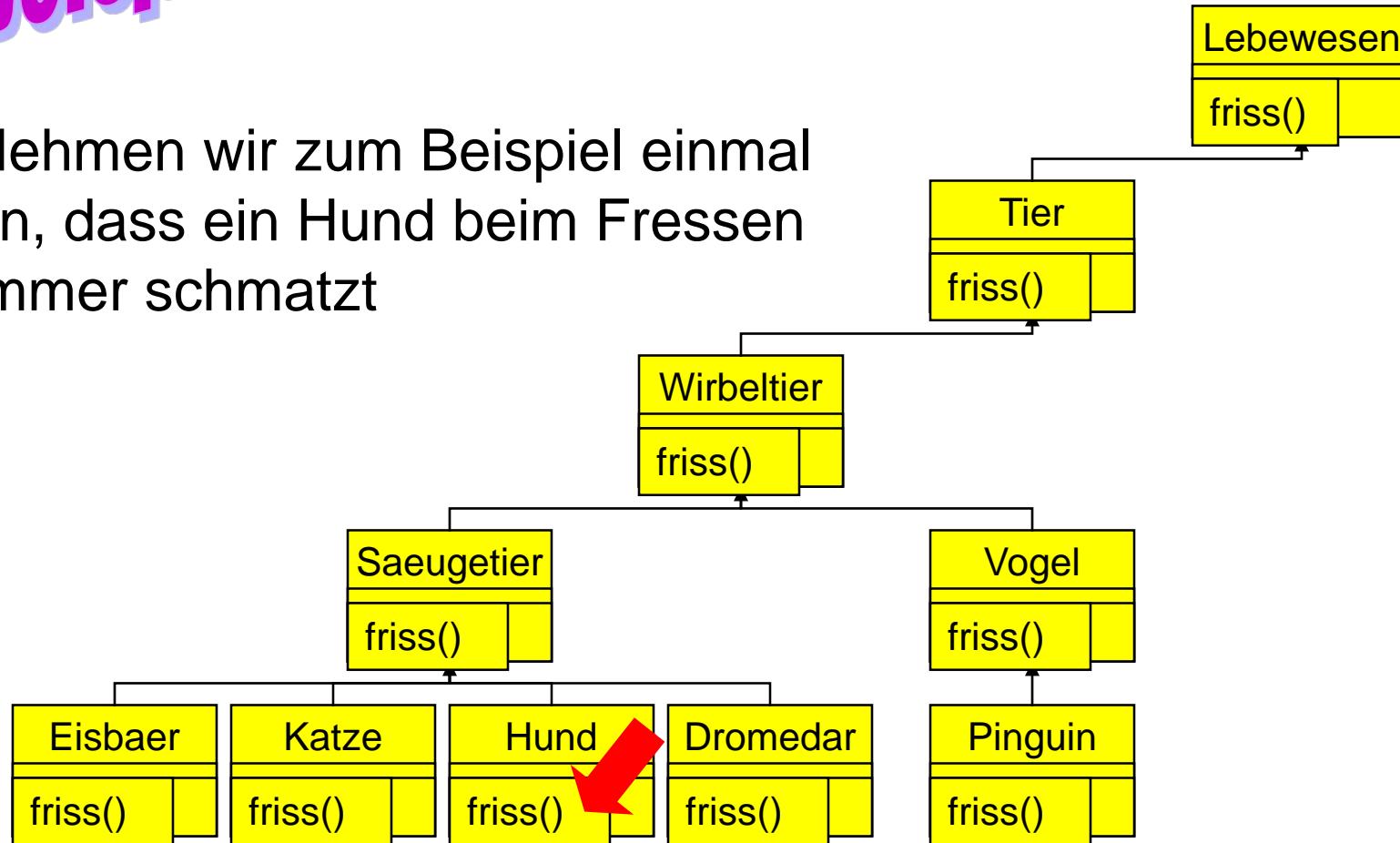
Werfen wir erneut einen Blick auf unser Tiermodell:



Beispiel

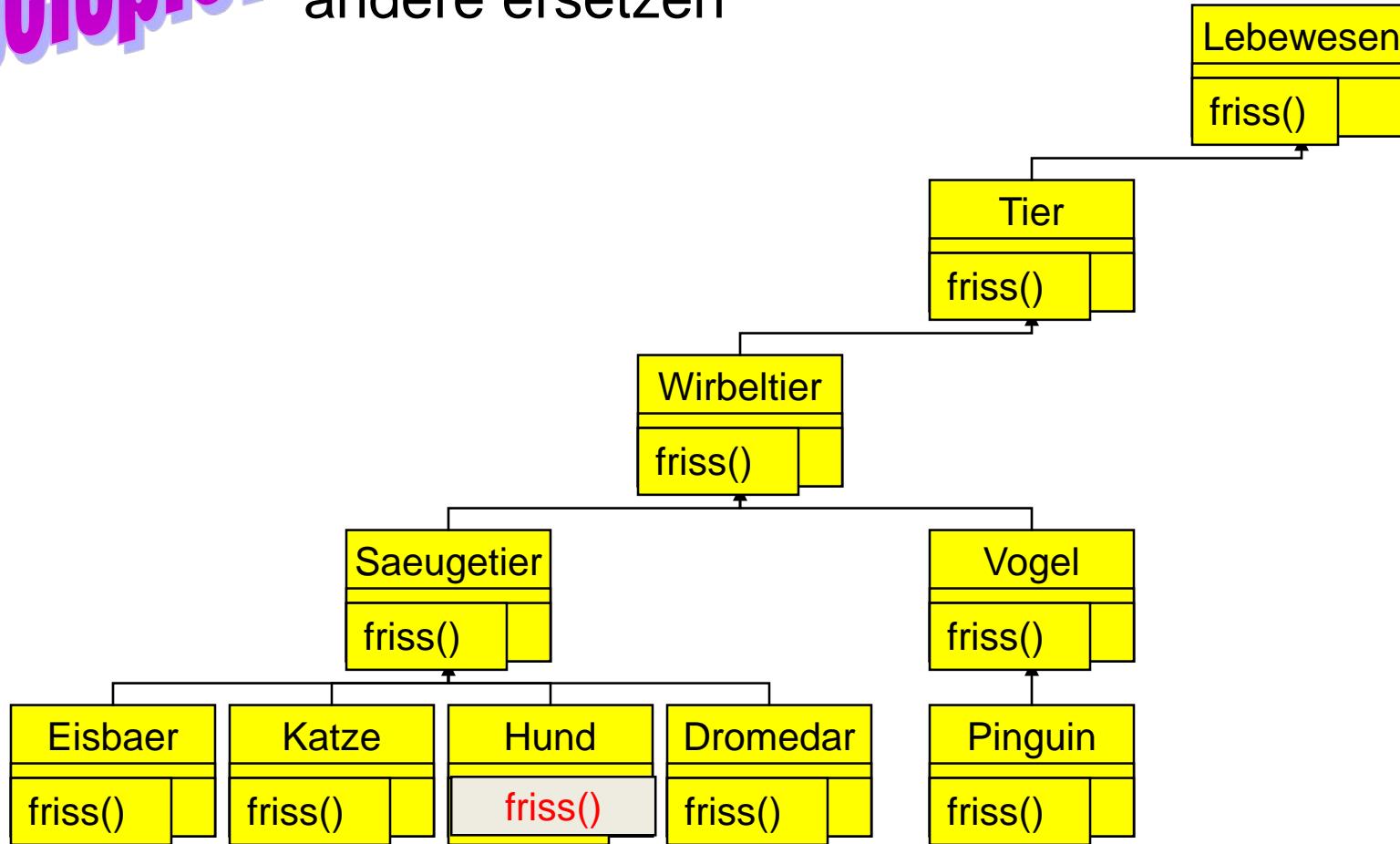
Nicht jedes Lebewesen frisst in unserer Simulation auf dieselbe Weise

Nehmen wir zum Beispiel einmal an, dass ein Hund beim Fressen immer schmatzt



Beispiel

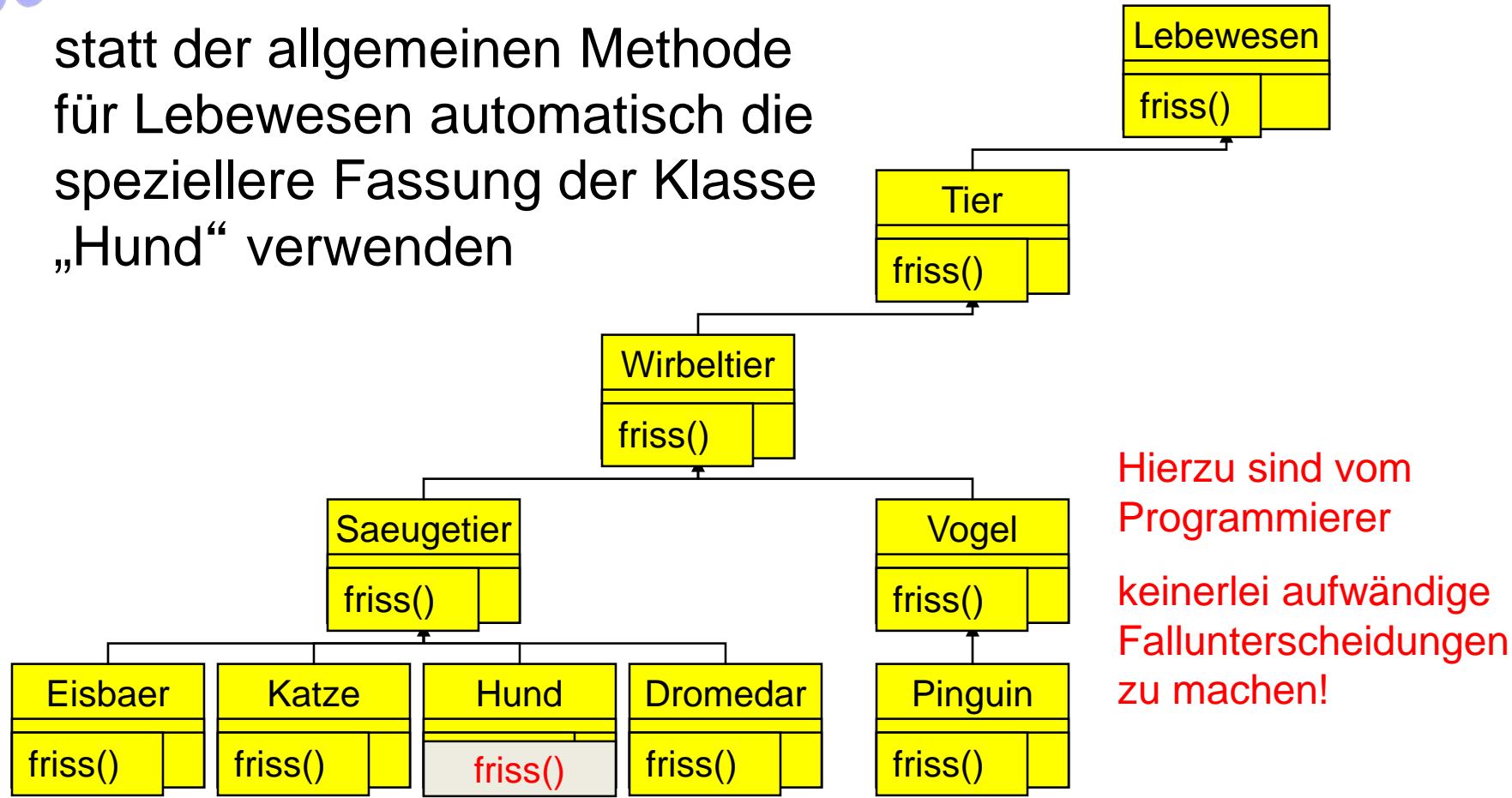
In dieser Situation können wir die alte Methode „friss“ für die Klasse Hund einfach durch eine andere ersetzen

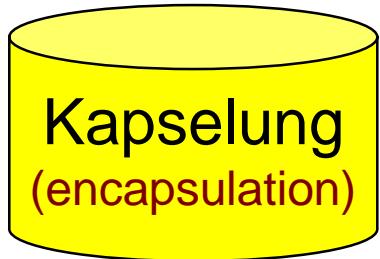


Beispiel

Wenn wir nun für ein Lebewesen der Gattung „Hund“ die Methode „friss“ aufrufen, wird das System

statt der allgemeinen Methode für Lebewesen automatisch die speziellere Fassung der Klasse „Hund“ verwenden





ist die Philosophie, **Daten und Methoden** fest an ihr zugehöriges Objekt zu **binden**

Hierbei werden Daten dem Benutzer normalerweise nicht direkt zugänglich gemacht, sondern vor direkten Zugriffen von außen geschützt (**information hiding**). Das gesamte Verhalten eines Objekts wird nur durch seine Methoden bestimmt.

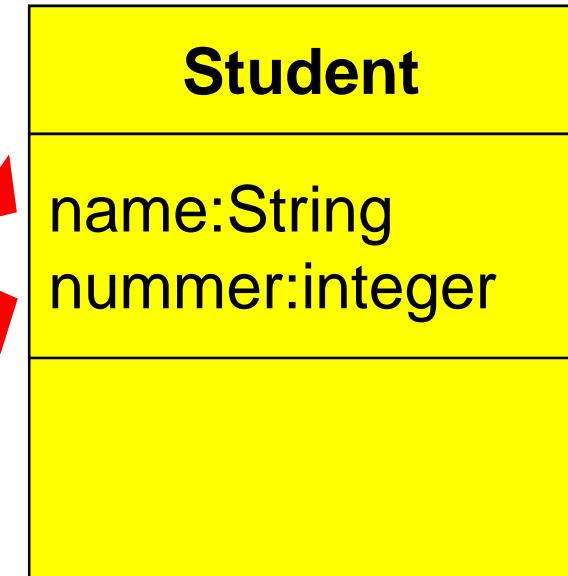
Durch diese Trennung von **Schnittstelle** (der nach außen sichtbaren Methoden) und der **internen Realisierung** werden Programmierer vor Problemen bewahrt, die sich bei einer internen Umstrukturierung der Klasse ergeben.

Beispiel

Wir wollen eine Klasse „Student“ entwerfen, die Namen und Matrikelnummer eines Studenten abspeichern kann

Hierzu spendieren wir der Klasse zwei Variablen:

- in einem String namens „name“ wollen wir den Namen des Studenten ablegen und
- in einer Integer-Variable namens „nummer“ soll die Matrikelnummer gespeichert werden



Beispiel

Wenn wir nun aus unserem „Bauplan“ verschiedene **Instanzen** erzeugen, können wir für jede Instanz die Variablen individuell belegen

Peter:Student
name=„Peter Petersen“
nummer=0978532

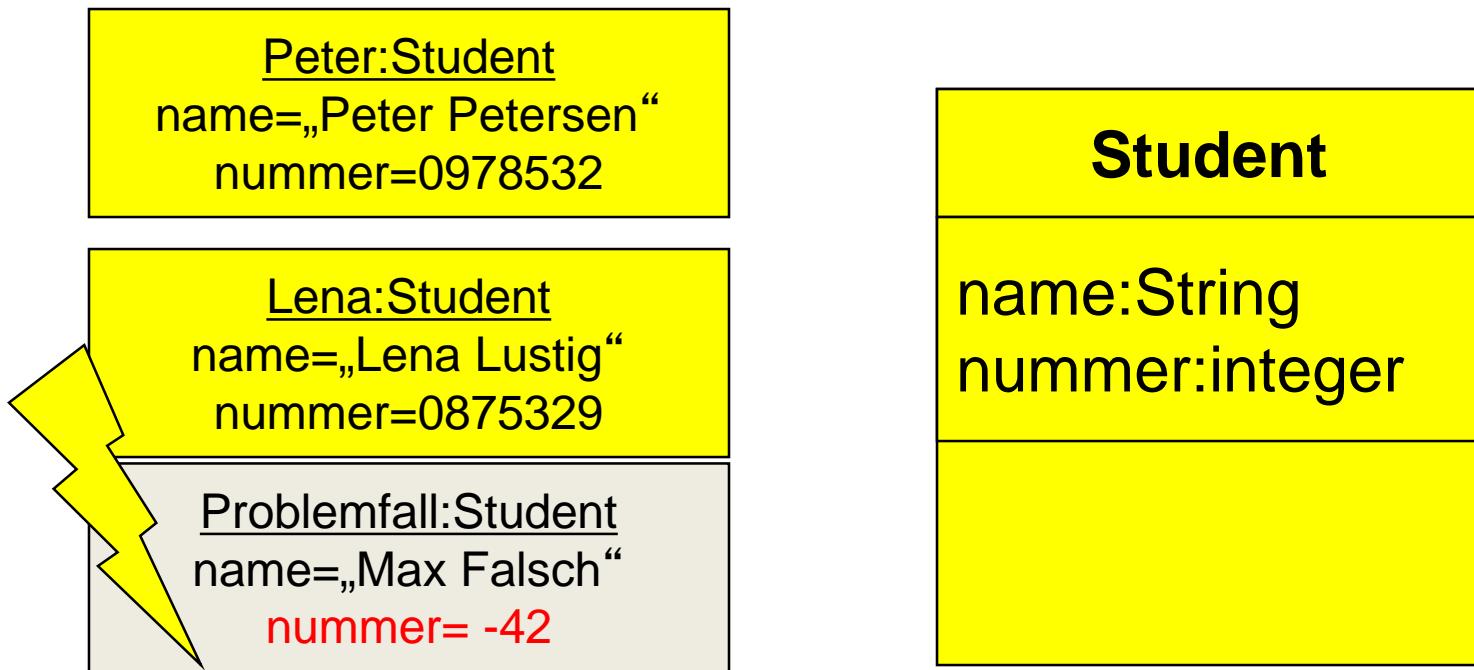
Lena:Student
name=„Lena Lustig“
nummer=0875329

Student

name:String
nummer:integer

Beispiel

Hierbei können wir jedoch nicht verhindern, dass etwa bei der Matrikelnummer eine fehlerhafte Zuweisung erfolgt



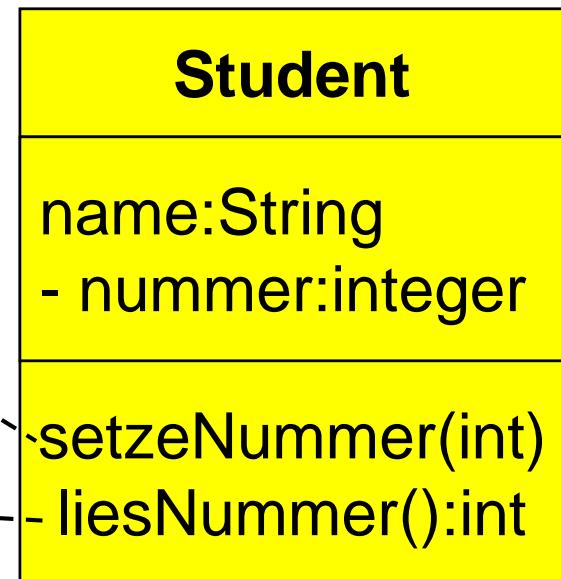
Beispiel

Um diese Probleme zu vermeiden, wird man die internen Datenstrukturen einer Klasse vor den Nutzern üblicherweise verbergen - in diesem Fall also etwa die Variable „nummer“

Zugriffe auf die entsprechenden Daten werden stattdessen über Methoden realisiert

Setzt die Matrikelnummer des Studenten auf den übergebenen Wert. Bei ungültigen Nummern wird die Matrikelnummer auf den Wert 0 gesetzt.

Gibt die Matrikelnummer des Studenten als int-Wert zurück.



Beispiel

Diesen Prozess bezeichnen wir üblicherweise als **information / data hiding**, das heißt als „Verstecken“ der Daten vor dem Benutzer

Student

name:String
- nummer:integer

setzeNummer(int)
liesNummer():int

Beispiel

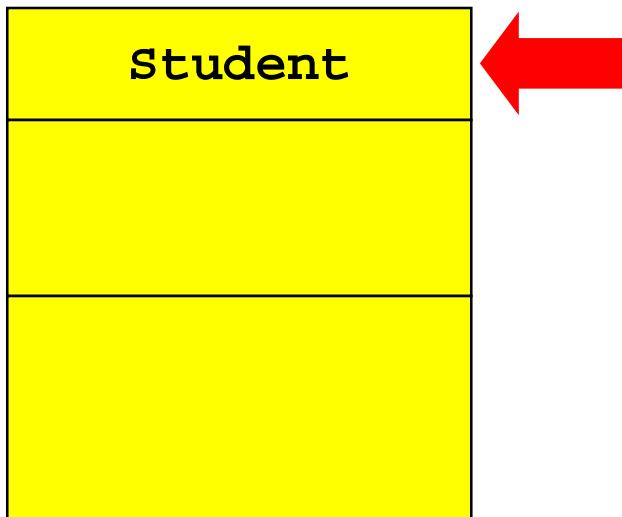
Diesen Prozess bezeichnen wir üblicherweise als **information / data hiding**, das heißt als „Verstecken“ der Daten vor dem Benutzer

Das Binden von Daten und Methoden an eine spezielle Klasse erhebt die Objekte somit über ein reines Konstrukt zum Speichern von Daten hinaus.

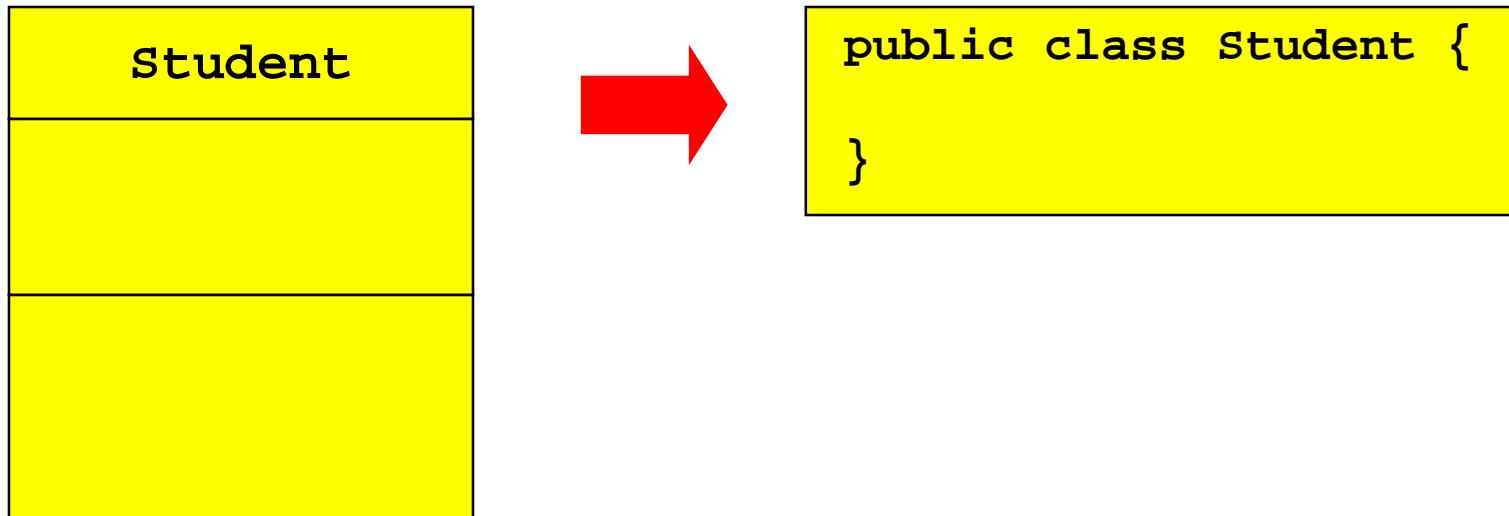
Der Programmierer ist schon beim Entwurf seiner „Datenspeicher“ in der Lage, das spezifische Verhalten seiner Objekte (etwa auf Datenfehler) zu modellieren.

Student
name:String - nummer:integer
setzeNummer(int) liesNummer():int

- Wir werden nun unsere erste Klasse von Objekten modellieren: die Klasse **Student**
- Zu diesem Zweck werden wir auf der einen Seite stets ein graphisches Modell der Klasse in UML betrachten...



- ... und auf der anderen Seite einen Blick auf die entsprechende Realisierung in Java werfen



- Unser zweizeiliger Java-Code stellt bereits eine gültige Klassenbeschreibung dar
- Wir können die Klasse bereits übersetzen und **instantiiieren** - also Objekte aus ihr erzeugen

Student

```
public class TestProgramm {  
    // Meine main-Methode  
    public static void main(String[] args) {  
        // Erzeuge ein Objekt  
        Student charly = new Student();  
    }  
}
```

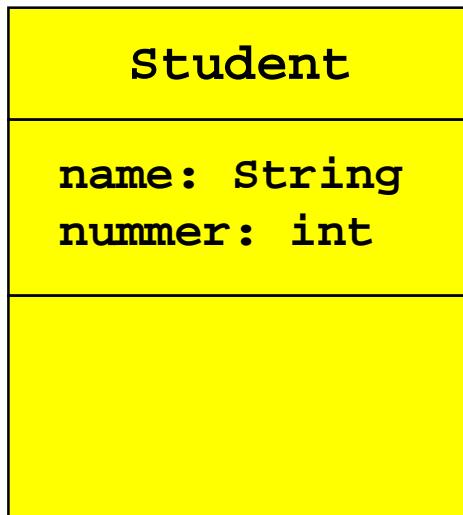
- Die Anweisung innerhalb der `main`-Methode ist für uns nicht neu
- Sie dient der **Instantiierung**, erzeugt also ein **Objekt**

Student

```
public class Student {  
}
```

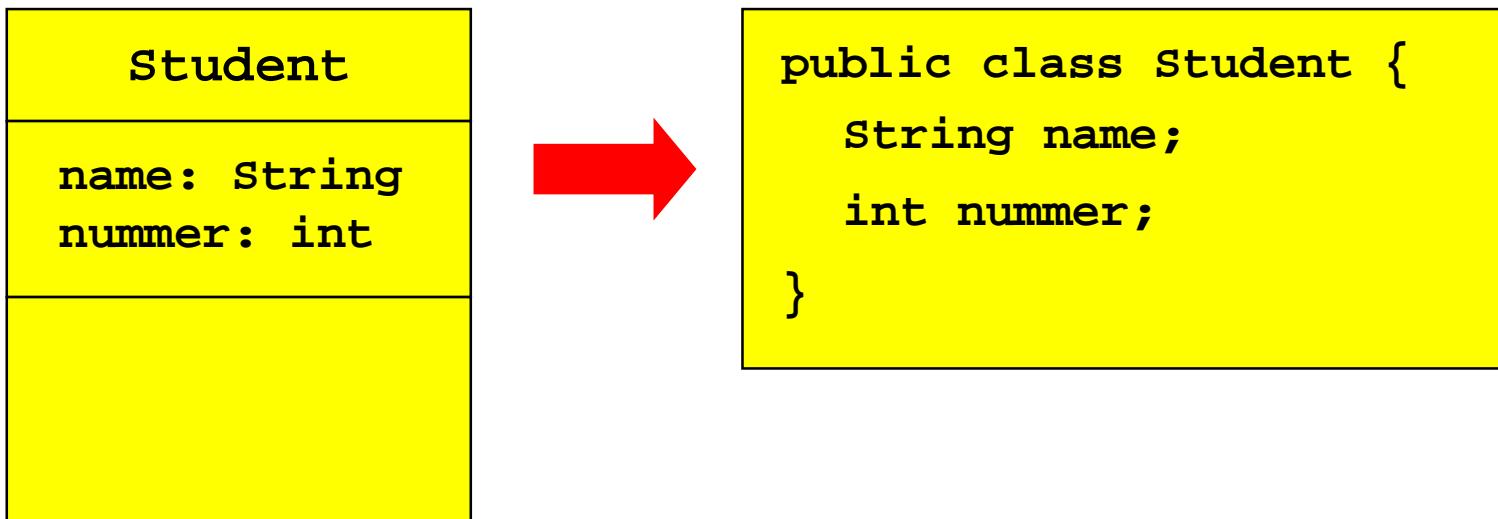
```
Student charly = new Student();
```

- Unsere Klasse soll in der Lage sein, Name und Matrikelnummer eines Studenten im zugehörigen Objekt zu hinterlegen
- Wir tragen unsere Erweiterungen in das UML-Klassendiagramm ein - hierbei stehen Daten immer im oberen der beiden Rechtecke



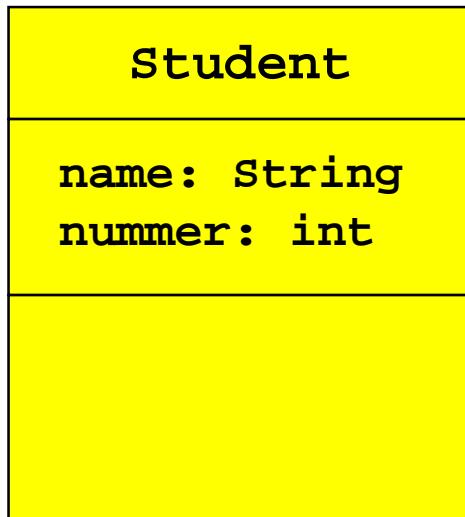
```
public class Student {  
}  
}
```

- Die so definierten Variablen werden beim Erzeugen eines neuen `Student`-Objekts für dieses Objekt automatisch miterzeugt und können vom Benutzer individuell belegt werden
- Sie gehören also der jeweiligen Instanz und werden deshalb als **Instanzvariablen** bezeichnet



Beispiel

Wir wollen in unserem „Testprogramm“ einen Studenten namens „Kalle Karlsson“ mit der Matrikelnummer 978432 erstellen. Der Variablename des Objekts soll hierbei **charly** lauten.



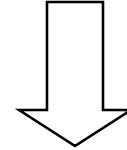
Beispiel

Der Zugriff auf Instanzvariablen erfolgt in Java in der Form
<Objektname>. <Variablenname>

Student

name: String
nummer: int

```
Student charly = new Student();
```



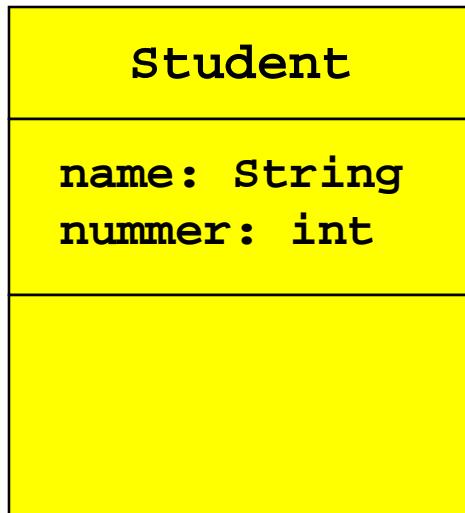
charly:Student

name = ?

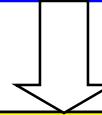
nummer = ?

Beispiel

Wir können die Werte in `name` und `nummer` des Objekts `charly` durch einfache Wertzuweisungen manipulieren

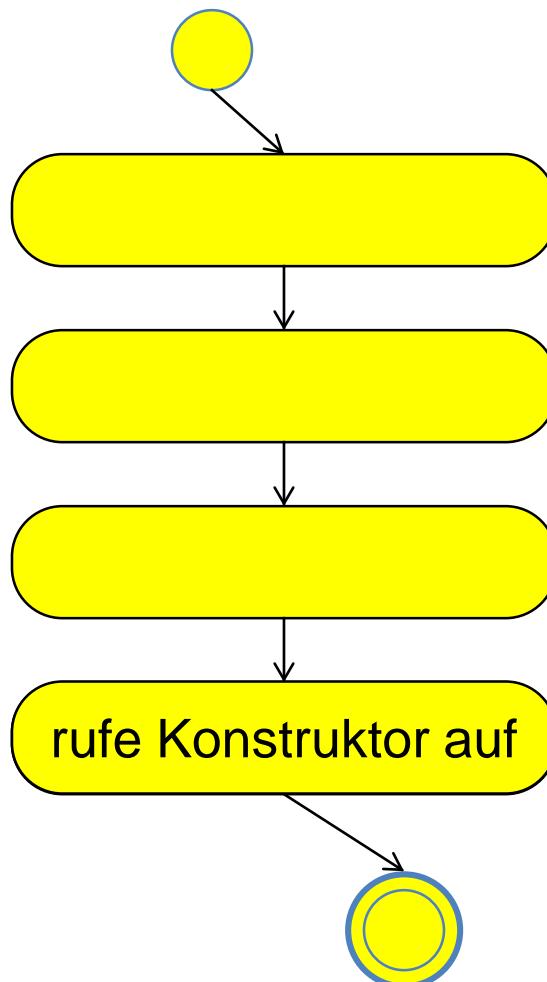


```
Student charly = new Student();
charly.name = "Kalle Karlsson";
charly.nummer = 978432;
```



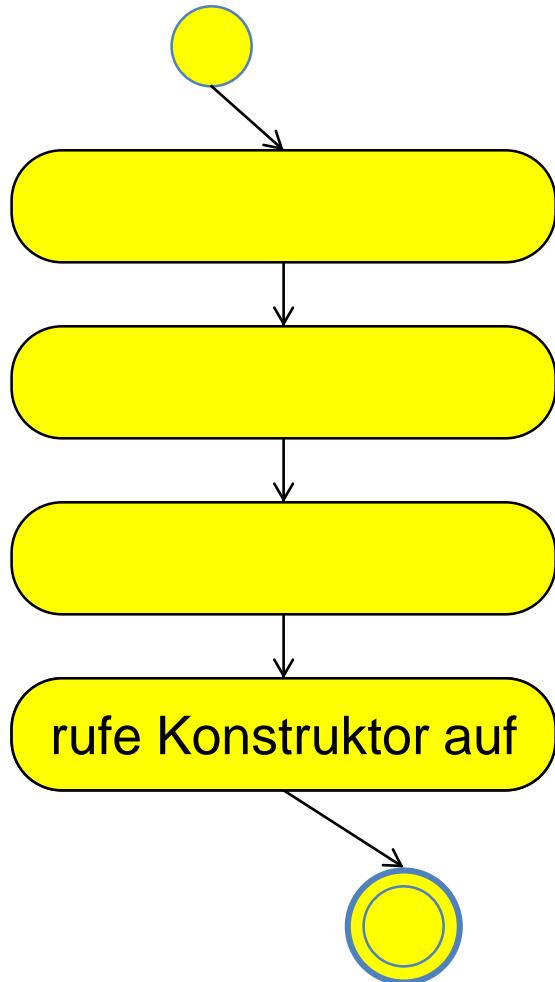
<u>charly:Student</u>
name = „Kalle Karlsson“
nummer = 978432

- Auch in Java fallen Objekte nicht einfach „vom Himmel“, sondern werden nach einem fest vorgegebenen Schema erstellt



Wir wollen an dieser Stelle nicht auf jede einzelne dieser Aktivitäten eingehen; für uns ist lediglich der letzte Schritt von Interesse:

der **Aufruf des Konstruktors**



- Konstruktoren kommen erst am Ende eines Instantiierungsprozesses ins Spiel. Das Objekt ist prinzipiell schon erzeugt: Speicher wurde belegt, Instanzvariablen vereinbart, gewisse Initialisierungen bereits vorgenommen...
- Es ist nun Aufgabe des Konstruktors, Initialisierungsarbeiten zu erledigen, die das System nicht „automatisch“ durchführen kann
- *Hierzu zählt unter anderem das Belegen von Instanzvariablen mit vom Benutzer vorgegebenen Startwerten*

- Werfen wir erneut einen Blick auf unsere Klasse `Student`
- In unserem momentanen Entwicklungsstadium haben wir in der Klasse noch keinen Konstruktor definiert. *Das System benötigt jedoch unbedingt einen Konstruktor, um Objekte bilden zu können!*
- Damit also Java dennoch in der Lage ist, die Klasse zu instantiiieren, wird bei der Übersetzung vom Compiler in der Klassendatei automatisch ein „leerer Konstruktor“ angefügt, dessen einzige Aufgabe es ist, „nichts“ zu tun. Wir bezeichnen diesen Konstruktor in Zukunft als den **Standard-Konstruktor** bzw. den **Default-Konstruktor**

```
public class Student {  
    String name;  
    int nummer;  
}
```

- Würden wir den „leeren“ Default-Konstruktor in unseren Quellcode aufnehmen, hätte dieser die folgende Gestalt:

```
public class Student {  
    String name;  
    int nummer;  
  
    Student() {  
    }  
  
}
```

- Es fällt eine gewisse Ähnlichkeit mit der Vereinbarung von Methoden auf. Wir stellen jedoch fest, dass
 - der Name des Konstruktors **kein beliebiger Bezeichner** sondern der Name der Klasse ist
 - **kein Rückgabetyp** angegeben wurde (nicht einmal void)

- Kommen wir zurück zu unserer Klasse `Student`
- Anstelle des Default-Konstruktors wollen wir uns einen eigenen Konstruktor definieren
- Diesem wollen wir Namen und Matrikelnr. eines Studenten als Argumente übergeben können

```
public class Student {  
    String name;  
    int nummer;  
  
    Student(String n, int nr) {  
        }  
}
```

- In diesem Fall ist also der Name der Klasse wieder

<Klassenname> = `Student`

- und die Liste der Argumente nicht leer, sondern

<Argumentliste> = `String n, int nr`

- Erinnern wir uns nun daran, wie wir unsere Objekte bisher mit Werten initialisiert haben
- Um die einzelnen Werte setzen zu können, haben wir uns eines Zugriffs der Form
`<Objektname>.<Variablenname>` bedient
- **Wie** sollen wir aber innerhalb des Konstruktors wissen, ob unser Objekt nun `charly`, `lena` oder `lassy` heißt?

```
public class Student {  
    String name;  
    int nummer;  
  
    Student(String n, int nr) {  
        name = n;  
        nummer = nr;  
    }  
}
```

```
Student charly = new Student();  
charly.name = "Kalle Karlsson";  
charly.nummer = 978432;
```

- Innerhalb eines Objektes können wir das Objekt selbst mit dem Schlüsselwort **this** ansprechen

```
public class Student {  
    String name;  
    int nummer;  
  
    Student(String n, int nr) {  
  
    }  
}
```

Der Zugriff auf Instanzvariablen innerhalb des Konstruktors erfolgt somit in der Form
this.<Variablename>

Konstruktoren

- Wollten wir also innerhalb des Konstruktors die Instanz-variablen `name` und `nummer` auf „Kalle Karlsson“ und 978432 setzen, so könnten wir dies in der rechts angegebenen Form bewerkstelligen.

```
public class Student {  
    String name;  
    int nummer;  
  
    Student(String n, int nr) {  
        this.name = "Kalle Karlsson";  
        this.nummer = 978432;  
    }  
}
```



```
Student charly = new Student();  
charly.name = "Kalle Karlsson";  
charly.nummer = 978432;
```

Genau genommen können wir das `this` auch weglassen. Da es im Konstruktor keine lokale Variable `name` oder `nummer` gibt, "weiß" die Methode trotzdem, dass sie die Instanzvariablen des eigenen Objekts ansprechen muss!

- Natürlich wollen wir unseren Konstruktor allgemeiner halten und ihm die Werte aus der Argumentliste (die Werte von **n** und **nr**) zuweisen
- Das Vorgehen ist hierbei jedoch vollkommen identisch

```
public class Student {  
  
    String name;  
    int nummer;  
  
    Student(String n, int nr) {  
        this.name= n;  
        this.nummer= nr;  
    }  
}
```

- Natürlich wollen wir unseren Konstruktor allgemeiner halten und ihm die Werte aus der Argumentliste (die Werte von `n` und `nr`) zuweisen
- Das Vorgehen ist hierbei jedoch vollkommen identisch

```
public class Student {  
  
    String name;  
    int nummer;  
  
    Student(String name,  
            int nummer) {  
        this.name = name;  
        this.nummer = nummer;  
    }  
}
```

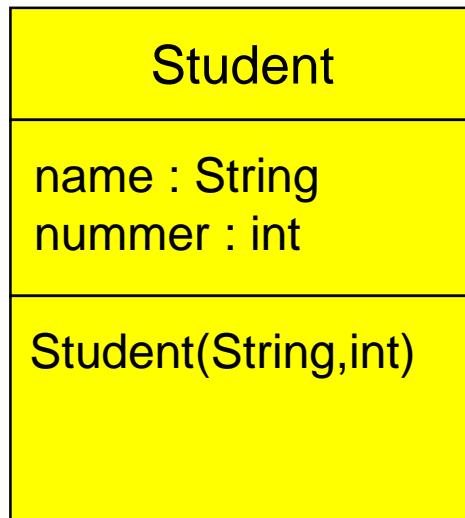
- Natürlich wollen wir unseren Konstruktor allgemeiner halten und ihm die Werte aus der Argumentliste (die Werte von **n** und **nr**) zuweisen
- Das Vorgehen ist hierbei jedoch vollkommen identisch



```
public class Student {  
  
    String name;  
    int nummer;  
  
    Student(String name,  
            int nummer) {  
        this.name = name;  
        this.nummer = nummer;  
    }  
}
```

Anstelle der Parameter **n** und **nr** könnte der Konstruktor auch die Parameter **name** und **nummer** haben, und alles würde funktionieren. Dann könnten wir jedoch das **this** nicht weglassen, da es benötigt wird, um die lokalen Variablen bzw. die Parameter von den Instanzvariablen zu unterscheiden!

- Hierbei tragen wir den Konstruktor in das untere Feld unseres Klassendiagrammes ein
- Wir haben in unserer Klasse einen eigenen Konstruktor definiert. Beim Übersetzen unseres Quelltextes wird der Compiler also keinen Standardkonstruktor mehr einfügen. Der unten angegebene Aufruf des **new**-Operators ist somit also **falsch!**



```
Student charly = new Student();  
charly.name = "Kalle Karlsson";  
charly.nummer = 978432;
```

charly:Student
name = „Kalle Karlsson“
nummer = 978432

- Um unser Objekt `charly` also zu erzeugen, müssen wir dem Operator die von uns gewünschten Initialwerte als Argumente übergeben. Diese werden dann an den Konstruktor weitergeleitet.

Student

name : String
nummer : int

Student(String,int)

```
Student charly =  
new Student("Kalle Karlsson",978432);
```

charly:Student

name = „Kalle Karlsson“
nummer = 978432

- Wir können also mit Hilfe des Konstruktors nicht nur Objekte erzeugen, sondern diesen auch gleich die gewünschten Startwerte zuweisen. Statt drei Befehlen benötigen wir nun nur noch einen!
- Diese Ersparnis kommt uns beim Programmieren insbesondere dann zugute, wenn wir viele Instanzen unserer Klasse erzeugen wollen.

Student
name : String nummer : int
Student(String,int)

```
Student charly =
    new Student("Kalle Karlsson",978432);
Student lena =
    new Student("Lena Lustig",888808);
Student rudi =
    new Student("Rudi Rentier",117704);
Student marion =
    new Student("Marion Maus",123456);
Student jessica =
    new Student("Jessica Java",9875632);
```

Hinweis

- Ähnlich wie bei Methoden ist es natürlich auch möglich, **Konstruktoren zu überladen**. Wir können mehrere Konstruktoren mit unterschiedlicher Argumentliste definieren, so dass der Benutzer je nach Bedarf zw. verschiedenen Alternativen wählen kann.

Student
name : String nummer : int
Student(String,int) Student()

Hinweis

Student
name : String nummer : int
Student(String,int) Student()

- Ähnlich wie bei Methoden ist es natürlich auch möglich, **Konstruktoren zu überladen**. Wir können mehrere Konstruktoren mit unterschiedlicher Argumentliste definieren, so dass der Benutzer je nach Bedarf zw. verschiedenen Alternativen wählen kann.
- Eine mögliche Anwendung hierfür wäre beispielsweise, den inzwischen verschwundenen Default-Konstruktor zu ersetzen. Wir könnten einen neuen Konstruktor mit leerer Argumentliste definieren, der entweder nichts tut oder die Instanzvariablen auf irgendeinen Standardwert setzt.

- Eine Instanzmethode ist eine Methode, die **zu einer speziellen Instanz** einer Klasse (also einem Objekt) gehört
- Instanzmethoden werden (wie auch Instanzvariablen) innerhalb der Klassendefinition vereinbart
- Alle in den vorherigen Lektionen erlernten „Handgriffe“ im Umgang mit Methoden (Überladen, rekursive Definition,...) lassen sich auch auf Instanzmethoden problemlos anwenden

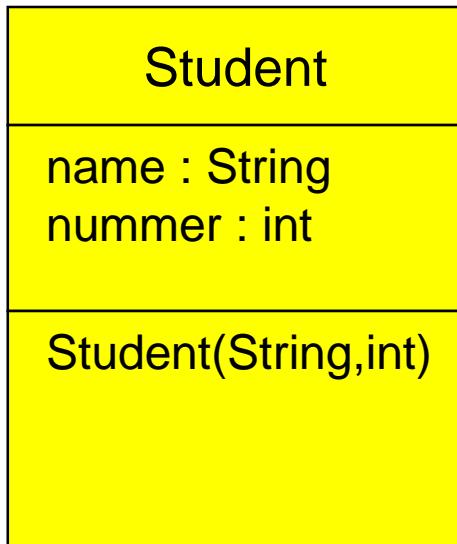
Beispiel

Wir wollen unsere Klasse `Student` so erweitern, dass die in ihr gespeicherten Daten in aufbereiteter Form dargestellt werden können.

Hierbei soll diese „Aufbereitung“ einen String als Ergebnis haben, der in der Form

`Student <name> hat die Matrikelnummer <nummer>.`

gestaltet sein soll.



Beispiel

Student
name : String
nummer : int
Student(String,int)
+ toString() : String

- Wir wollen nun diesen Vorgang der „Aufbereitung“ in unser UML-Klassendiagramm eintragen:
- Wir wollen eine derartige Aufbereitung „auf Befehl“ ausführen können und entwerfen sie deshalb als Methode
- In unserem Modell werden Methoden in das untere Feld eingetragen
- Wir geben der Methode den Namen **toString**, da wir unser Studenten-Objekt in eine Darstellung als String umwandeln wollen
- Wir wollen einen String als Ergebnis zurückgeben

Beispiel

Wir wollen die Methode als Instanzmethode entwerfen, das heisst sie gehört zu dem zu bearbeitenden Objekt und hat somit bereits vollen Zugriff auf die Daten. Wir brauchen ihr also keine Argumente zu übergeben.

Student
name : String nummer : int
Student(String,int) + toString() : String

```
public class Student {  
    ...  
  
    public String toString() {  
        return "Student "  
            + this.name  
            + " hat die Matrikelnummer "  
            + this.nummer  
            + ".";  
    }  
} // Ende der Klasse
```

Beispiel

Der Kopf der Methode entspricht im Prinzip den von uns bisher definierten Methoden - mit der Ausnahme, dass wir das **Schlüsselwort static weggelassen** haben. Wir werden jedoch erst später erfahren, welche Bedeutung dieses Wort genau hat.

Student

name : String
nummer : int

Student(String,int)
+ toString() : String

```
public class Student {  
    ...  
  
    public String toString() {  
        return "Student "  
            + this.name  
            + " hat die Matrikelnummer "  
            + this.nummer  
            + ".";  
    }  
} // Ende der Klasse
```

Beispiel

Da wir im Methodenkopf den Rückgabetyp **String** vereinbart haben, muss unsere Methode natürlich auch einen Wert zurückgeben. Wir geben also durch die return-Anweisung das Ergebnis an die aufrufende Methode weiter.

Student
name : String nummer : int
Student(String,int) + toString() : String

```
public class Student {  
    ...  
  
    public String toString() {  
        return "Student "  
            + this.name  
            + " hat die Matrikelnummer "  
            + this.nummer  
            + ".";  
    }  
} // Ende der Klasse
```

Beispiel

Der Zugriff auf die Instanzvariablen des Objektes erfolgt hierbei wieder durch das bereits bekannte Schlüsselwort **this**

```
public class Student {  
    ...  
  
    public String toString() {  
        return "Student "  
            + this.name  
            + " hat die Matrikelnummer "  
            + this.nummer  
            + ".";  
    }  
} // Ende der Klasse
```

Student
name : String nummer : int
Student(String,int) + toString() : String

Beispiel

Bei dem entstandenen Objekt handelt es sich um einen normalen String, den wir wie gewohnt über die Methode `println` auf dem Bildschirm ausgeben können

Student
name : String nummer : int
Student(String,int) + <code>toString()</code> : String

```
Student charly =  
    new Student("Kalle Karlsson", 978432);  
  
String beschreibung = charly.toString();  
  
System.out.println(beschreibung);
```

charly:Student
name = „Kalle Karlsson“
nummer = 978432

Hinweis

Die soeben vorgestellten Befehlszeilen lassen sich natürlich noch verkürzt darstellen. Wenn man (wie in unserem Beispiel) die textuelle Beschreibung nur in der Ausgabe verwenden will, braucht man sie natürlich nicht explizit mit einem Variablenamen zu benennen.

Student
name : String nummer : int
Student(String,int) + toString() : String

```
Student charly =  
    new Student("Kalle Karlsson", 978432);  
  
String beschreibung = charly.toString();  
  
System.out.println(beschreibung);  
  
System.out.println(charly.toString());
```

Hinweis

In diesem speziellen Fall können wir sogar den Aufruf der `toString`-Methode wegfassen lassen.

Grund hierfür ist die Definition der Methode `println` für Objekte. Diese ruft für das auszugebende Objekt automatisch die Methode `toString` auf und druckt das Ergebnis auf dem Bildschirm.

Student

name : String
nummer : int

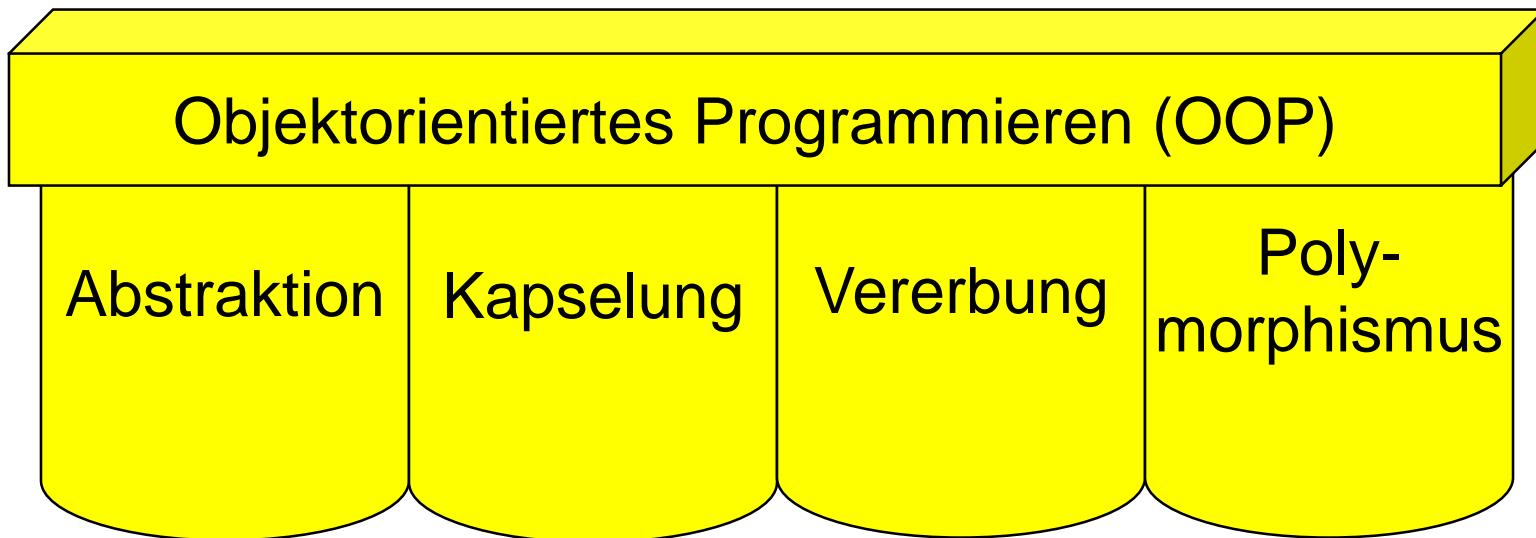
Student(String,int)
+ `toString()` : String

```
Student charly =  
    new Student("Kalle Karlsson", 978432);  
  
String beschreibung = charly.toString();  
  
System.out.println(beschreibung);
```

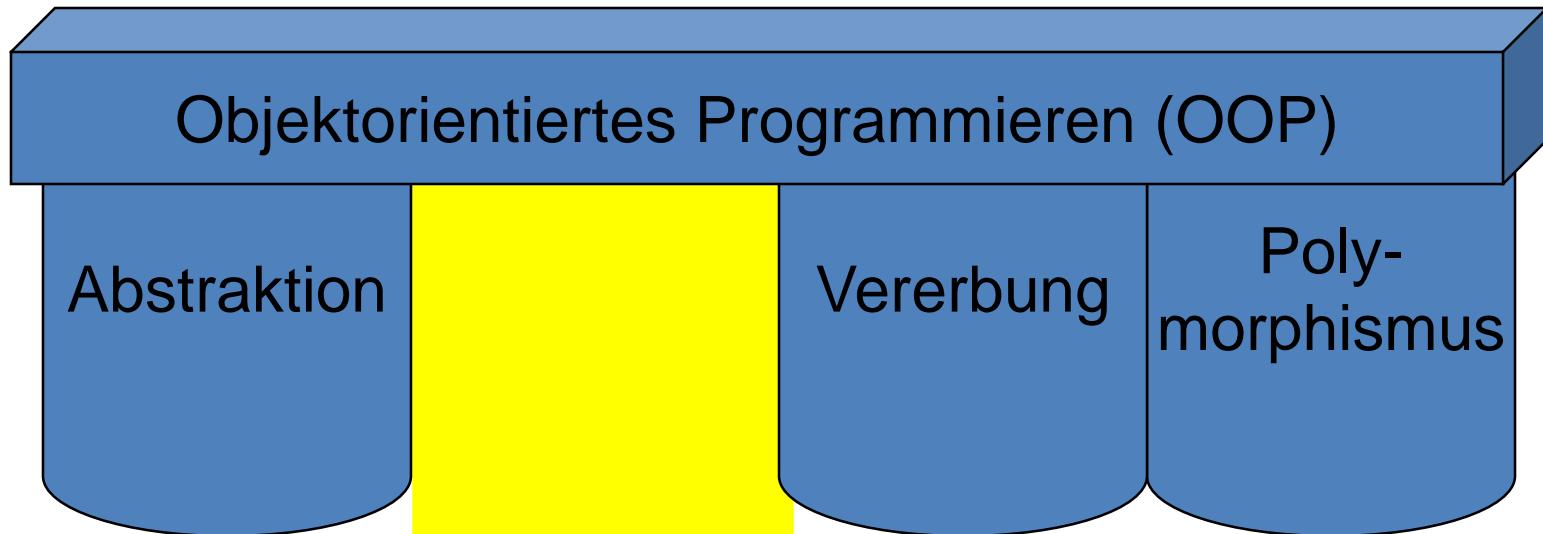
```
System.out.println(charly.toString());
```

```
System.out.println(charly);
```

Vier grundlegende Prinzipien des objektorientierten Programmierens:



Als nächstes: **Kapselung**





ist die Philosophie, Daten und Methoden fest an ihr zugehöriges Objekt zu binden

Daten werden

- dem Benutzer nicht direkt zugänglich gemacht
 - vor Zugriffen von außen geschützt (**information hiding / data hiding**)
- Das Verhalten eines Objekts wird nur durch seine Methoden bestimmt

Die Trennung der **Schnittstelle** (der nach außen sichtbaren Methoden) und der **internen Realisierung** bewahrt Programmierer vor Problemen, die sich bei einer internen Umstrukturierung der Klasse ergeben.

Bis jetzt, Zugriff auf Instanzvariablen mit:

<Objektname>.<Variablenname>

Student
name:String nummer:int
Student(String,int) + <code>toString()</code> : String

Beispiel **Student**:

Daten (**name** und **nummer**) sowie Methoden (**toString**) sind an ihre zugehörigen Objekte gebunden - eine **Grundforderung** des Prinzips der **Kapselung**

Direkter Zugriff auf **name** und **nummer** verstößt gegen die zweite Grundforderung:
das data hiding

Bis jetzt, Zugriff auf Instanzvariablen mit:

<Objektname>.<Variablenname>

Student
name:String nummer:int
Student(String,int) + <code>toString()</code> : String

Beispiel

Dat

(toS

Obje

des Prin

**Wie können wir unsere
Daten besser
„verbergen“?**

oden

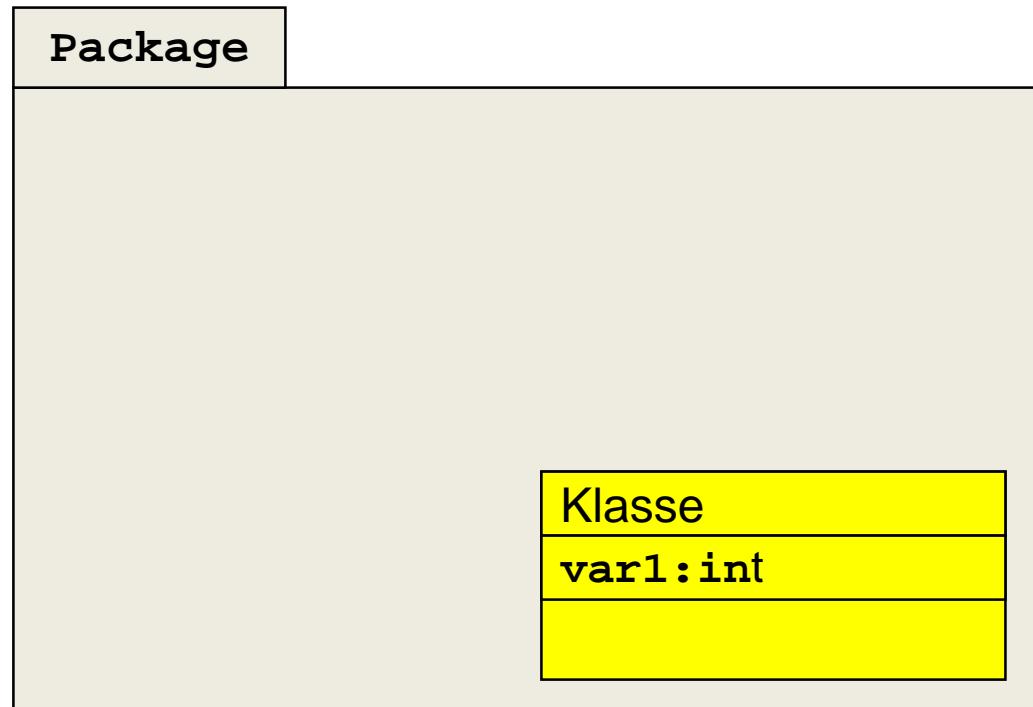
derung

g

Direkter Zugriff auf **name** und **nummer** verstößt gegen die zweite Grundforderung:
das **data hiding**

- Java erlaubt **Zugriffs-Modifizierer (access modifier)**, davon gibt es vier verschiedene:

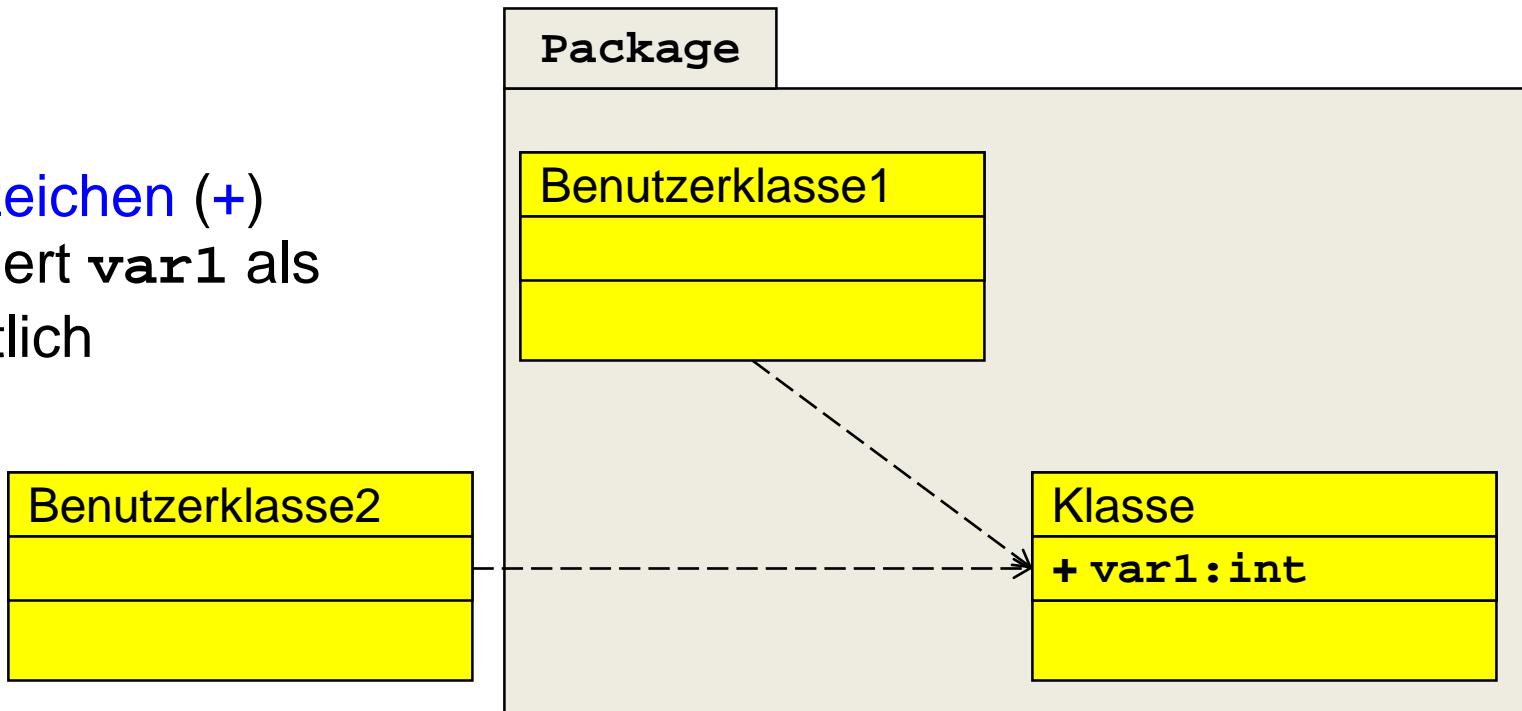
- ① **public**
- ② **private**
- ③ **protected**
- ④ **package**



public (öffentlich): Jede Klasse darf auf eine öffentliche Klasse, Methode oder Variable zugreifen

UML:

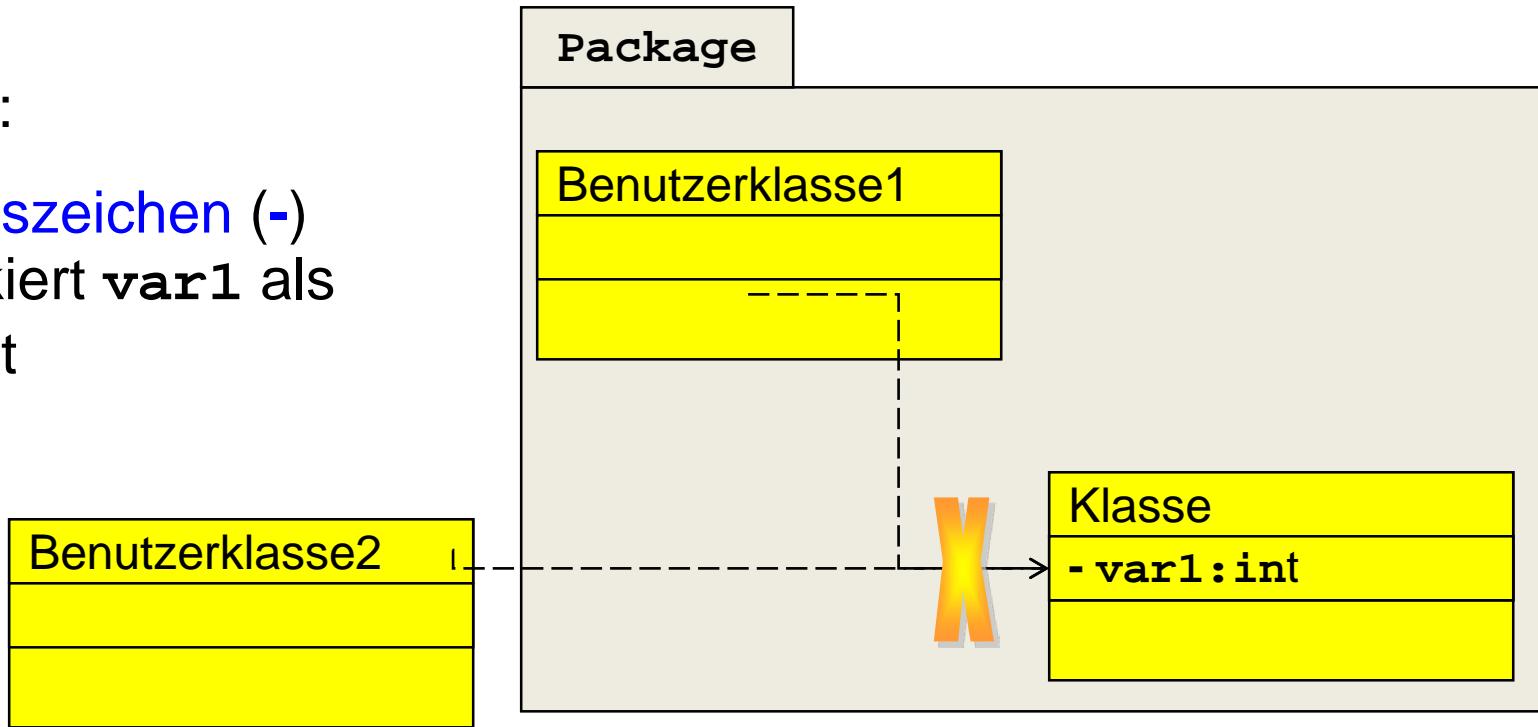
Pluszeichen (+)
markiert var1 als
öffentlich



private (privat): Private Methoden und Variablen können **nur innerhalb der Klasse** verwendet werden, in der sie definiert sind (sie sind nur dort sichtbar)

UML:

Minuszeichen (-)
markiert var1 als
privat



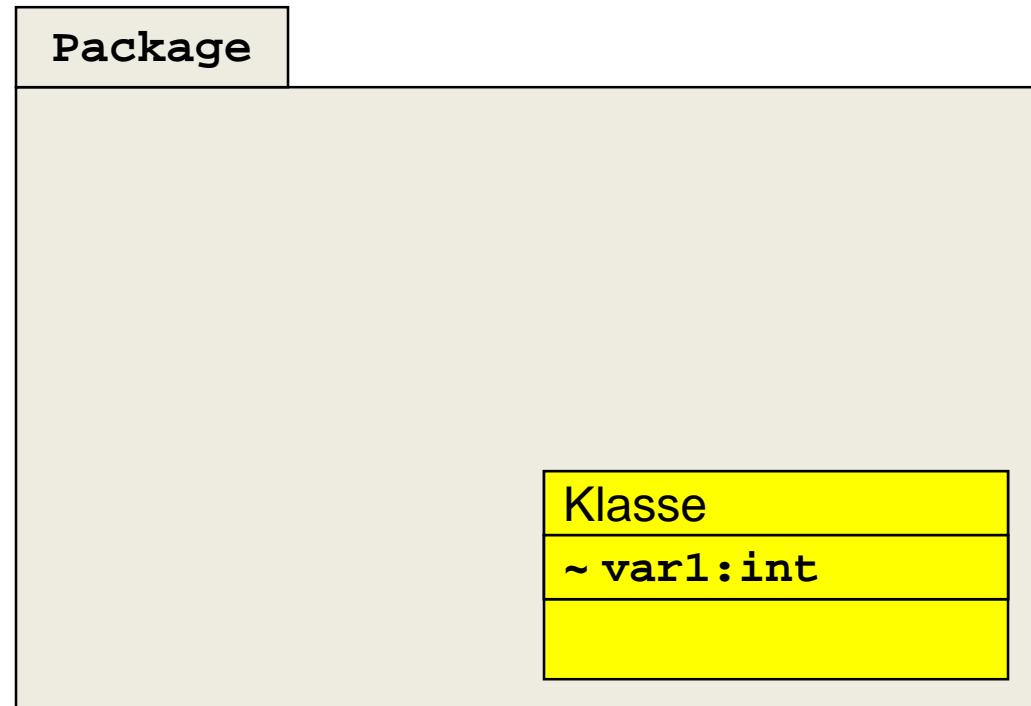
package (Paket): Kann nur innerhalb des Pakets, in dem sich die Klasse befindet, verwendet werden

Anmerkung: Dies ist das Standard-Verhalten (kein Modifizierer)

UML:

keine Unterstützung!

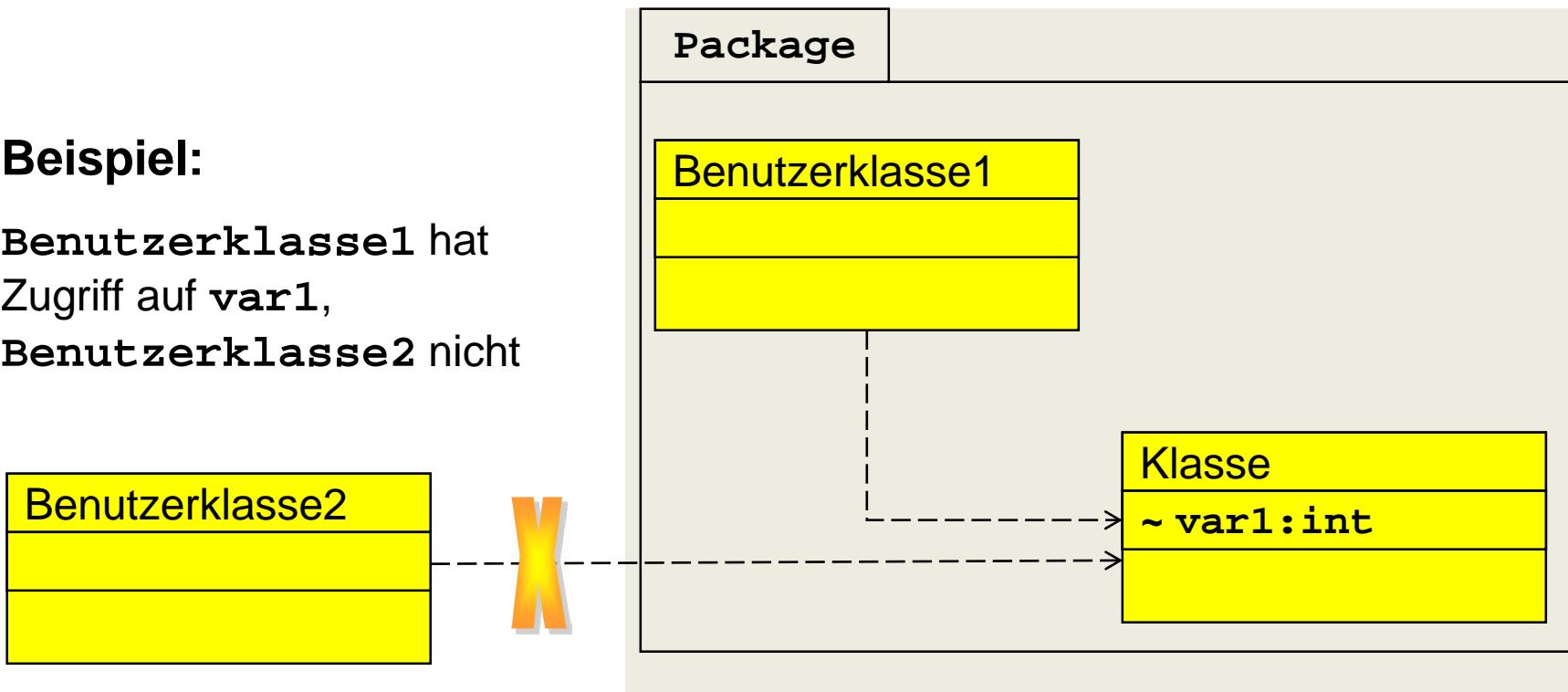
Wir verwenden **Tilde (~)**



Pakete sind Zusammenfassungen von Klassen
z.B.: `java.lang` oder das `java.util`

Beispiel:

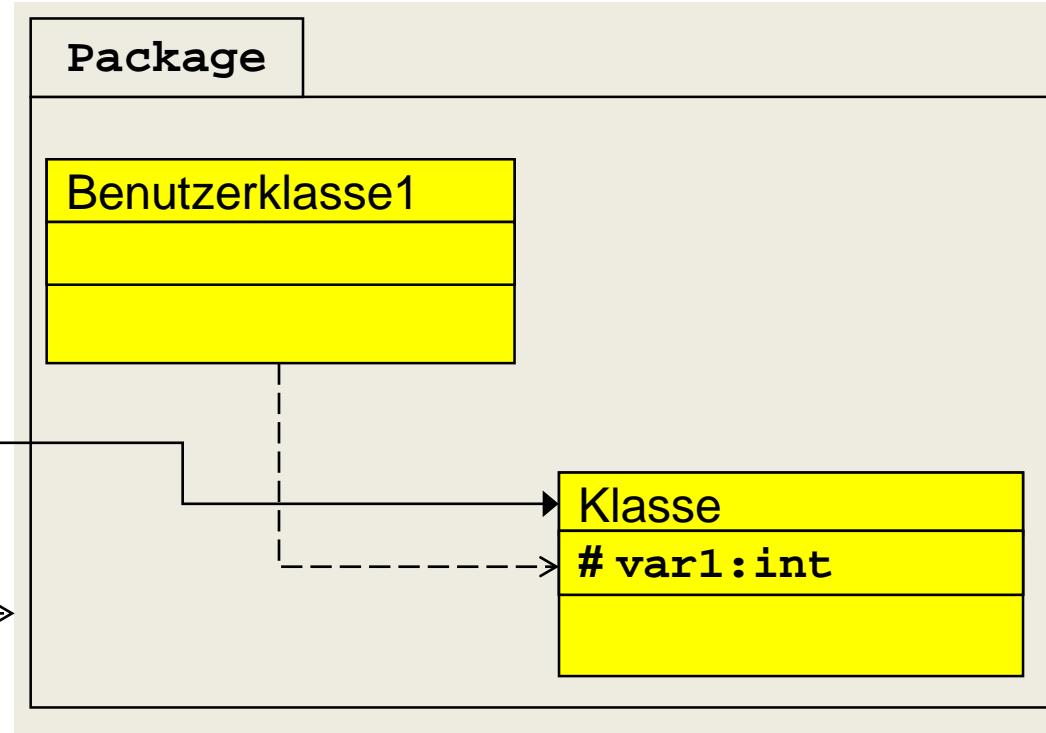
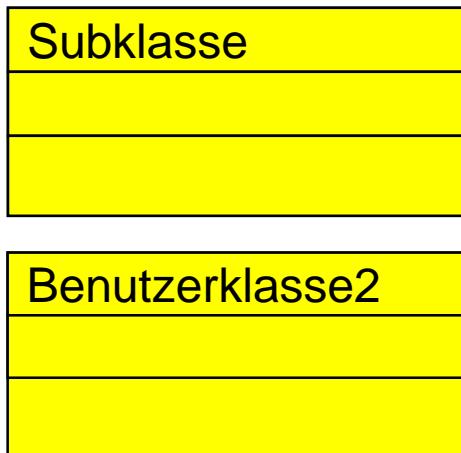
`Benutzerklasse1` hat
Zugriff auf `var1`,
`Benutzerklasse2` nicht



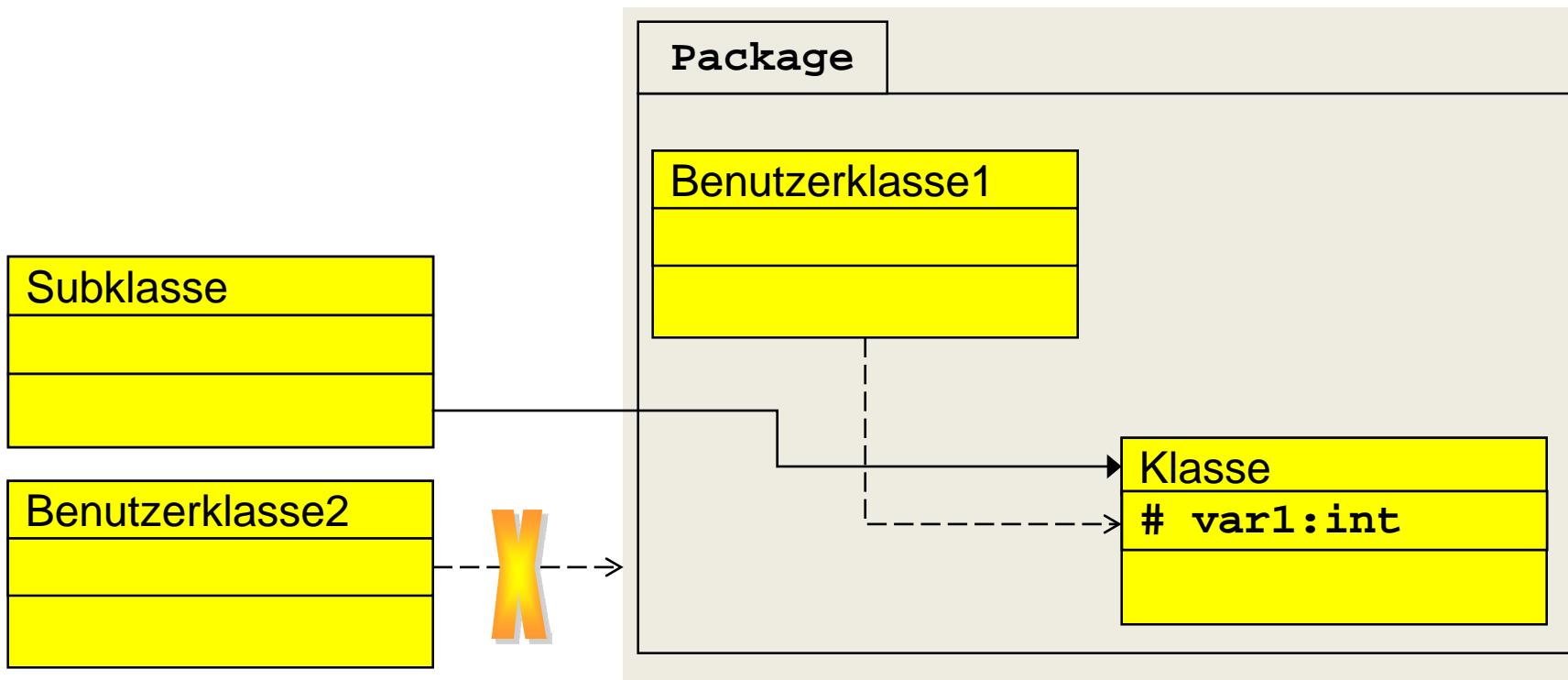
protected (geschützt): Kann nur innerhalb des Pakets, in dem sich die Klasse befindet verwendet werden oder von Subklassen der Klasse

Subklassen sind Klassen mit einer Vererbungsbeziehung (→ VL 13)

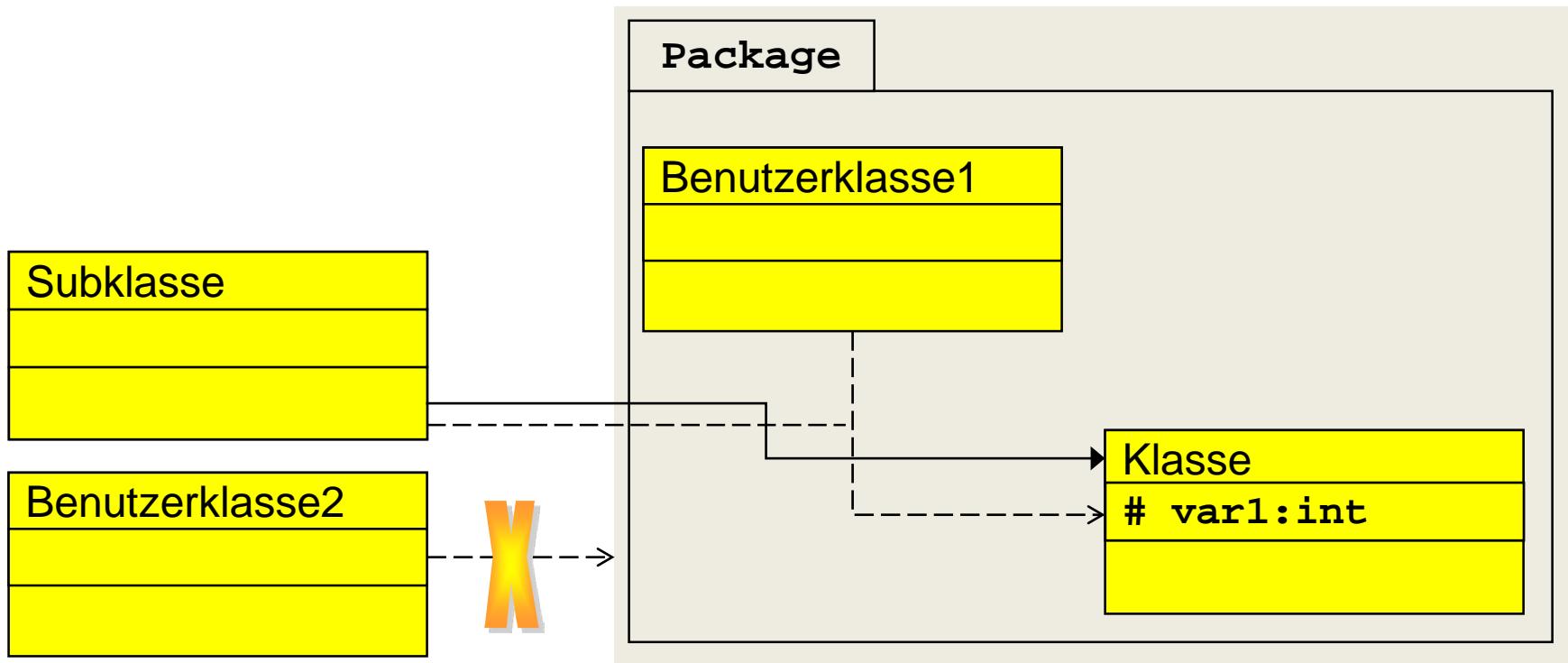
UML: Lattenkreuz (#)
markiert var1 als
geschützt



Der Modus **protected** lässt sich als eine Erweiterung des Standardmodus **package** betrachten. Neben Klassen, die sich in demselben Paket befinden, haben hier auch alle **Subklassen** Zugriff auf ihre Superklasse.



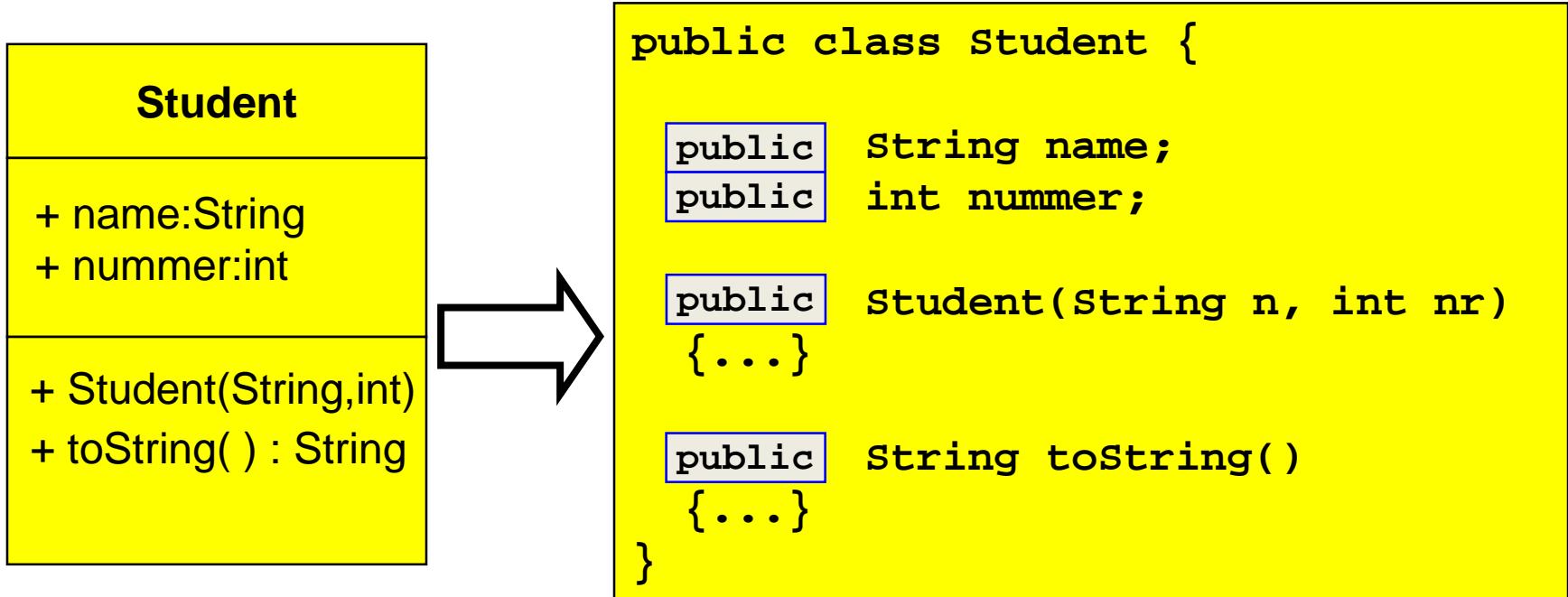
In unserem Klassenmodell steht **Subklasse** in einer „ist-ein“-Beziehung zur **Klasse**. Sie erhält somit ebenfalls Zugriff auf die Variable **var1**



Wir wollen nun alle Bestandteile von `Student` als `public` deklarieren

Dazu fügen wir ein:

- `public` in der Variablen-deklaration / Methodensignatur
- + im UML-Diagramm



Beispiel

Gemäß dem Prinzip des **data hiding** wollen wir unsere Instanzvariablen vor einem Zugriff von außen schützen. Wir ändern die Rechte für **name** und **nummer** deshalb in „privat“ um.

Student
- name:String - nummer:int
+ Student(String,int) + toString() : String

```
public class Student {  
  
    private String name;  
    private int nummer;  
  
    public Student(String n, int nr)  
    {...}  
  
    public String toString()  
    {...}  
}
```

Beispiel

Unsere Daten sind nun vor unbefugten Zugriffen „geschützt“.

Problem: Wie kann eine andere Klasse Name und Matrikelnummer erfragen?

Student

- name:String
- nummer:int

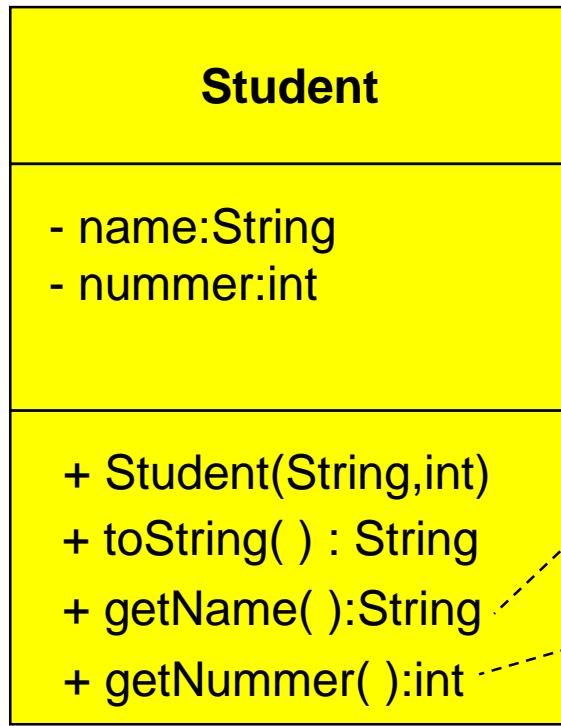
+ Student(String,int)
+ toString() : String

```
public class Student {  
  
    private String name;  
    private int nummer;  
  
    public Student(String n, int nr)  
    {...}  
  
    public String toString()  
    {...}  
}
```

Beispiel

Zugriff erreichen wir mit Instanzmethoden

- **getName**
- **getNummer**



```
public String getName() {  
    return this.name;  
}
```

```
public int getNummer() {  
    return this.nummer;  
}
```

Beispiel

Aufruf der Instanzmethode `charly.getName()`, um den Namen zu erhalten

Student

- name:String
- nummer:int

- + Student(String,int)
- + `toString()` : String
- + `getName()`:String
- + `getNummer()`:int

```
String charlysName = charly.getName();  
System.out.println(charlysName);
```

charly:Student
name=„Kalle Karlsson“
nummer=978432

Wir erweitern die Studentenkartei für den Gebrauch an mehreren Universitäten

Problem:

Student
- name:String - nummer:int
+ Student(String,int) + toString(): String + getName():String + getNummer():int

Name und Matrikelnummer sind nur in **einer** Universität eindeutig. Wir haben keine Garantie dafür, dass es nicht beispielsweise an den Universitäten Würzburg und München zwei **verschiedene** Studenten namens „Kalle Karlsson“ mit Matrikelnummer 978432 gibt.

Wie sollen wir diese Studenten voneinander unterscheiden?

Zwei Objekte sind gleich, wenn sie **das selbe Objekt** sind (`charly1==charly2`). Dies ist nur genau dann der Fall, wenn ihre Referenzen auf die gleiche Stelle zeigen.

Student
- name:String - nummer:int - id:int
+ Student(String,int) + toString() : String + getName():String + getNummer():int

Jedes `Student`-Objekt soll mit einer eindeutigen Identifikationsnummer (`id`) versehen werden. Die Nummer wird im Konstruktor gesetzt und kann zu Lebzeiten des Objektes nie wieder verändert werden.

Ziel: Zwei Objekte sollen als gleich gelten, wenn sie dieselbe Nummer besitzen.

Problem: Wie können wir die Instanzvariable `id` mit einem Wert belegen, der für jede Instanz der Klasse `Student` unterschiedlich ist? Woher soll der Konstruktor `wissen`, welche Werte schon vergeben wurden?

Student
- name:String - nummer:int - id:int
+ Student(String,int) + toString() : String + getName():String + getNummer():int

```
...
private int id;

public Student(String n,int nr) {
    this.name=n;
    this.nummer=nr;
    this.id = ????
}
...
```

Der Konstruktor unserer Klasse `Student` benötigt Informationen darüber, welche Nummern bereits für andere Objekte vergeben wurden

Allgemeiner: Wie speichern wir eine Information, die nicht zu einem speziellen Objekt, sondern **zu der gesamten Klasse gehört?**

Klassenvariablen und Klassenmethoden:

- Existieren mit der Klasse
- Werden nicht mit jedem Objekt neu erzeugt
- Werden auch **statische** Variablen/Methoden genannt
- Werden mit dem Schlüsselwort **static** deklariert

Der Zugriff auf statische Teile einer Klasse erfolgt in der Form

`<Klassename>.<Methoden-/Variablenname>`

oder in der Form

`<Objektname>.<Methoden-/Variablenname>`

wobei `<Objektname>` der Name einer Instanz der Klasse ist

Beispiel

Konstruktor unserer Klasse **Student**:

Student

- name:String
- nummer:int
- id:int

- + Student(String,int)
- + toString():String
- + getName():String
- + getNummer():int

```
...
private int id;

public Student(String n, int nr) {
    this.name = n;
    this.nummer = nr;
    this.id = ???

}
```

...

Beispiel

Wir speichern die nächste freie ID in einer **Klassenvariable** namens **zaehler**

UML: statische Elemente werden **unterstrichen**

Student
- name:String - nummer:int - id:int <u>- zaehler:int</u>
+ Student(String,int) + toString():String + getName():String + getNummer():int

```
...
private int id;

public Student(String n, int nr) {
    this.name = n;
    this.nummer = nr;
    this.id = ????
}

...
```

Beispiel

- Statische Klassenvariable **zaehler**
- **id** wird mit dem Wert von **zaehler** initialisiert
- Danach wird der **zaehler** inkrementiert

Student
- name:String - nummer:int - id:int <u>- zaehler:int</u>
+ Student(String,int) + toString():String + getName():String + getNummer():int

```
...
private int id;
private static int zaehler = 1;

public Student(String n,int nr) {
    this.name=n;
    this.nummer=nr;
    this.id = Student.zaehler;
    Student.zaehler++;
}

...
```

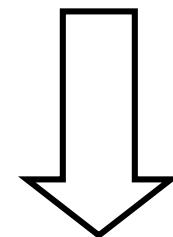
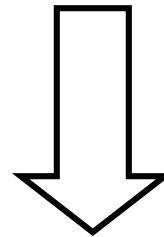
Beispiel

Die **ids** sind unterschiedlich, obwohl wir an dem Aufruf des Konstruktors keine Veränderungen vorgenommen haben!

Student
- name:String - nummer:int - id:int - zaehler:int
+ Student(String,int) + toString():String + getName():String + getNummer():int

```
Student charly = new Student  
          ("Kalle Karlsson", 978432);
```

```
Student lena = new Student  
          ("Lena Lustig", 888808);
```



charly:Student
name=„Kalle Karlsson“
nummer=978432
id=7

lena:Student
name=„Lena Lustig“
nummer=888808
id=8

Hinweis

Wichtige Regel zur Verwendung statischer Methoden:

```
public static void main(String[] args) {  
    ...  
}  
  
public static int fakultaet(int n) {  
    if (n==0) return 1;  
    else return n*fakultaet(n-1);  
}  
  
public static int tuwas(double d) {  
    ...  
}
```

Hinweis

Wichtige Regel zur Verwendung statischer Methoden:

**Statische Methoden haben keinen
Zugriff auf nicht-statische Elemente!**

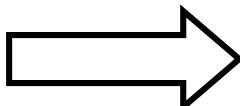
Beispiel

Student
- name:String - nummer:int - id:int <u>- zaehler:int</u>
+ Student(String,int) + toString():String + getName():String + getNummer():int <u>+ letzteID():int</u>

Statische Methode: **letzteID** soll die letzte vergebene Identifikationsnummer zurückliefern

zaehler ist statisch → **letzteID** kann darauf zugreifen

Der Klassename (**Student**) könnte auch entfallen, weil wir uns in der Klasse **Student** befinden



```
public static int letzteID() {  
    return Student.zaehler - 1;  
}
```

Beispiel

Die statische Methode **schreibeName** liefert folgenden Compilerfehler:

**Can't make a static reference
to a nonstatic variable**

Student

- name:String
- nummer:int
- id:int
- zaehler:int

+ Student(String,int)
+ toString():String
+ getName():String
+ getNummer():int
+ letzteID():int

Grund: Zugriff auf Instanzvariable **name**!

Wir befinden uns aber in keiner Instanz (also einem Objekt) der Klasse **Student**, d.h. die Variable existiert nicht!

```
public static void schreibeName() {  
    System.out.println(name);  
}
```

Beispiel

Zugriffe auf **Instanzvariablen** benötigen ein Objekt!

Zugriffe auf **Klassenvariablen** benötigen **kein** Objekt!

Student
- name:String - nummer:int - id:int <u>- zaehler:int</u>
+ Student(String,int) + toString():String + getName():String + getNummer():int <u>+ letzteID():int</u>

```
public static void schreibeName(Student s) {  
    System.out.println(s.name);  
}
```

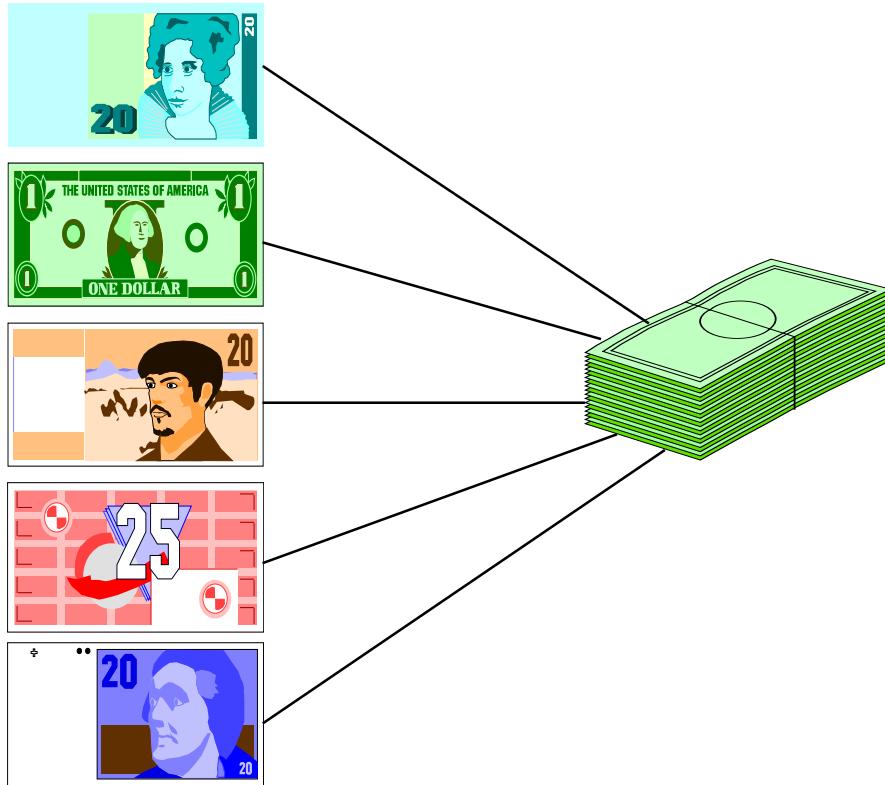
Instanzvariable

Klassenvariable

```
public static int letzteID() {  
    return Student.zaehler-1;  
}
```

Aufgabenstellung

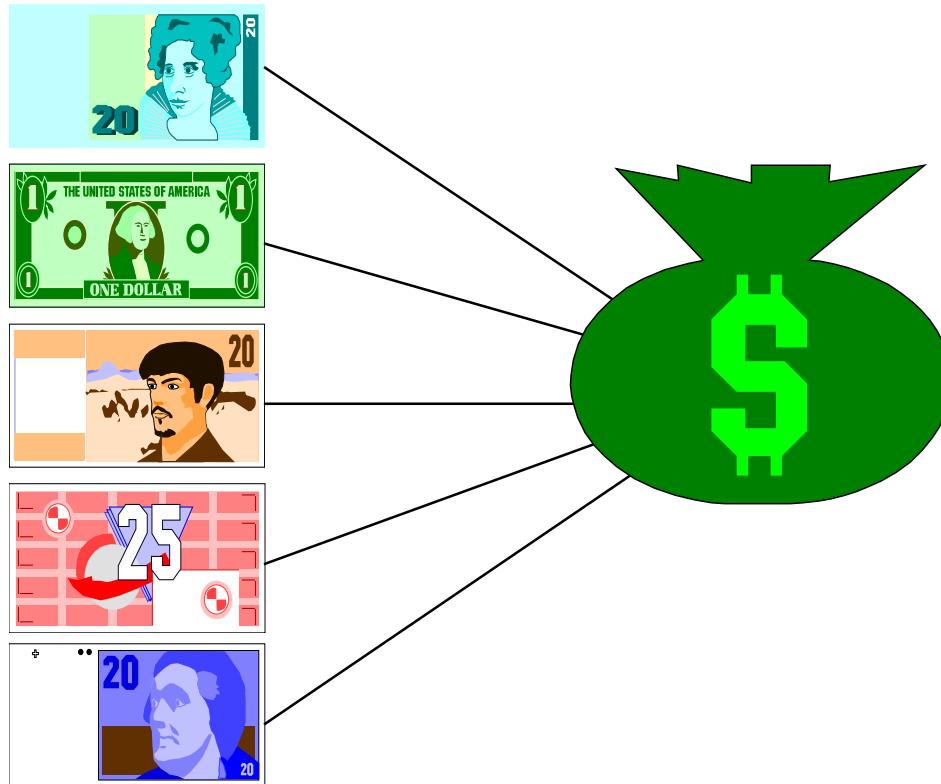
Eine internationale Hotelkette lässt für die Finanzbuchhaltung ein neues Softwaresystem entwickeln. Das Unternehmen ist in vielen Ländern vertreten und muss deshalb mit diversen Währungen umgehen.



Aufgabenstellung

Da die Firma ihren Hauptsitz in den vereinigten Staaten hat, wird der **amerikanischen Dollar als Referenzwährung** verwendet.

Sämtliche Vermögenswerte in den verschiedenen Ländern werden auf dieser Basis miteinander verrechnet.



Aufgabenstellung

Nehmen wir einmal an, verschiedene Posten dieser Aufstellung sind in einer Tabelle zusammengefasst

Währung	Betrag	Wert in US-\$
\$	17.300.000	?
€	1.400.000	?
£	3.000.000	?
...

Es gilt nun, diese Vermögen in die Referenzwährung umzurechnen!

Aufgabenstellung

Nehmen wir einmal an, verschiedene Posten dieser Aufstellung sind in einer Tabelle zusammengefasst

Währung	Betrag	Wert in US-\$
\$	17.300.000	17.300.000
€	1.400.000	1.391.788
£	3.000.000	1.852.072
...

Es gilt nun, diese Vermögen in die Referenzwährung umzurechnen!

Aufgabenstellung

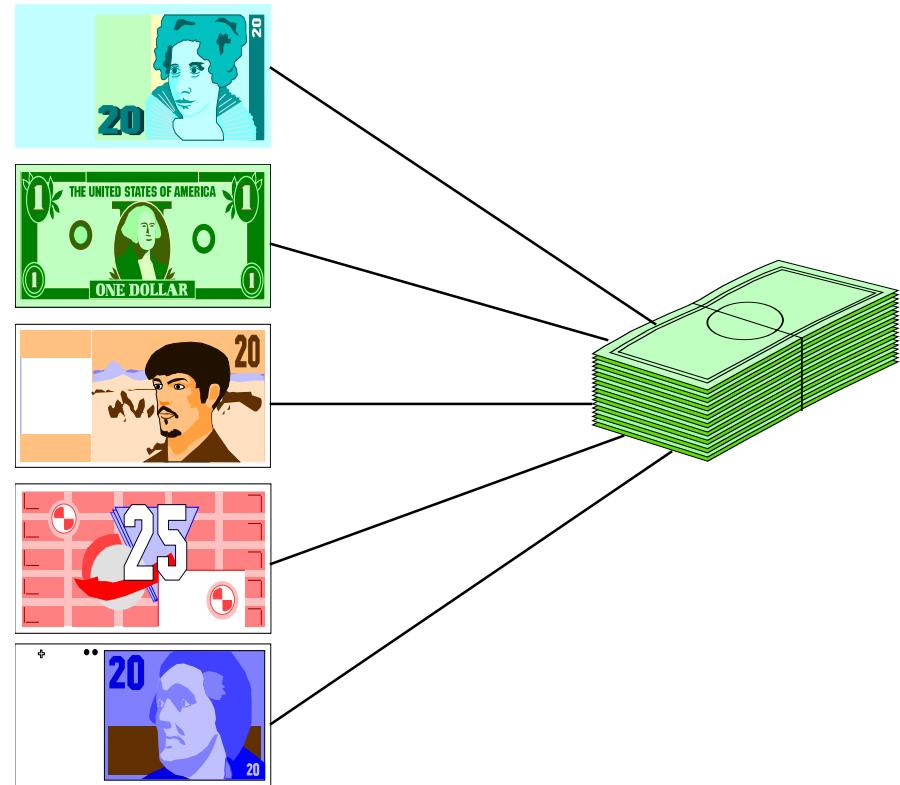
Leider ist der Wechselkurs der verschiedenen Währungen nicht fest; die Devisenkurse ändern sich an den verschiedenen Börsen im Minutentakt. Wie können wir jedoch in Java dafür sorgen, dass die Umrechnung stets nach dem aktuellen Kurs **automatisch** erledigt wird?

Währung	Betrag	Wert in US-\$
\$	17.300.000	17.300.000
€	1.400.000	1.391.788
£	3.000.000	1.852.072
...

Werfen wir erneut einen Blick auf verschiedene Währungen, die wir in dem vorherigen Diagramm kennengelernt haben

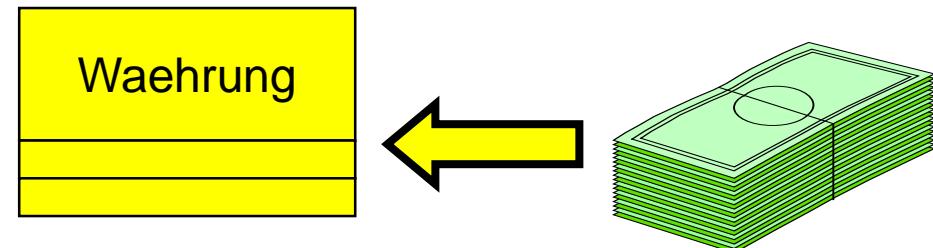
Wir stellen fest, dass jede dieser Währungen

- gültiges Zahlungsmittel ist
- in US-\$ umgerechnet werden kann, wobei sich
- diese Berechnung je nach Währungsart unterschiedlich gestalten kann



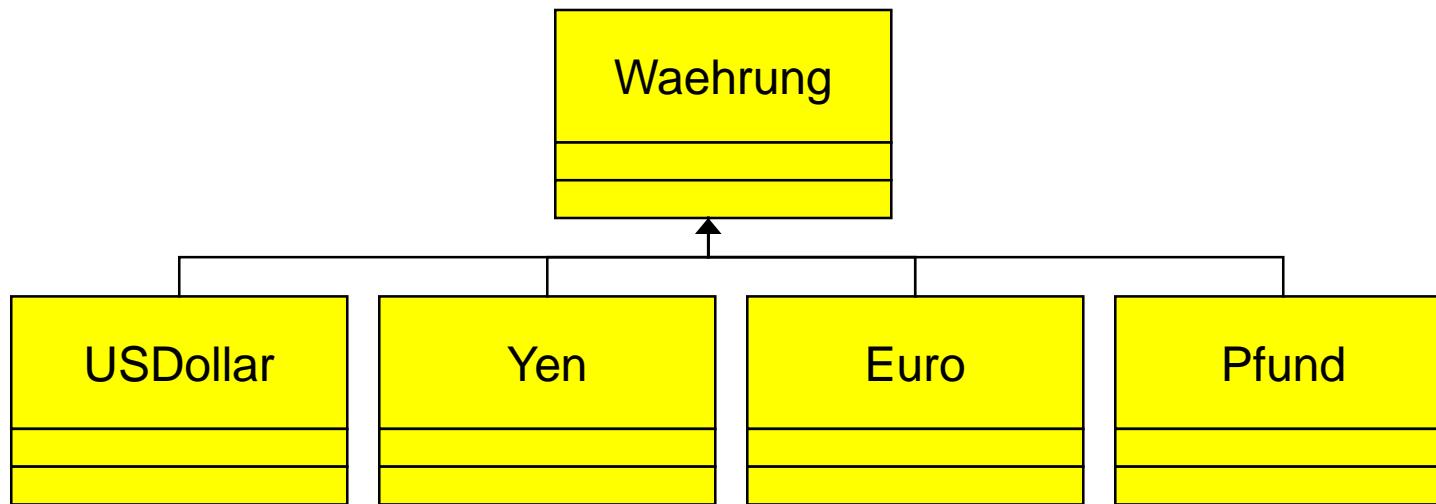
Wenn wir also die verschiedenen Währungen betrachten, so stellen wir bei ihnen gewisse Gemeinsamkeiten fest. Wir machen uns diesen Umstand zunutze und fassen diese in einer Klasse namens **Waehrung** zusammen.

Man bezeichnet diesen Vorgang des Abstrahierens und Zusammenfassens gemeinsamer Eigenschaften auch als **Generalisierung**.



Ein erster Klassenentwurf

Von dieser Klasse **Waehrung** ausgehend, werden wir die verschiedenen anderen konkreteren Währungen entwerfen



Instanzen der Klasse **Waehrung** sollen einen bestimmten Geldbetrag symbolisieren. Der Wert dieses Geldbetrages soll in der Referenzwährung (US-Dollar) ausgegeben werden. Wir sehen hierzu eine entsprechende Methode namens **dollarBetrag** vor.

Da **Waehrung** eine sehr allgemeine Struktur repräsentiert (und kein spezielles Zahlungsmittel wie etwa Dollar, Rubel oder Yen), haben wir keine Ahnung, was für einen Wert die Methode **dollarBetrag** konkret zurückgeben sollte.



Wir beginnen nun mit der Umsetzung unserer Ideen und realisieren die Klasse **Waehrung**.

Instanzen der Klasse **Waehrung** sollen einen bestimmten Geldbetrag symbolisieren. Der Wert dieses Geldbetrages soll in der Referenzwährung (US-Dollar) ausgegeben werden. Wir sehen hierzu eine entsprechende Methode namens **dollarBetrag** vor.

Weitere Details sind für die Klasse (im Moment) nicht vorgesehen, da alle zusätzlich denkbaren Eigenschaften (wie etwa das Verfahren zur Umrechnung bzw. der Wechselkurs) in den verschiedenen speziellen Währungen unterschiedlich modelliert werden müssen.



Wenn wir unsere Klasse jedoch derart allgemein halten wollen, stoßen wir bald auf ein Problem:

Da **Waehrung** eine sehr allgemeine Struktur repräsentiert (und kein spezielles Zahlungsmittel wie etwa Dollar, Rubel oder Yen), wissen wir nicht, was für einen Wert die Methode **dollarBetrag** konkret zurückgeben sollte.

Auch macht es im Moment nur wenig Sinn, die Klasse **Waehrung** zu instantiiieren. Diese steht schließlich für das allgemeine Konzept „Geld“ und nicht speziell für eine konkrete Realisierung...

Waehrung

+ dollarBetrag():double

Wir lösen dieses Problem, indem wir die Klasse als **abstrakt** deklarieren.

Im Gegensatz zu unserer bisherigen Klassendefinition können wir hier (ähnlich wie in UML) eine Ansammlung von Instanzmethoden definieren, ohne die Methoden explizit ausformulieren zu müssen (man spricht in diesem Zusammenhang auch von der Definition einer **Schnittstelle** – **bitte trotz gleicher Wortwahl nicht verwechseln mit dem später einzuführenden ähnlichen Konstrukt `interface`**).

Waehrung

+ dollarBetrag():double

Wir lösen dieses Problem, indem wir die Klasse als **abstrakt** deklarieren.

Im Gegensatz zu unserer bisherigen Klassendefinition können wir hier (ähnlich wie in UML) eine Ansammlung von Instanzmethoden definieren, ohne die Methoden explizit ausformulieren zu müssen (man spricht in diesem Zusammenhang auch von der Definition einer **Schnittstelle** – **bitte trotz gleicher Wortwahl nicht verwechseln mit dem später einzuführenden ähnlichen Konstrukt `interface`**).

Natürlich würde es wenig Sinn machen, eine derart „unvollständige“ Klasse zu instantiiieren. Aus diesem Grund ist es auch nicht möglich, aus abstrakten Klassen direkt Objekte zu bilden.



Wir übertragen diese Überlegungen nun auf unser Klassenmodell. Wenn wir in UML eine Klasse als abstrakt markieren wollen, schreiben wir ihren Namen in *Kursivschrift*.

Auch unsere Methode **dollarBetrag** soll abstrakt gehalten (also nicht konkret ausformuliert) werden. Wir setzen sie deshalb ebenfalls *kursiv*.

Es stellt sich allerdings noch die Frage, wie sich dieses Modell in der Sprache Java umsetzen lässt...

Waehrung

+ *dollarBetrag():double*

Werfen wir also einen Blick auf die Realisierung in Java.

Da die Methode **dollarBetrag** nun abstrakt ist, muss diese nicht weiter ausformuliert werden. Wir beenden sie **statt** mit dem **Rumpf** ({...}) mit einem einfachen Semikolon.

```
public abstract class Waehrung {  
    public abstract double dollarBetrag();  
}
```



Hinweis

Nicht jede Methode, die innerhalb einer abstrakten Klasse definiert wird, muss zwangsläufig auch abstrakt formuliert werden.

Andererseits muss jedoch jede Klasse, die eine abstrakte Methode enthält, zwangsläufig auch abstrakt sein!

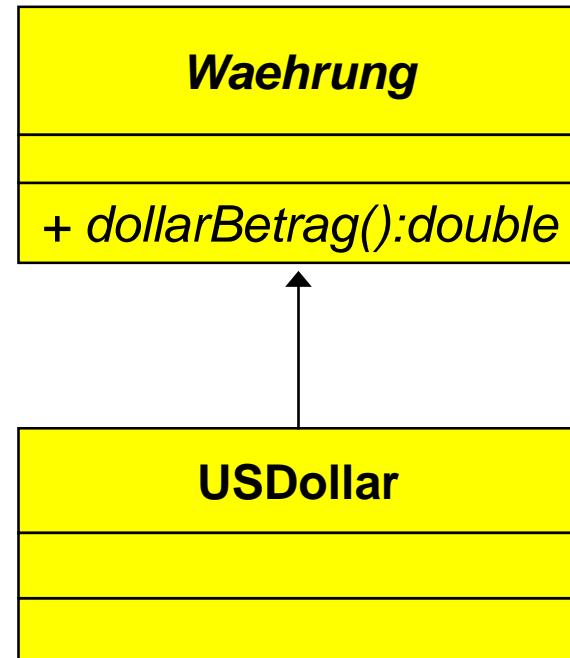
```
public abstract class Waehrung {  
    public abstract double dollarBetrag();  
}
```

Waehrung

+ *dollarBetrag():double*

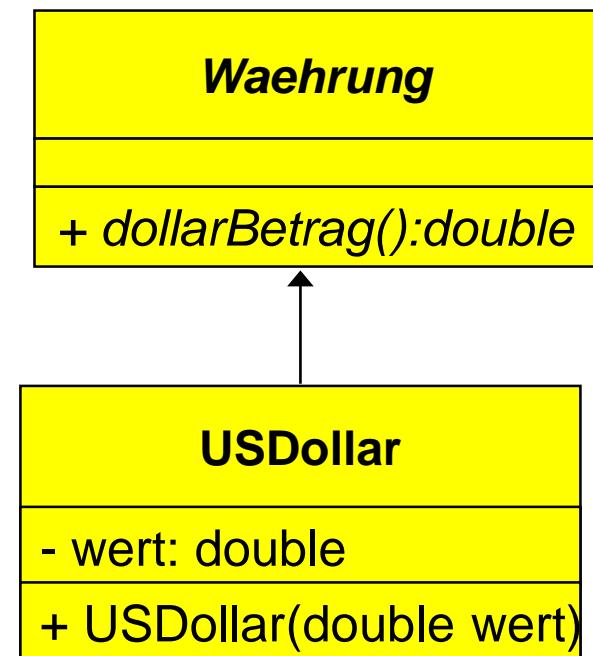
Im nächsten Schritt wollen wir unsere abstrakte Klasse nutzen, um eine konkrete Währung zu realisieren.

Hierzu beginnen wir mit dem einfachsten Fall - dem amerikanischen Dollar.



Instanzen der Klasse **USDollar** sollen einen Geldbetrag in der amerikanischen Währung repräsentieren. Um diesen Wert im Objekt hinterlegen zu können, definieren wir eine (**private**) Instanzvariable namens **wert**.

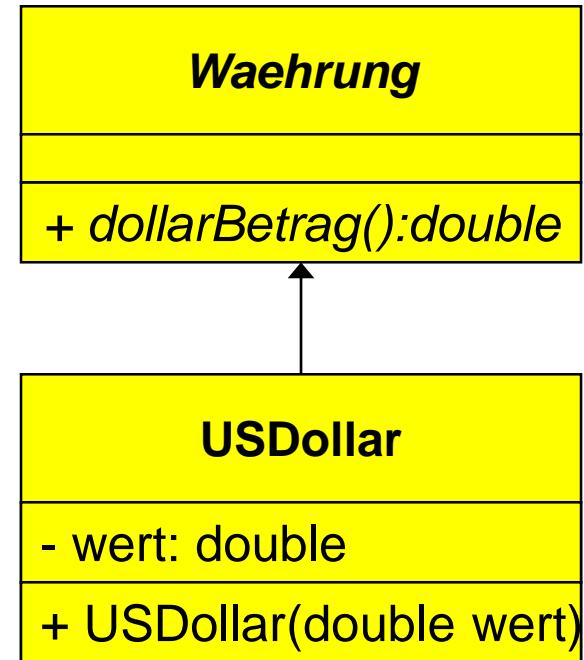
Die Belegung der Instanzvariable mit ihrem Wert wollen wir (wie schon in der Klasse **Student** geschehen) im Konstruktor vornehmen. Wir nehmen einen entsprechenden Eintrag in unserem UML-Modell vor.



Wir werden nun versuchen, das Klassenmodell in Java umzusetzen.

Der unten angegebene Quelltext setzt hierbei jenen Teil des Modells um, den wir mit unseren bisherigen Kenntnissen in Java formulieren können. Auf die Beziehung zwischen den Klassen **Waehrung** und **USDollar** sind wir hierbei noch nicht eingegangen...

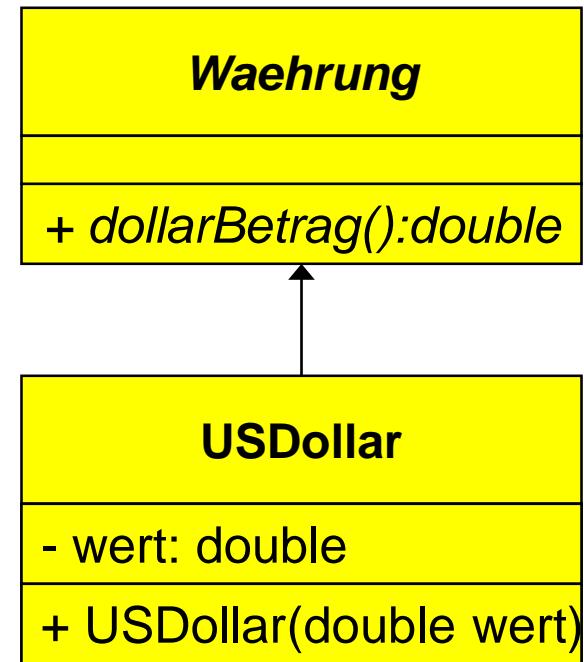
```
public class USDollar {  
  
    private double wert;  
  
    public USDollar(double wert) {  
        this.wert = wert;  
    }  
  
}
```



Die Klassen **Waehrung** und **USDollar** stehen hierbei in einer „ist-ein“-Beziehung. Man sagt, **Waehrung** ist **Superklasse** von **USDollar**; **USDollar** ist **Subklasse** von **Waehrung**.

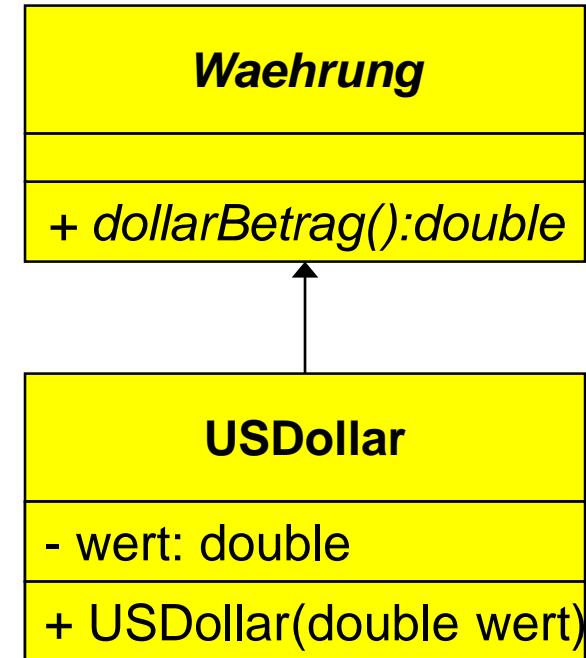
Eine andere Sprechweise ist auch, die Klasse **USDollar** erweitert (engl. **extends**) die Klasse **Waehrung**.

```
public class USDollar extends Waehrung {  
  
    private double wert;  
  
    public USDollar(double wert) {  
        this.wert = wert;  
    }  
}
```



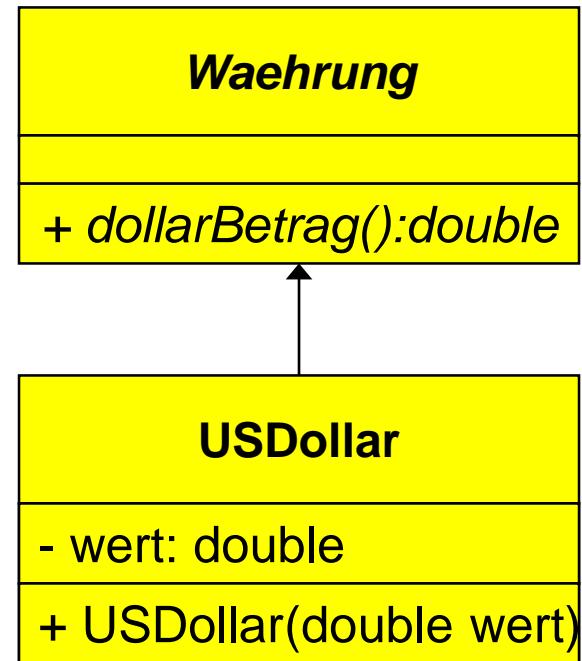
Die Formulierung („**USDollar** erweitert/extends **Waehrung**“) motiviert die Schreibweise, in der die Beziehung zwischen den beiden Klassen im Java-Programm formuliert wird. Das Schlüsselwort „**extends**“ dient als Mittel, um eine Verwandtschaft unter Klassen auszudrücken.

```
public class USDollar extends Waehrung {  
  
    private double wert;  
  
    public USDollar(double wert) {  
        this.wert = wert;  
    }  
}
```



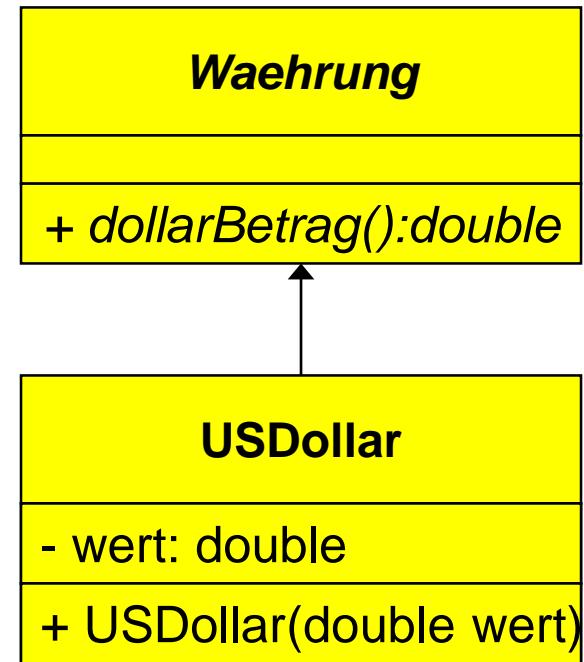
Die Klasse **USDollar** erbt von ihrer Superklasse (**Waehrung**) sämtliche **Eigenschaften** (also **alle Variablen** und **alle Methoden**). Eine Instanz der Klasse **USDollar** besitzt somit also automatisch auch eine Methode **dollarBetrag**, da diese für die Superklasse **Waehrung** vereinbart wurde.

```
public class USDollar extends Waehrung {  
  
    private double wert;  
  
    public USDollar(double wert) {  
        this.wert = wert;  
    }  
}
```



Nun befinden wir uns jedoch in der Situation, daß **Waehrung** eine abstrakte Klasse ist. Die Methode **dollarBetrag** ist ebenfalls abstrakt formuliert, so dass unsere nicht-abstrakte Klasse **USDollar** eine abstrakte Methode besitzt. Der Compiler honoriert dieses Verhalten mit einem **Übersetzungsfehler!**

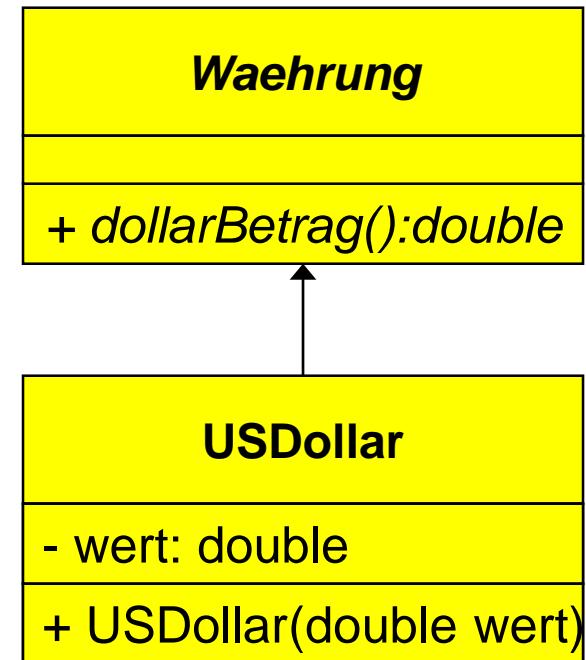
```
public class USDollar extends Waehrung {  
  
    private double wert;  
  
    public USDollar(double wert) {  
        this.wert = wert;  
    }  
  
}
```



Anstatt einer reinen Schnittstellenvorgabe spendieren wir der Klasse also eine gültige Methode inklusive eines funktionsfähigen Rumpfes.

Man bezeichnet diesen Vorgang auch als **Überschreiben** (override)

```
public class USDollar extends Waehrung {  
  
    private double wert;  
  
    public USDollar(double wert) {  
        this.wert = wert;  
    }  
  
    public double dollarBetrag() {  
        return wert;  
    }  
}
```



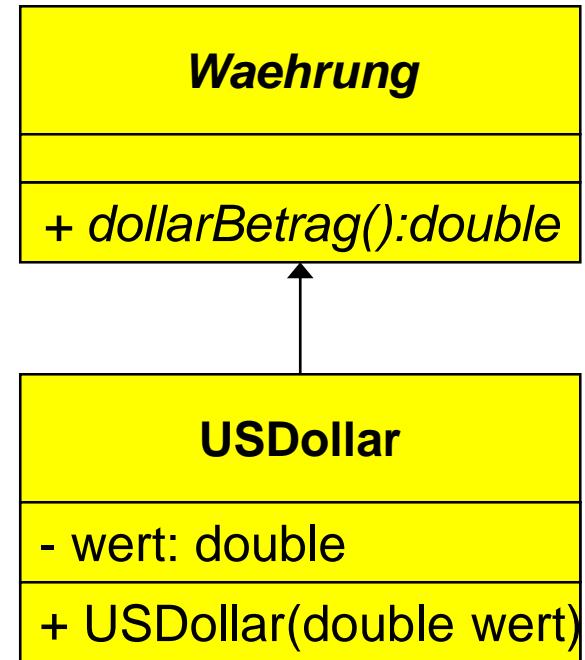
Wir können unsere Klasse **USDollar** nun wie gewohnt in Programmen einsetzen. Um beispielsweise ein Objekt „im Werte“ von \$0.25 zu erzeugen, genügt die Programmzeile

```
USDollar quarter = new USDollar(0.25);
```

Wegen der Beziehung zwischen den Klassen können wir eine Instanz der Klasse **USDollar** aber auch als ein beliebiges Waehrungsobjekt behandeln. Die Zeile

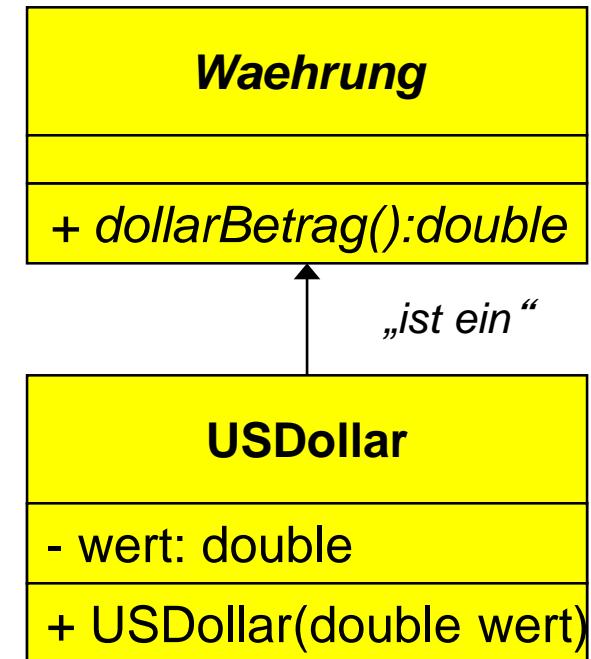
```
Waehrung quarter = new USDollar(0.25);
```

wäre somit ebenfalls korrekt!

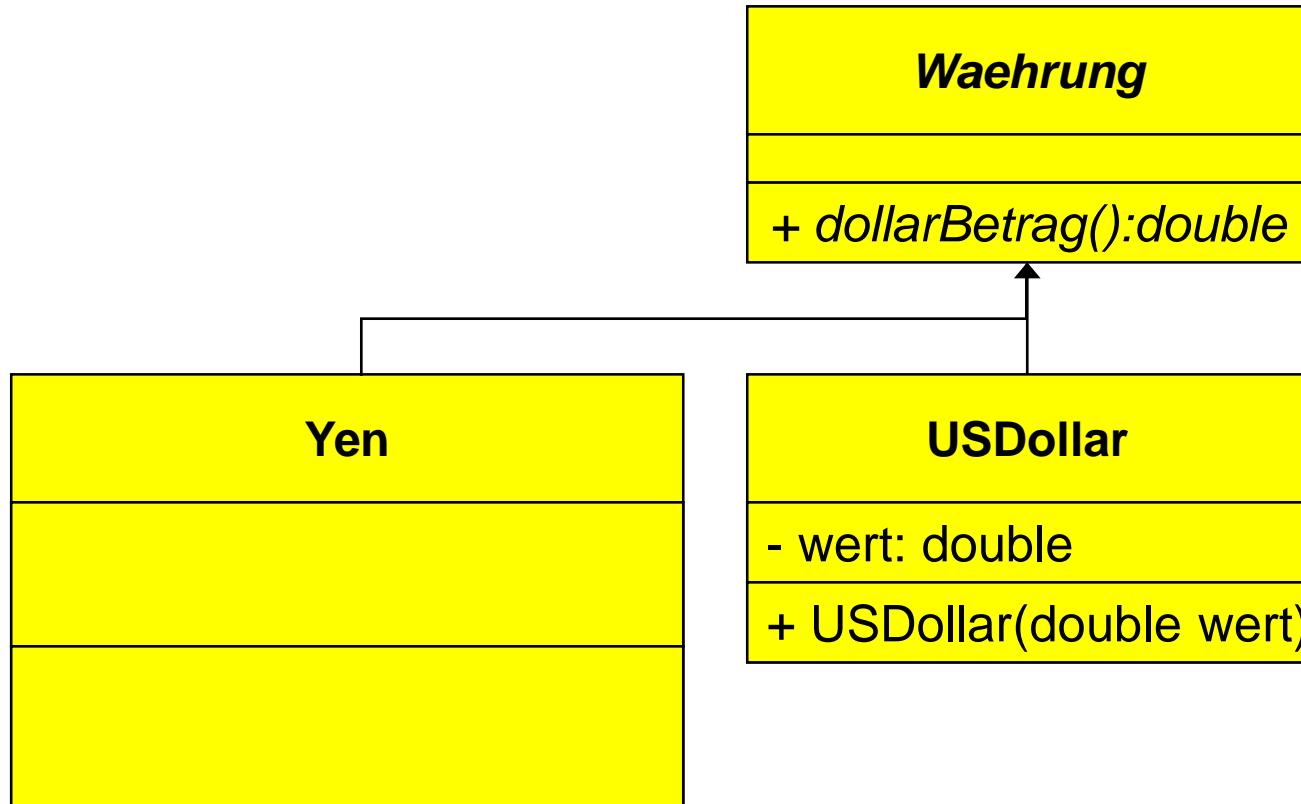


Wir haben mit der Klasse **USDollar** eine Subklasse unserer Klasse **Waehrung** geschaffen. Wegen der verwandschaftlichen Beziehung zwischen den beiden Klassen („**USDollar extends Waehrung**“) haben wir festgestellt, dass ein Objekt der Subklasse **USDollar** auch als ein Objekt der Superklasse **Waehrung** aufgefasst werden kann.

*Wir wissen allerdings noch nicht, welchen **praktischen** Nutzen wir von dieser Beziehung haben!*



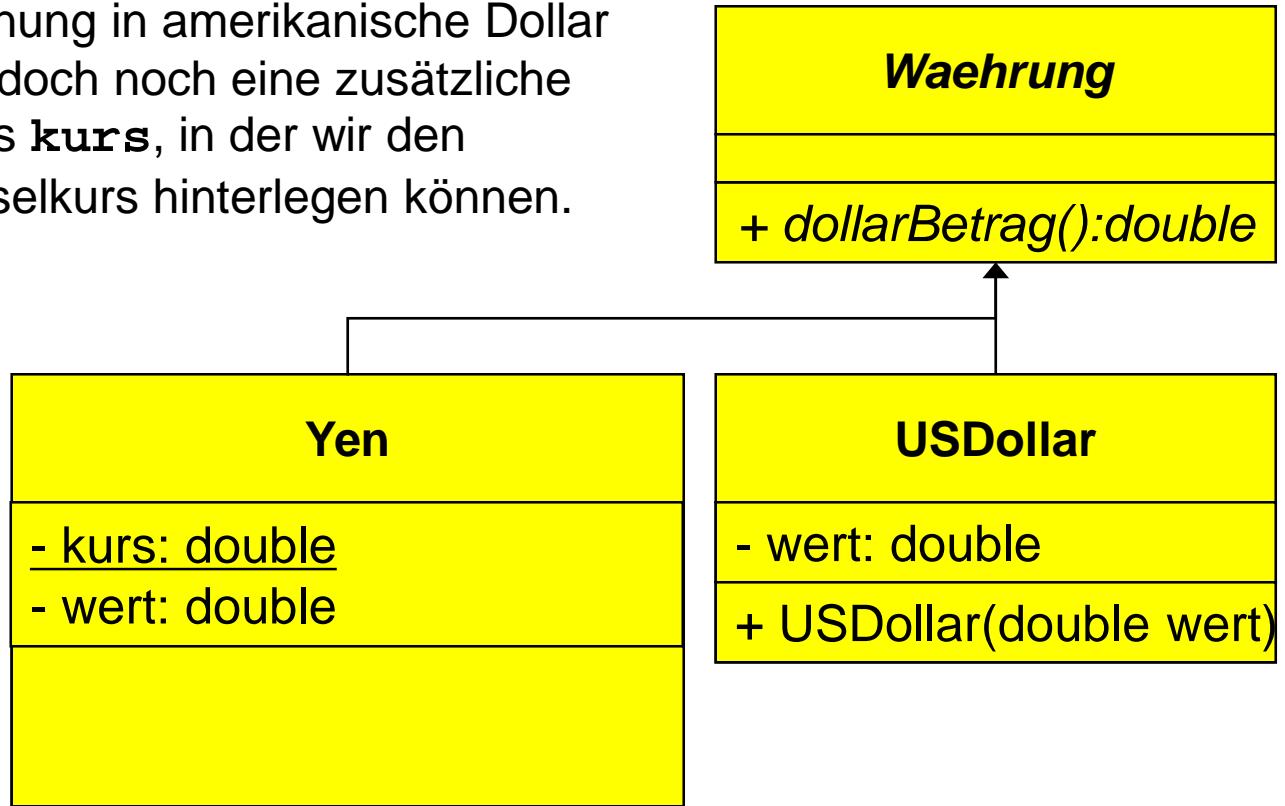
Wir werden diesen Nutzen deshalb an einem Beispiel verdeutlichen.
Zuvor müssen wir allerdings noch eine zweite Subklasse von **Waehrung** bilden: wir modellieren das Zahlungsmittel **Yen**.



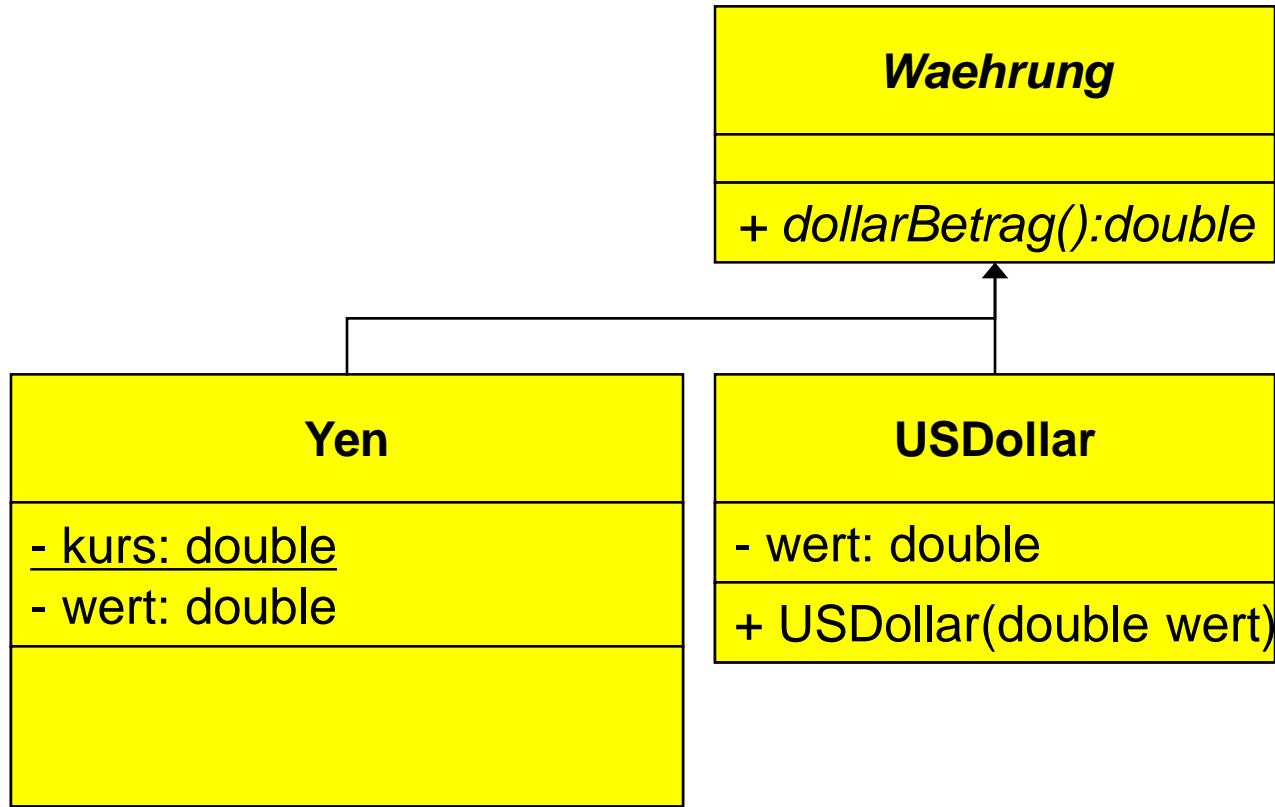
Vererbung in der Praxis

Ähnlich wie bei der Klasse `USDollar` wollen wir den Wert eines Währungsobjektes der Kategorie `Yen` in einer privaten Instanzvariablen namens `wert` speichern.

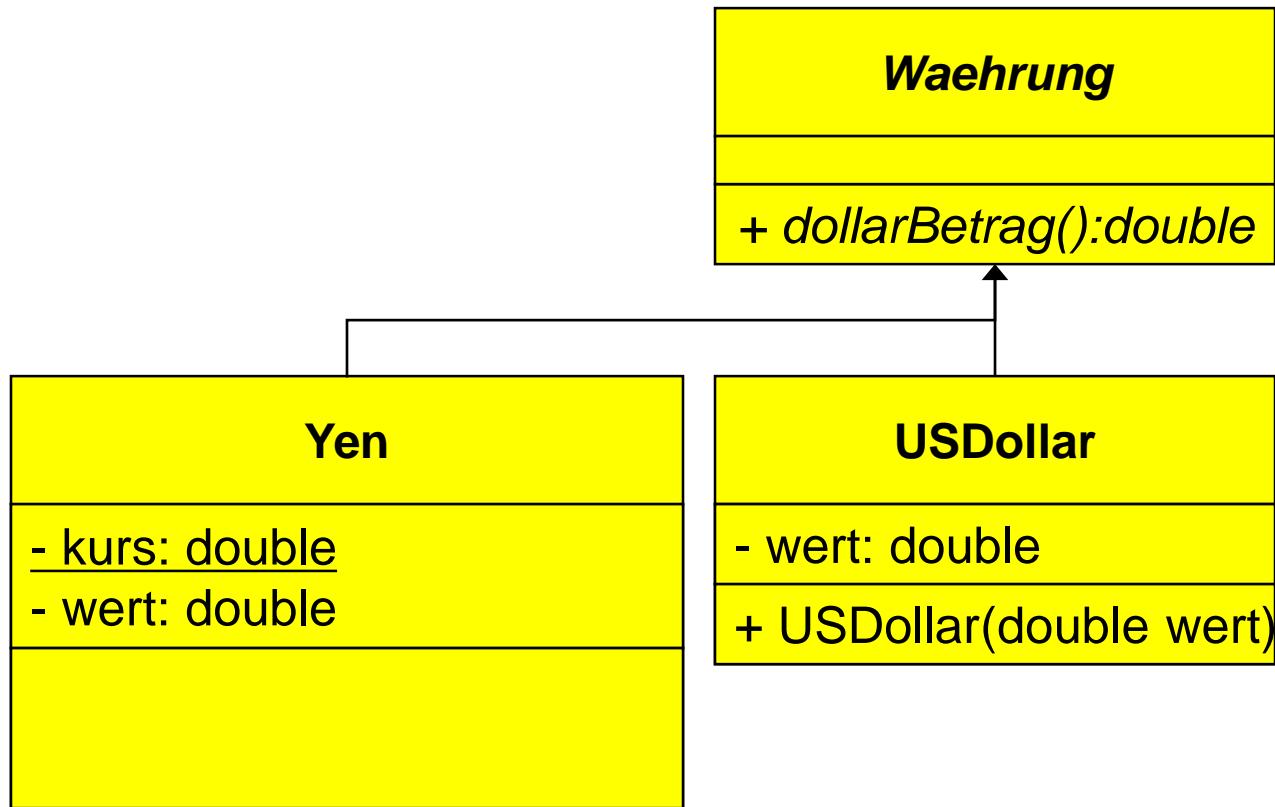
Für die Umrechnung in amerikanische Dollar benötigen wir jedoch noch eine zusätzliche Variable namens `kurs`, in der wir den aktuellen Wechselkurs hinterlegen können.



Der Wechselkurs für eine Währung zu einen bestimmten Zeitpunkt hängt nicht von einem speziellen Geldobjekt ab - er ist für alle Instanzen der Klasse **Yen** identisch. Es ist deshalb ratsam, die Variable **kurs** **statisch** anzulegen (also als Klassenvariable).



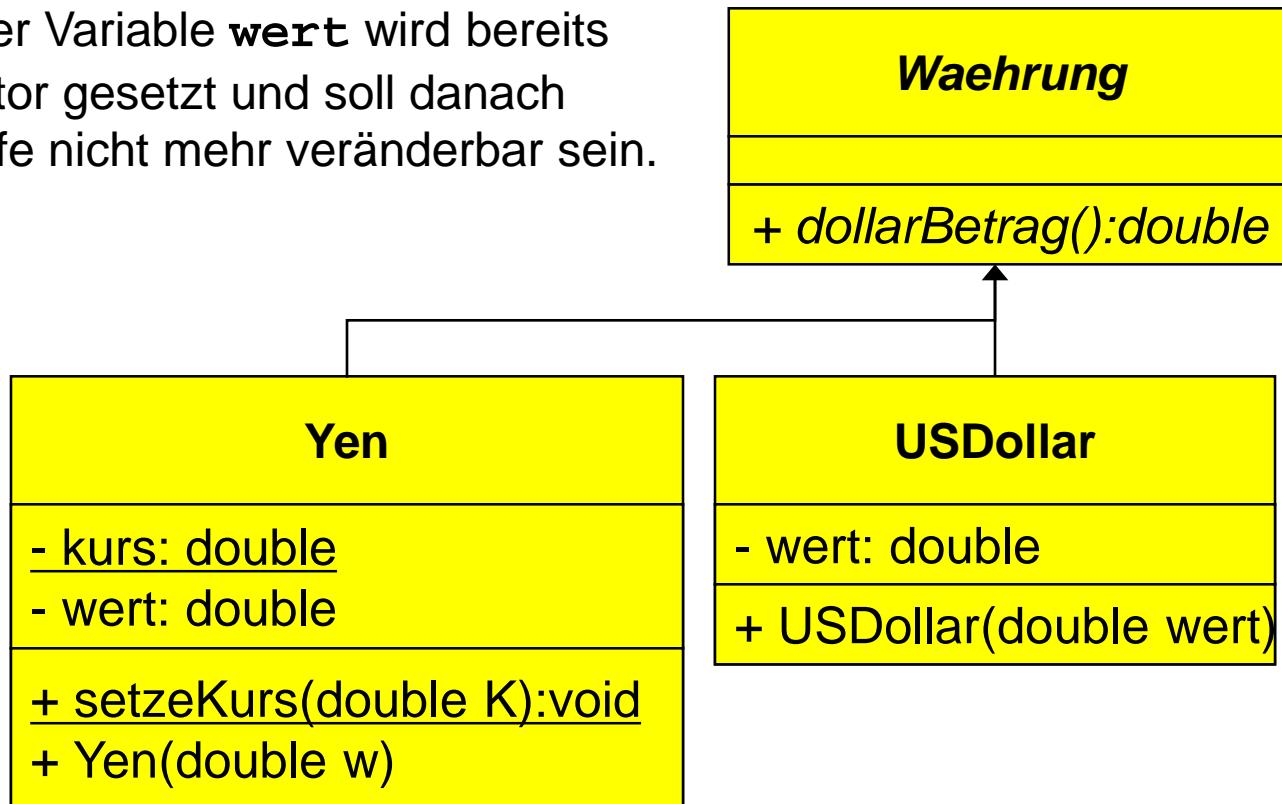
Dieses Vorgehen hat den Vorteil, dass der Kurs einer speziellen Währung global gesetzt werden kann - eine Aktualisierung kann somit stattfinden, ohne dass hierzu sämtliche Objekte überprüft werden müssen...



Vererbung in der Praxis

Natürlich benötigen wir auch eine Möglichkeit, die (**private**) Klassenvariable auf einen bestimmten Wert zu setzen. Hierzu definieren wir eine Klassenmethode **setzeKurs**, die dieses Problem in die Hand nimmt.

Der Inhalt der Variable **wert** wird bereits im Konstruktor gesetzt und soll danach durch Zugriffe nicht mehr veränderbar sein.

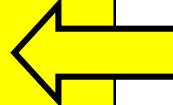


```
public class Yen extends Waehrung {  
  
    private static double kurs;  
    private double wert;  
  
    /** Konstruktor */  
    public Yen(double w) {wert = w;}  
  
    /** Kurs setzen */  
    public static void setzeKurs(double K) {  
        kurs = K;  
    }  
  
    /** Wert in US-Dollar */  
    public double dollarBetrag() {  
        return wert * kurs;  
    }  
}
```

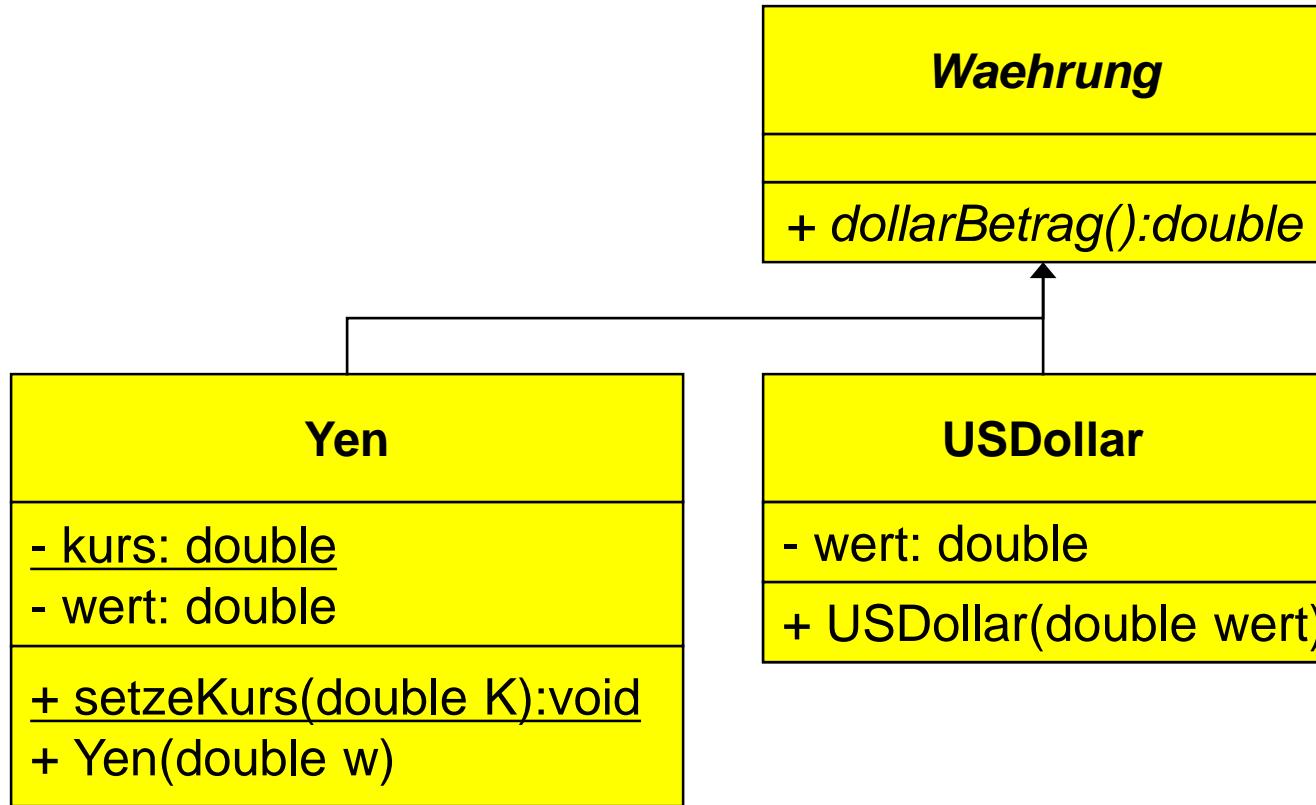
Werfen wir einen Blick auf die Umsetzung des Modells in Java.

Yen

- kurs: double
- wert: double
+ setzeKurs(double K):void
+ Yen(double w)



Zurück zu unserem Softwareprojekt. Unsere Hotelkette möchte mit ihrer Finanzbuchhaltung unter anderem ihre **Steuererklärung** machen können. Die Firma wird in den USA veranlagt.



Nach einigen schlaflosen Nächten kommen die Steuerberater zu folgendem Schluss: Die Firma muss eine Steuer von exakt 8 Prozent auf ihr gesamtes Barvermögen zahlen.

Diese Summe soll von unserem Java-Programm automatisch berechnet und bei Fälligkeit an die Finanzbehörde überwiesen werden.

Unglücklicherweise besitzt die Firma Vermögen in den verschiedensten Devisen, so dass die Beträge zu dem entsprechenden Datum alle umgerechnet werden müssen.

Hätten wir es nur mit **einer Klasse** von Objekten (also einer Währungsart) zu tun, wäre das Problem schnell gelöst:

```
public static double berechneSteuer(Waehrung[] geld) {  
    double summe = 0;                                // der Gesamtbetrag  
    for (int i=0; i < geld.length; i++)      // wird in einer Schleife  
        summe += geld[i].dollarBetrag();           // summiert und danach  
    double steuer = summe * 0.08;                  // mit 8 Prozent  
    return steuer;                                  // multipliziert  
}
```

Alle Geldmittel, die in dem Feld **geld** gespeichert sind, werden summiert - und danach 8% dieser Summe berechnet.

Aufgrund ihrer verwandtschaftlichen Beziehungen können wir ohne Probleme ein „gemischtes“ Feld erstellen. Jedes Währungsobjekt lässt sich in diesem Feld unterbringen:

```
Waehrung[] geld = new Waehrung[3];
geld[0] = new USDollar(2500);
geld[1] = new Yen(200000);
geld[2] = new USDollar(20);

double steuer = berechneSteuer(geld);
```

Dank der Beziehungen zwischen Sub- und Superklasse stellt also das Array **geld** ein gültiges Feld von **Waehrung**-Objekten dar. Wir können unsere Methode **berechneSteuer** auf dieses Feld anwenden.

Fazit

USDollar und **Yen** sind zwei verschiedene Währungen mit unterschiedlichen Eigenschaften.

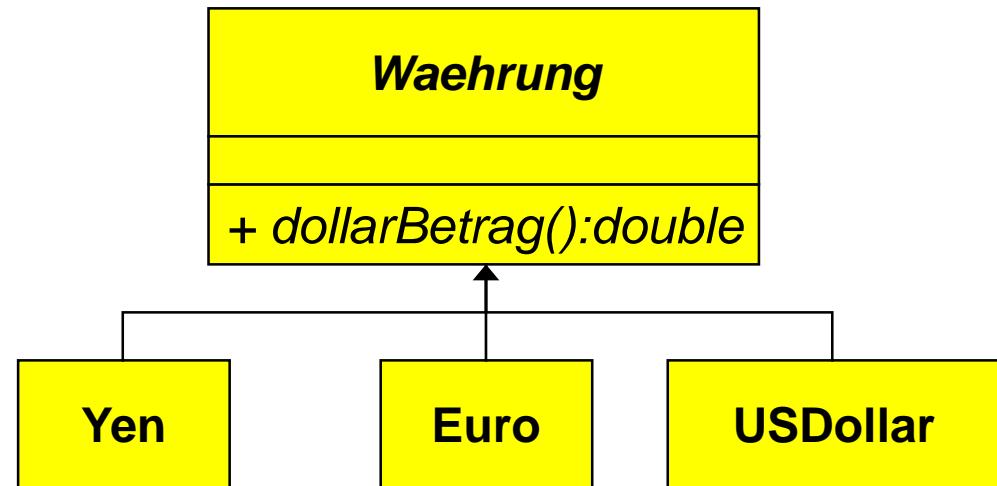
Trotz dieser Unterschiede haben die beiden Klassen jedoch Gemeinsamkeiten in ihrer Schnittstelle - nämlich die abstrakte Superklasse **Waehrung**. Obwohl beide Subklassen diese abstrakte **Waehrung** völlig unterschiedlich erweitert haben (vgl. die beiden Implementierungen von **dollarBetrag**), reichen die gemeinsamen Eigenschaften jedoch bereits aus, um Operationen von der abstrakten Klasse auf beide Kindklassen zu übertragen.

Die Methode **berechneSteuer** muss nur ein einziges Mal verfasst werden - egal, wieviele verschiedene Währungen wir implementieren!

Wir haben am Beispiel einer internationalen Hotelkette eine Hierarchie von Klassen entwickelt, mit deren Hilfe wir verschiedene Währungen miteinander in Verbindung bringen und automatisch in eine Referenzwährung (den US Dollar) umrechnen konnten.

Als Basis verwendeten wir eine Klasse namens **Waehrung**, die wir als **abstrakt** (Schlüsselwort: **abstract**) erklärt hatten.

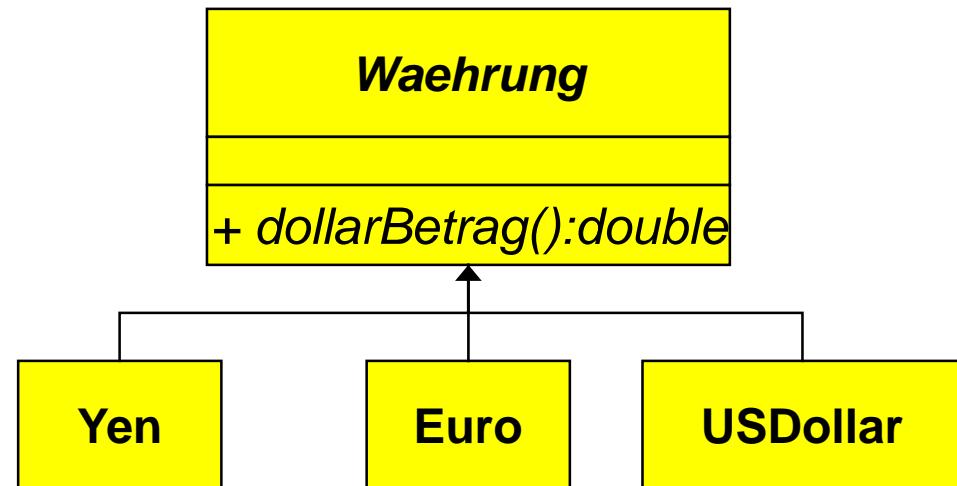
Die Klassen **Yen**, **Euro** und **USDollar** erweitern die abstrakte Klasse.

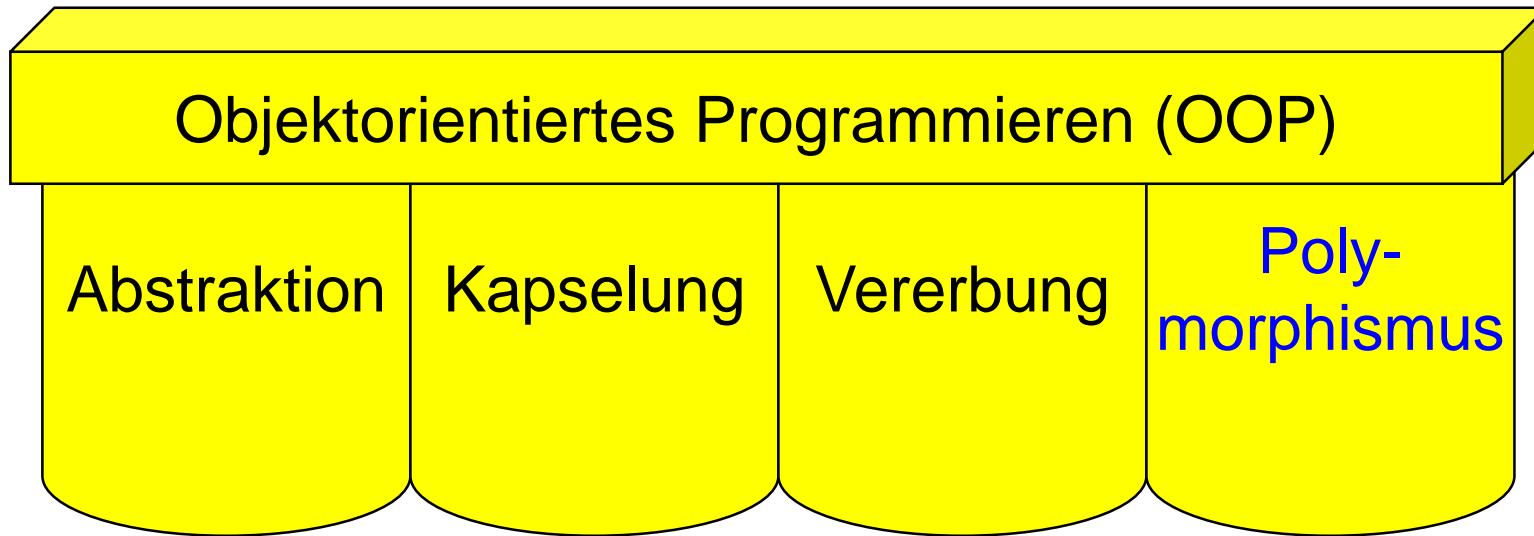


Die Methode `dollarBetrag` wird in jeder Subklasse überschrieben (overriden), d.h. durch eine andere, spezielle Methode ersetzt.

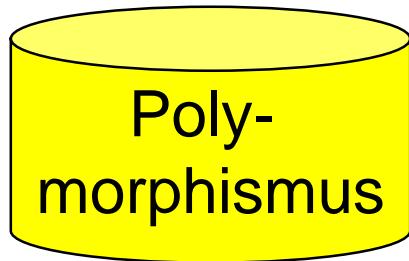
Obwohl sich die Implementierungen hierbei unterscheiden, können doch alle Objekte über die einheitliche Schnittstelle angesprochen werden, welche sie von der Klasse `Waehrung` erben.

Die neuen Methoden nahmen den Platz ihres abstrakten Vorgängers ein - sie wurden an seiner statt aufgerufen.





Zur Wiederholung



Die Möglichkeit, Methoden innerhalb der Objekthierarchie zu **Überschreiben** (override) und somit individuelles Verhalten verschiedener Klassen auf dieselbe Anweisung zu modellieren.

Mit Hilfe des Polymorphismus kann der Entwickler:

- ohne Probleme **Besonderheiten** bei den verschiedenen Subklassen **berücksichtigen**,
- **einheitliche Schnittstellen** erzeugen und somit
- Entwicklungszeit sparen und
- eine Vielzahl von Fehlerquellen ausschalten.

Beispiel

Konzert im Hinterhof

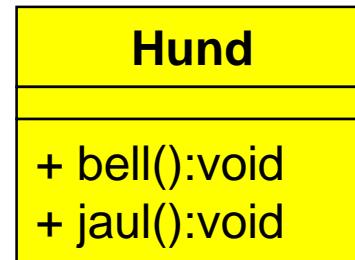
Pünktlich zu jedem Vollmond trifft sich eine Meute von vier braven Haushunden, um „den Wolf“ in sich zu spüren. A capella jaulen sie dann gemeinsam den Vollmond an.

Der erste Hund ist eine lebensfrohe Promenadenmischung, die man keiner besonderen Rasse zuordnen kann. Hund Nr. 2 ist ein kleiner Pekinese - er sorgt für den Sopran. Der dritte Hund ist ein Bernhardiner mit tiefer, sonorer Bassstimme. Der letzte im Bunde gehört zur seltenen Gattung der Echohunde: er klingt zwar wie jeder andere Hund, pflegt sich aber stets zu wiederholen.

Jeder der vier Hunde tritt hervor, um einmal zu jaulen, zu bellen und dann wieder zu jaulen. Danach geben die Tiere bis zum nächsten Vollmond Ruhe.

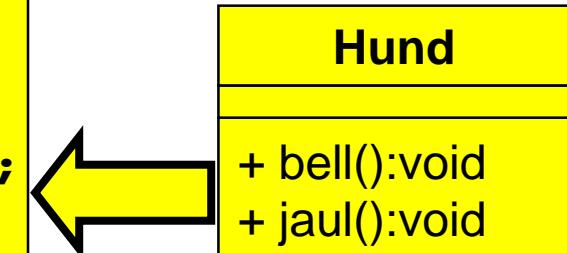
Beispiel

- Da wir natürlich nicht bis zum nächsten Vollmond warten wollen, möchten wir das Konzert auf dem Computer simulieren.
- Zu diesem Zweck entwerfen wir eine Klasse **Hund**, die ein Mitglied aus dem stimmkräftigen Quartett darstellen soll.
- Die Darstellung des Gesangs (bellen und jaulen) soll auf dem Bildschirm erscheinen.



Beispiel

```
public class Hund {  
  
    public void bell() {  
        System.out.println("WAU    ");  
    }  
  
    public void jaul() {  
        System.out.println("JAUL    ");  
    }  
}
```



```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute = new Hund[4];  
        meute[0]=new Hund();  
        meute[1]=new Hund();  
        meute[2]=new Hund();  
        meute[3]=new Hund();  
        for (int i=0; i < meute.length; i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```

Beispiel

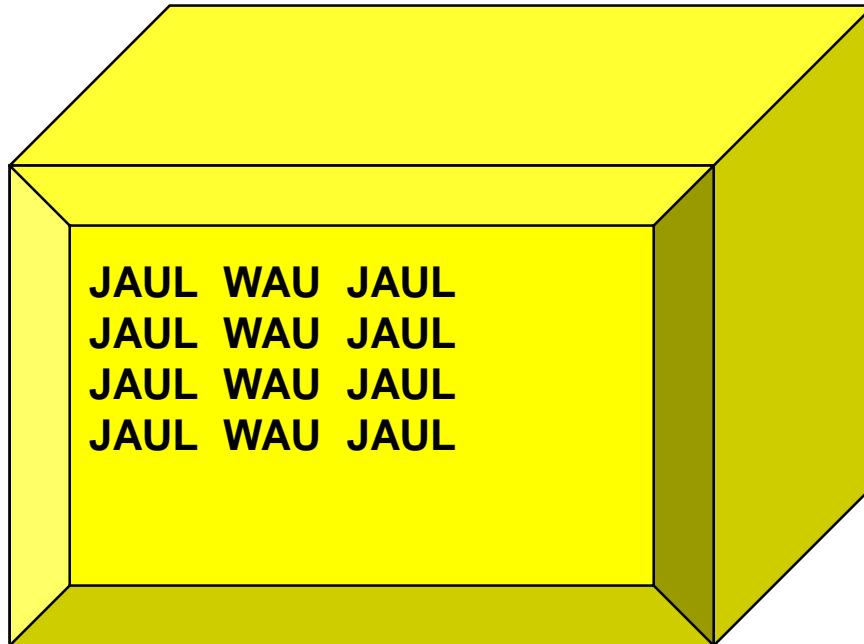
Hund

+ bell():void
+ jaul():void

Leider „klingt“ die Ausgabe auf dem Bildschirm im Moment noch wenig realistisch.

Grund hierfür ist ein Mangel an Individualität. Ein Bernhardiner sollte nicht so klingen wie ein Pekinese!

Wir müssen unser Klassenmodell also verfeinern...



Beispiel

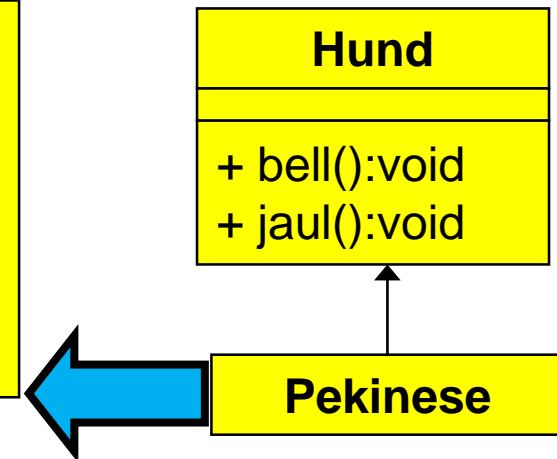
Hund
+ bell():void
+ jaul():void

Beispiel

Für die Promenadenmischung (Hund Nr. 1) können wir auch weiterhin die Klasse **Hund** verwenden - diese ist eh nicht weiter zu klassifizieren.

Wir kümmern uns deshalb um den zweiten Hund und entwerfen eine Klasse namens **Pekinese**.

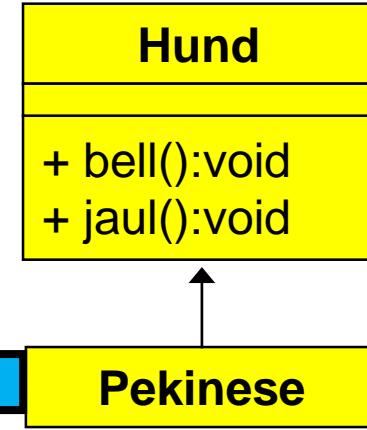
```
public class Pekinese extends Hund {  
}
```



Beispiel

Wir wollen also, dass sich Instanzen der Klasse **Pekinese** beim Aufruf der Methode **bell()** anders verhalten als sonstige Instanzen der Klasse **Hund**. Wir müssen die Methode **bell()** für diese Klasse also **überschreiben**:

```
public class Pekinese extends Hund {  
  
    public void bell() {  
        System.out.println("Wiff ");  
    }  
}
```

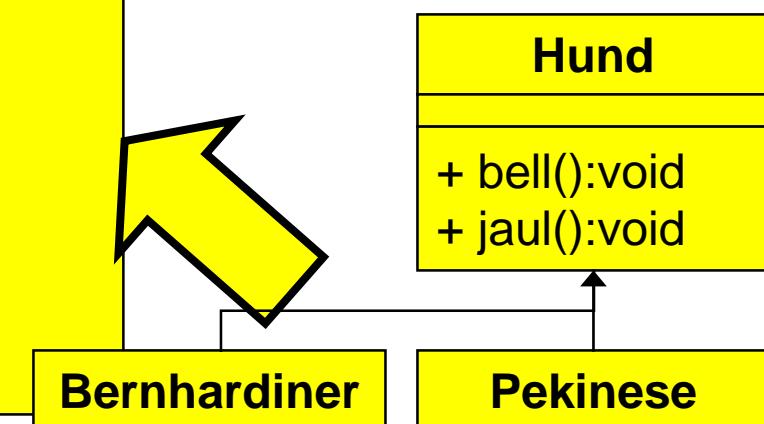


Beispiel

Analog hierzu entwerfen wir für Hund Nr. 3 eine Klasse namens **Bernhardiner**.

```
public class Bernhardiner extends Hund {  
  
    public void bell() {  
        System.out.print("WRRRUFF!    ");  
    }  
  
    public void jaul() {  
        System.out.print("JAUUUUUUUL    ");  
    }  
}
```

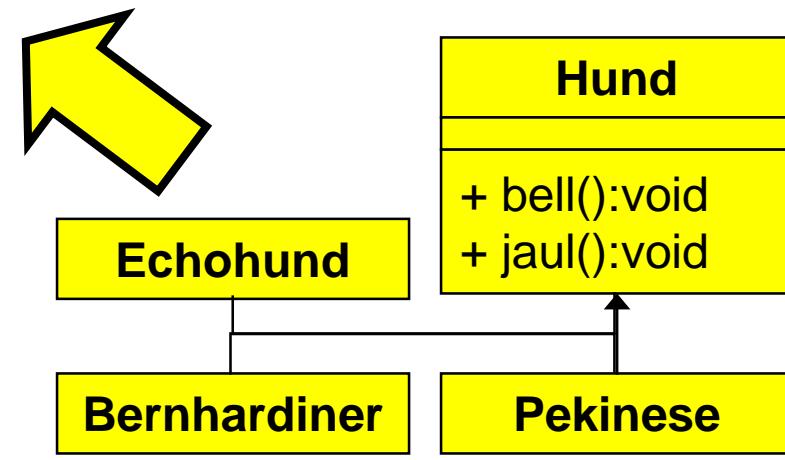
Java



Beispiel

Da der **Echohund** wie ein normaler Hund bellt (und jault), seine Laute allerdings immer wiederholt, liegt der Gedanke nahe, die Original-Methoden einfach zweimal aufzurufen.

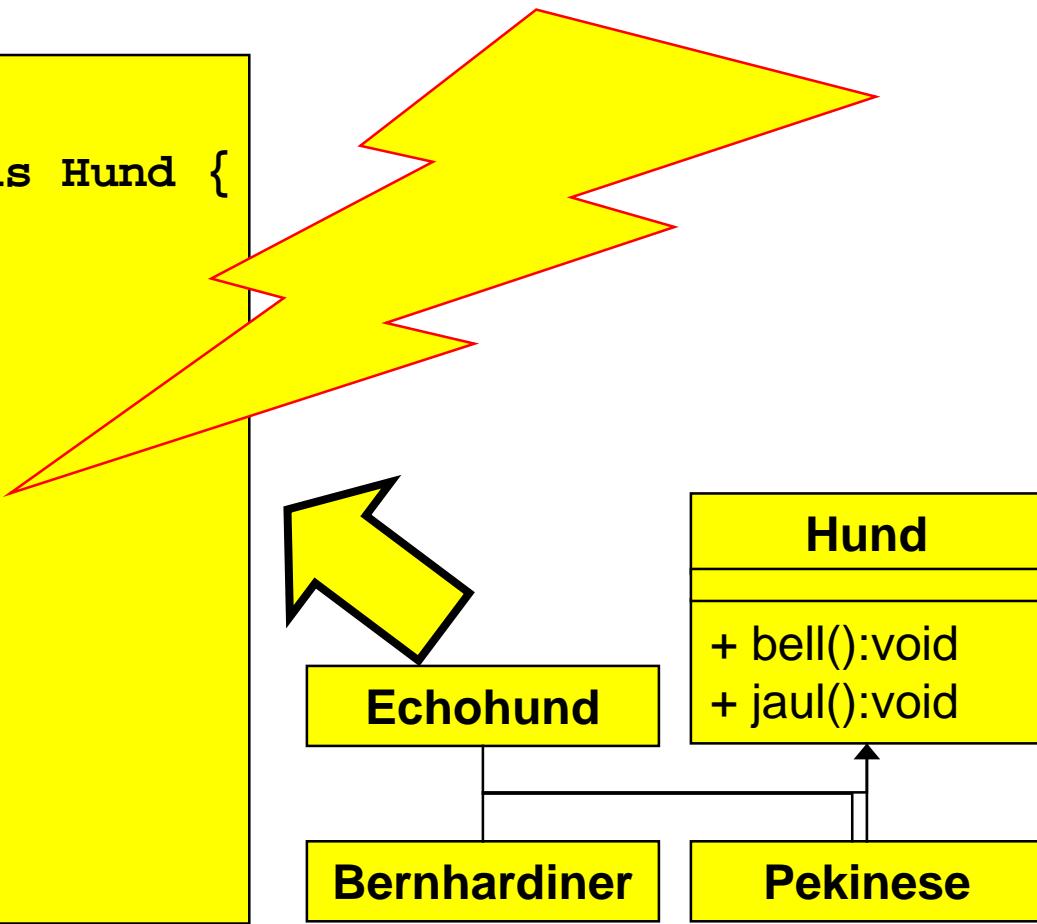
```
public class Echohund extends Hund {  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
}
```



Beispiel

Würden wir jedoch bei einer derart gestalteten Klasse die Methoden `bell()` oder `jaul()` aufrufen, so würde das Programm mit einer **Fehlermeldung** abbrechen.

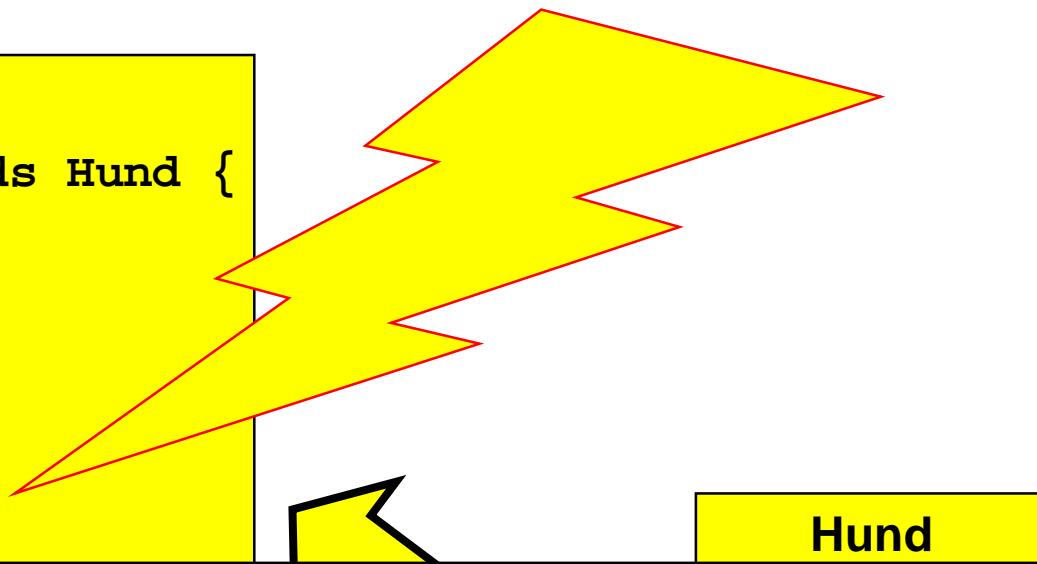
```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
}
```



Beispiel

Würden wir jedoch bei einer derart gestalteten Klasse die Methoden `bell()` oder `jaul()` aufrufen, so würde das Programm mit einer **Fehlermeldung** abbrechen.

```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
}
```



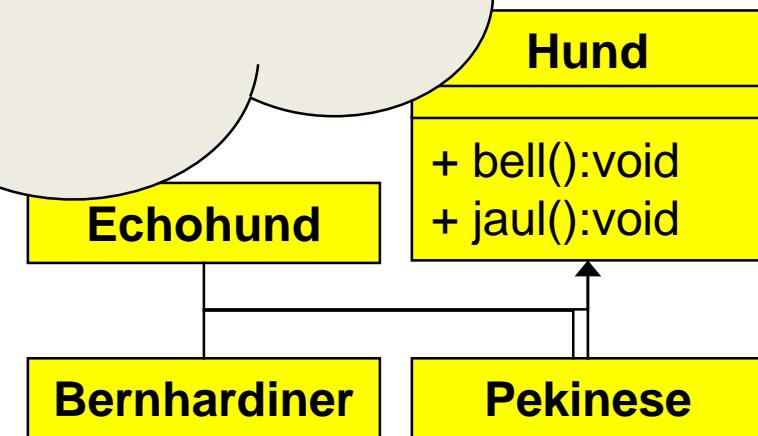
Der Grund hierfür liegt in einer Endlosschleife die wir produziert haben: Die neue Methode `bell()` ruft anstelle der originalen Methode (aus der Klasse `Hund`) erneut die in der Klasse `Echohund` überschriebene Methode `bell()` auf - also sich selbst! Das Programm bricht ab, sobald der für Methodenaufrufe reservierte **Speicher überläuft**.

Beispiel

Würden wir jedoch bei einer derart gestalteten Klasse die Methoden `bell()` oder `jaul()` aufrufen, so würde das Programm mit einer **Fehlermeldung** abbrechen.

```
public class Echohund {  
  
    public void bell()  
        bell(); // belle  
        bell(); // belle  
    }  
  
    public void jaul()  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    } }
```

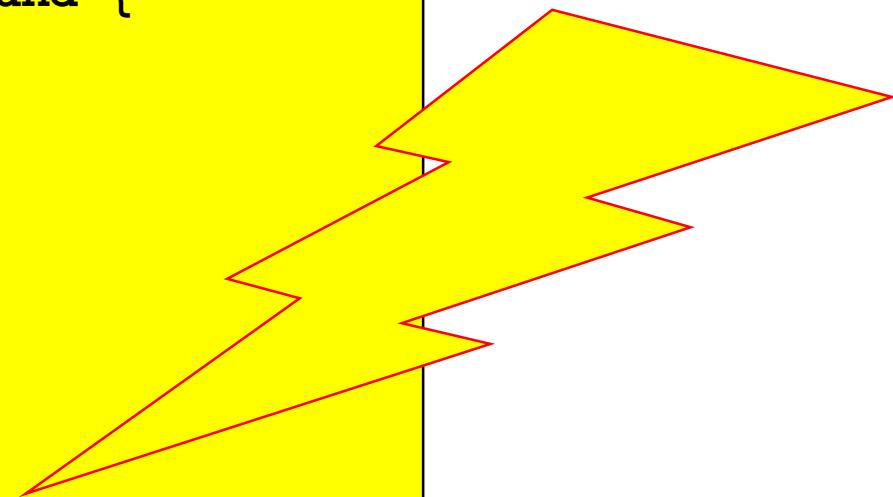
Wie lässt sich auf die Methode der Superklasse zugreifen?



Beispiel

Ein erster Ansatz wäre der Versuch, die Instanz einer Typumwandlung zu unterziehen. So wie sich etwa eine `double`-Zahl wie `3.14` durch eine Umwandlung der Form `(int) 3.14` in eine ganze Zahl umwandeln lässt, kann man ähnlich auch für Objekte verfahren.

```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaul() {  
        jaul(); // jaule einmal  
        jaul(); // jaule zweimal  
    }  
}
```



Beispiel

Eine entsprechende Umwandlung existiert also auch für Objekte in einer Hierarchie. Wir können eine Instanz **Fido** der Klasse **Echohund** durch einen Ausdruck der Form **(Hund) Fido** als eine Instanz der Klasse Hund betrachten (implizit haben wir das bereits getan).

```
public class Echohund extends Hund {  
  
    public void bell() {  
        bell(); // belle einmal  
        bell(); // belle zweimal  
    }  
  
    public void jaул() {  
        jaул(); // jaule einmal  
        jaул(); // jaule zweimal  
    }  
}
```



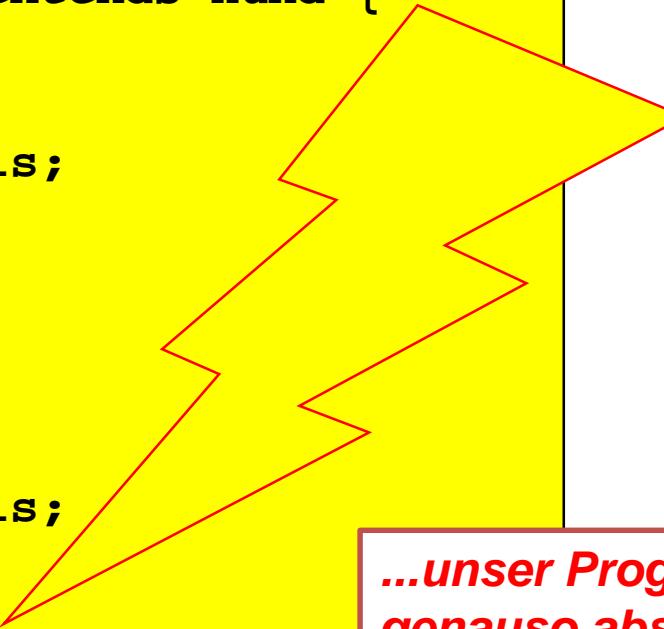
Können wir aber durch diese Umwandlung auf die alten Methoden zurückgreifen?

Beispiel

Wir wandeln die betroffene Instanz (**this**) in eine Instanz der Superklasse **Hund** um und rufen für diese dann die Methode **bell** bzw. **jaul** auf.

*Würden wir nach dieser Verbesserung für eine Instanz der Klasse die Methode **bell** aufrufen, so würde...*

```
public class Echohund extends Hund {  
  
    public void bell() {  
        Hund hund=(Hund)this;  
        hund.bell();  
        hund.bell();  
    }  
  
    public void jaul() {  
        Hund hund=(Hund)this;  
        hund.jaul();  
        hund.jaul();  
    }  
}
```



*...unser Programm dennoch
genauso abstürzen wie zuvor!*

Beispiel

Grund ist erneut das Prinzip des **Polymorphismus**. Eine Subklasse ist eine spezialisierte Fassung ihrer Superklasse. Wenn wir eine Methode überschreiben, so wird das aus der Subklasse instantiierte Objekt immer diese neue Methode statt der alten aufrufen - **egal**, ob sie als Instanz der Superklasse aufgefasst wird oder nicht!

```
public class Echohund extends Hund {  
  
    public void bell() {  
        Hund hund=(Hund)this;  
        hund.bell();  
        hund.bell();  
    }  
  
    public void jaul() {  
        Hund hund=(Hund)this;  
        hund.jaul();  
        hund.jaul();  
    }  
}
```

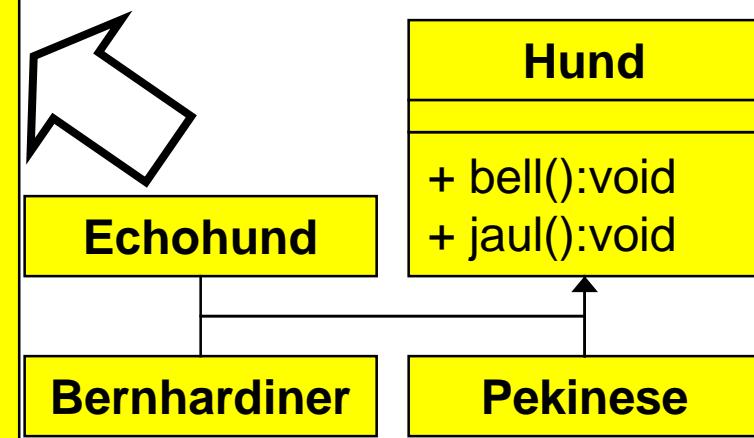


Wie lässt sich also auf die Methoden der Superklasse zugreifen?

Beispiel

Die Antwort steckt in dem Schlüsselwort **super**. So wie wir mit **this** auf alle Bestandteile einer Instanz zugreifen können, haben wir mit **super** Zugriff auf alle Bestandteile seiner Superklasse.

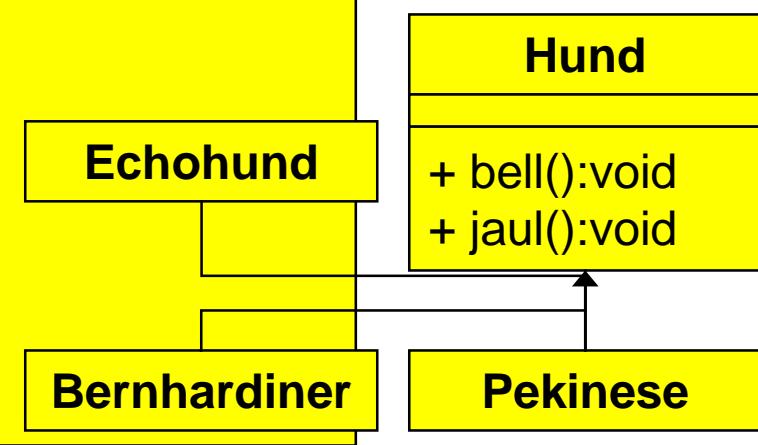
```
public class Echohund extends Hund {  
  
    public void bell() {  
        super.bell();  
        super.bell();  
    }  
  
    public void jaul() {  
        super.jaul();  
        super.jaul();  
    }  
}
```



Beispiel

Werfen wir nun noch einen Blick auf unser Hauptprogramm:

```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute=new Hund[4];  
        meute[0]=new Hund();  
        meute[1]=new Hund();  
        meute[2]=new Hund();  
        meute[3]=new Hund();  
        for (int i=0;i<meute.length;i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```



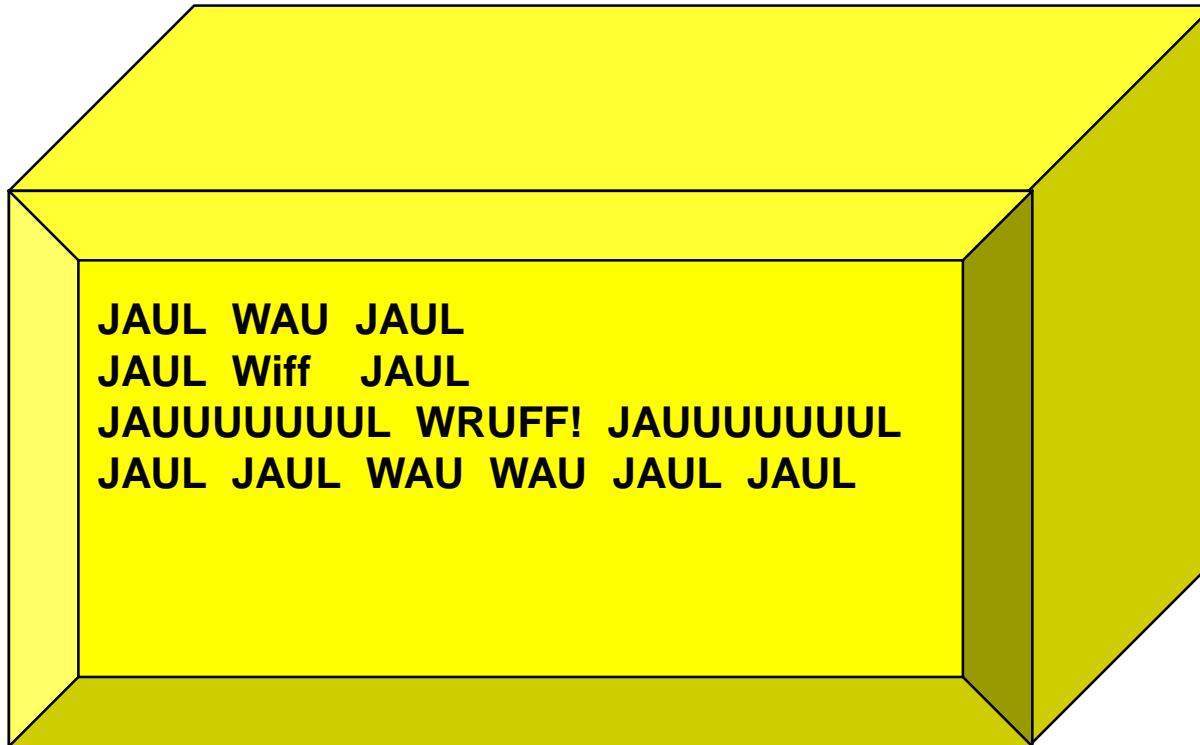
Beispiel

Werfen wir nun noch einen Blick auf unser Hauptprogramm:

```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute=new Hund[4];  
        meute[0]= new Hund(); // Promenadenmischung  
        meute[1]= new Pekinese(); // Hund Nr.2  
        meute[2]= new Bernhardiner(); // Hund Nr.3  
        meute[3]= new Echohund(); // Hund Nr.4  
        for (int i=0;i<meute.length;i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```

Beispiel

Das Prinzips des Polymorphismus erlaubt durch das Überschreiben von Methoden eine individuelle Ausprägung **verschiedener Objektklassen** unter der einheitlichen Schnittstelle **einer gemeinsamen Superklasse**.



- Es wird erst zur Laufzeit entschieden welche Version der Methode aufgerufen wird.
- Dies war abhängig davon, ob das jeweilige Objekt eine Instanz der Ober- oder einer Unterklasse ist.
- Bei diesem Vorgang des Überschreibens spricht man auch von **dynamischer Bindung** (**dynamic binding**).

Fazit

In einer Subklasse ist es möglich, Methoden neu zu verfassen, die mit derselben Schnittstelle bereits in der Superklasse definiert wurden. Man bezeichnet diesen Vorgang als **Überschreiben**.

In Verbindung mit dem Prinzip des **Polymorphismus** wird dieses Verfahren zu einem der wichtigsten Anwendungsgebiete der objektorientierten Programmierung:

Auf diese Weise können wir Algorithmen, die auf der allgemeineren Form einer Superklasse definiert sind, direkt auch auf alle ihre Subklassen anwenden, *obwohl sich diese in ihren Methoden vielleicht vollkommen unterschiedlich verhalten*.

- Kann man das Überschreiben von Methoden verhindern?
- Der Modifizierer **final** vor einer Methoden-Deklaration verhindert das Überschreiben dieser Methode in einer UnterkLASSE.
- Der compilierTE Programmcode wird **effizienter**, weil kein dynamisches Binden mehr durchgeföhrt werden muss.
- So wird auch für einige Methoden aus der nachfolgend beschriebenen Klasse **Object** durch den Modifizierer **final** sichergestellt, dass diese Methoden für alle Java-Objekte **das gleiche Verhalten** aufweisen.

Typumwandlung von Objekten

Konzert im Hinterhof, Teil 2

Die letzte „Mondscheinsonate“ unseres Hundequartetts blieb nicht ohne Folgen. Ein Nachbar beschwerte sich wegen des Lärms, so dass die Anwohnerschaft gezwungen war, drastische Schritte einzuleiten. Man tat sich zusammen und versah das Gelände mit einer engmaschigen Umzäunung, so dass kein Hund mehr in der Lage sein würde, den Hinterhof zu betreten. So glaubte man zumindest...

Als die vier Hunde in der nächsten Vollmondnacht zusammentrafen, fanden sie den Hof umzäunt und das Tor verschlossen. Der Zaun war so hoch, dass lediglich der Bernhardiner mit einem gewaltigen Satz hinüberhechten konnte. Der Pekinese, die Promenadenmischung und der Echohund zogen enttäuscht von dannen...

Beispiel

```
public class VollmondNacht {  
    public static void klaeffDichAus(Hund h) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
    public static void main(String[] args) {  
        Hund[] meute=new Hund[4];  
        meute[0]= new Hund(); // Promenadenmischung  
        meute[1]= new Pekinese(); // Hund Nr.2  
        meute[2]= new Bernhardiner(); // Hund Nr.3  
        meute[3]= new Echohund(); // Hund Nr.4  
        for (int i=0;i<meute.length;i++)  
            klaeffDichAus(meute[i]);  
    }  
}
```

Wir wollen unser Programm **so wenig wie möglich verändern**, um es weiterhin so allgemein wie möglich zu halten.

Vielleicht findet der Pekinese ja in einem Monat ein engmaschiges Loch, so dass auch er wieder am Konzert teilnehmen kann.

Die Belegung des Feldes und damit die Methode **main** soll also unangetastet bleiben.



Beispiel

Da wir in der `main`-Methode keine Veränderungen vornehmen wollen, müssen wir eine geeignete Erweiterung in der Methode `klaeffDichAus` vornehmen.

```
public static void klaeffDichAus(Hund h) {  
  
    if (h ist ein Bernhardiner) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
  
    }  
}
```

Wie erhalten wir Informationen darüber, ob es sich bei dem Objekt um die Instanz einer „verwandten“ Klasse handelt?

In komplexeren Programmen wird es oftmals notwendig sein, ein beliebiges Objekt auf seine Klassenzugehörigkeit zu prüfen. Aus diesem Grund haben die Entwickler der Programmiersprache Java den **instanceof**-Operator geschaffen.

Dieser Operator besagt, ob ein bestimmtes Objekt eine bestimmte Klasse instantiiert.

Der Operator wird in der Form

<Objektname> instanceof <Klassenname>

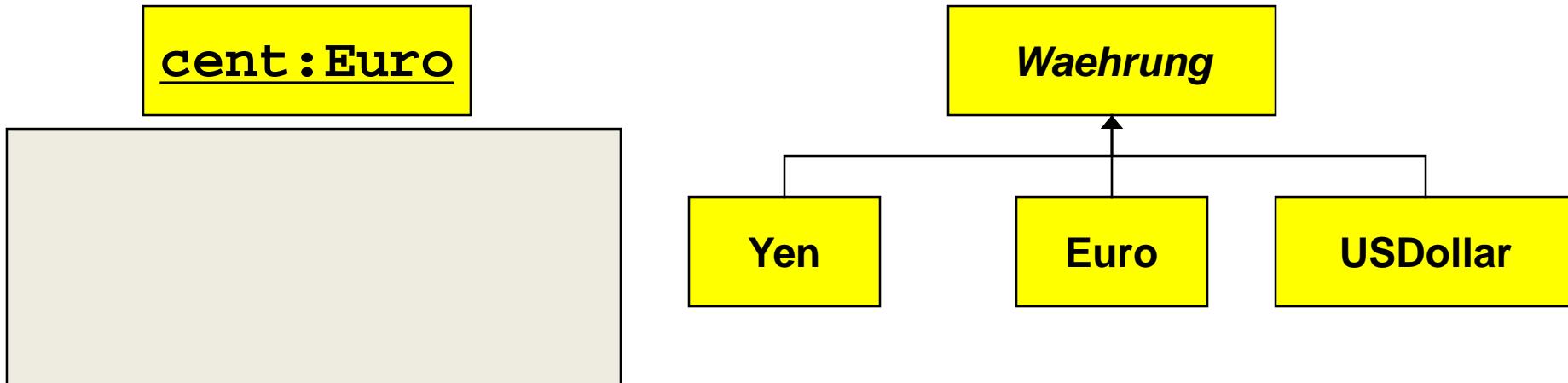
verwendet und liefert einen boolschen Wert als Ergebnis.

Beispiel

Wir instantiiieren ein **Euro**-Objekt durch die Zuweisung

```
Waehrung cent = new Euro(0.01);
```

und vergleichen dieses mit verschiedenen anderen Klassen aus unserer Klassenhierarchie.



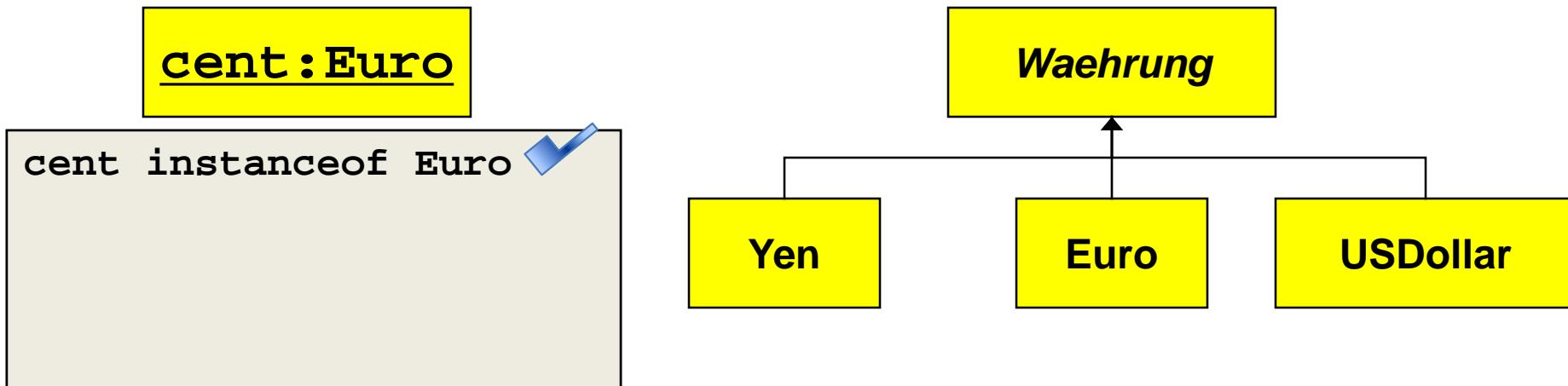
Beispiel

Die Operation

`cent instanceof Euro`

überprüft, ob das Objekt Instanz der Klasse **Euro** ist.

Diese Frage ist zu bejahen, denn obwohl wir das Objekt nur durch eine Variable vom Typ **Waehrung** referenzieren, handelt es sich dennoch um ein **Euro**-Objekt. Der Rückgabewert der Operation ist also **true**.



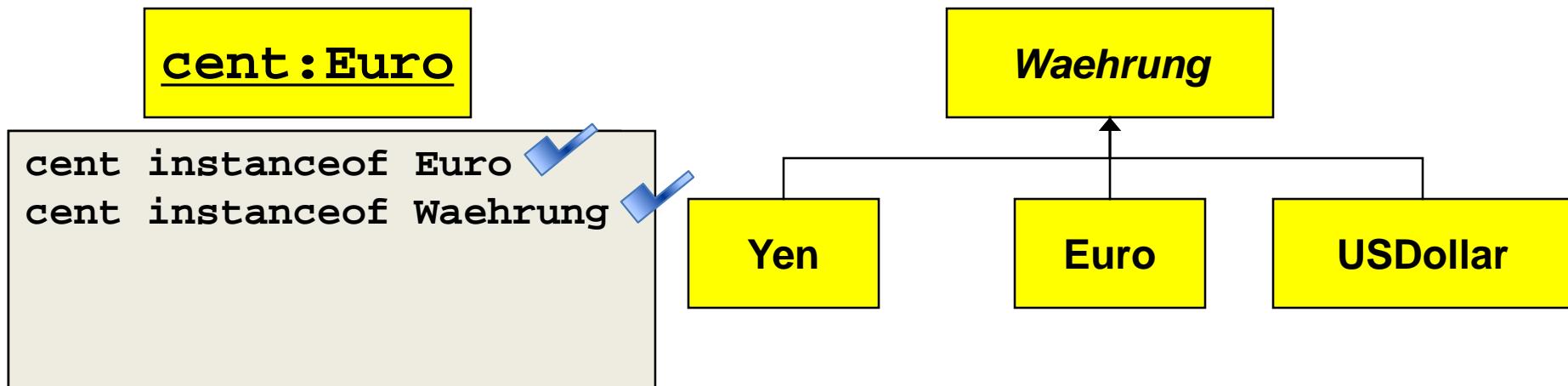
Beispiel

Die Operation

`cent instanceof Waehrung`

überprüft, ob das Objekt Instanz der Klasse **Waehrung** ist.

Auch diese Aussage trifft zu, denn **Waehrung** ist Superklasse von **Euro** - das Ergebnis ist also wiederum **true**.

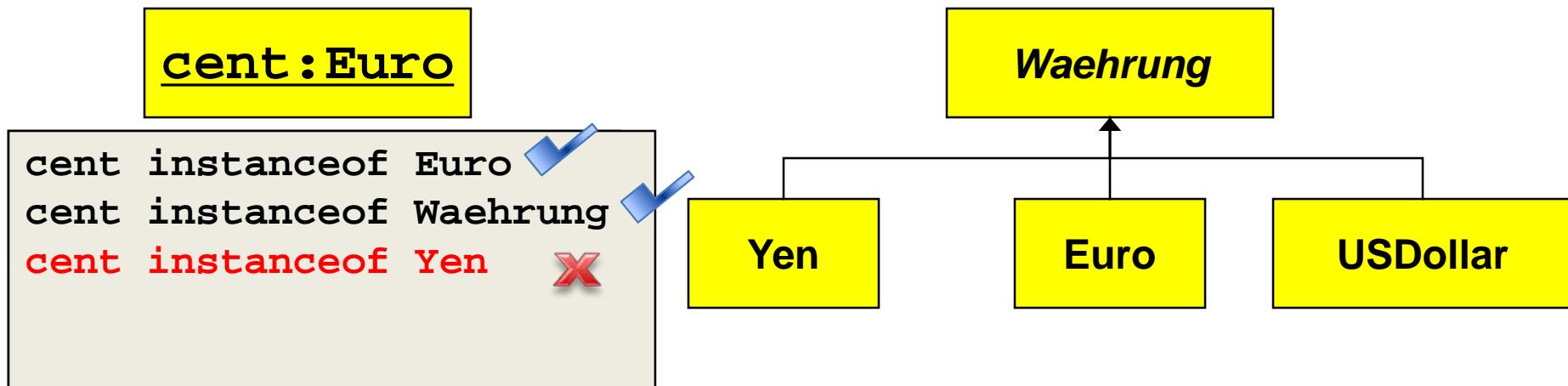


Beispiel

Die Operation

cent instanceof Yen

liefert im Gegensatz zu den anderen Vergleichen ein negatives Ergebnis. Die Klassen stehen in keiner direkten Beziehung zueinander!



Beispiel

Zurück zu unserem ursprünglichen Problem:

```
public static void klaeffDichAus(Hund h) {  
    if (h instanceof Bernhardiner) {  
        h.jaul();  
        h.bell();  
        h.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Wir wollten das System überprüfen lassen, ob die Variable `h` eine Instanz der Klasse **Bernhardiner** referenziert. Mit Hilfe des `instanceof`-Operators ist dies nun kein Problem mehr.

Hinweis

In den bisherigen Beispielen war es noch nicht notwendig gewesen, explizite Typumwandlung an Objekten anzuwenden. Die Programmzeilen

```
Waehrung eineWaehrung = new Euro(13);  
Hund fifi = new Bernhardiner();
```

wurden fehlerfrei übersetzt, da das System von der Sub- zur Superklasse implizit (also automatisch) konvertieren kann.

Die umgekehrte Richtung (**von der Super- zur Subklasse**)
lässt sich **nicht** automatisch bewerkstelligen!

Beispiel

Um unsere Methode **klaeffDichAus** etwas übersichtlicher zu gestalten, wollen wir das übergebene Objekt in einer anderen Variablen speichern. Sobald sicher ist, dass es sich um ein **Bernhardiner**-Objekt handelt, wollen wir dieses durch eine Variable namens **berni** referenzieren.

```
public static void klaeffDichAus(Hund h) {  
    if (h instanceof Bernhardiner) {  
        Bernhardiner berni=h;  
        berni.jaul();  
        berni.bell();  
        berni.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Der Versuch, das so veränderte Programm zu kompilieren, schlägt jedoch fehl. Statt des gewünschten Resultats erhalten wir eine Fehlermeldung der Form **Explicit cast needed to convert Hund to Bernhardiner.**

```
public static void klaeffDichAus(Hund h) {  
    if (h instanceof Bernhardiner) {  
        Bernhardiner berni = (Bernhardiner) h;  
        berni.jaul();  
        berni.bell();  
        berni.jaul();  
        System.out.println(); // kurze Pause  
    }  
}
```

Beispiel

Obwohl wir dank der **if**-Abfrage garantieren können, dass es sich bei dem von **h** referenzierten Objekt um einen **Bernhardiner** handelt, kann das System dies nicht automatisch erkennen.

Wir müssen eine explizite Typumwandlung vornehmen, wie wir sie bereits von den einfachen Datentypen her kennen.

Wir haben uns bereits sehr intensiv mit der Frage beschäftigt, wie Objekthierarchien, das Prinzip des Polymorphismus und das Überschreiben von Methoden miteinander in Zusammenhang stehen. Hierbei haben wir festgestellt, dass durch das Überschreiben von Methoden eben jene „Spezialisierung“ von Subklassen möglich war, die eine große Stärke der objektorientierten Modellierung darstellt.

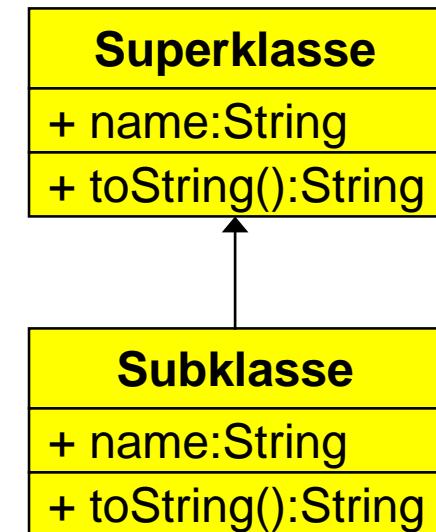


- Tatsächlich ist es in Java auch möglich, Variablen zu überschreiben.
- Jedoch gilt das Prinzip des **Polymorphismus** in Java **nicht für Variablen!**
- Auch wenn das Original durch eine Variable gleichen Namens verdeckt wird, unterscheidet das System klar zwischen der Variablen der Sub- und der Superklasse.

Beispiel

Wir entwerfen zwei einfache Klassen, Sub- und Superklasse, die wir zweckmäßigerweise auch so nennen.

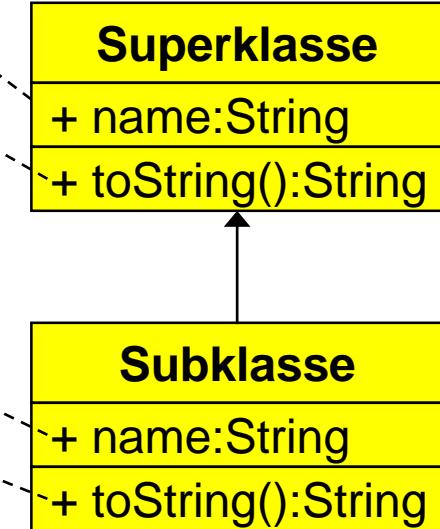
Beide verfügen über eine Instanzvariable „**name**“ und eine Methode **toString**. Diese Methode wird in der **Subklasse** neu definiert, das heißt sie überschreibt die entsprechende Methode ihrer **Superklasse**.



Beispiel

Werfen wir zuerst einen Blick auf den erzeugenden Quelltext:

```
public class Superklasse {  
    public String name="Super";  
  
    public String toString(){  
        return this.name;  
    }  
}  
  
public class Subklasse extends Superklasse {  
    public String name="Sub";  
  
    public String toString() {  
        return this.name;  
    }  
}
```



Beispiel

Wir vermerken diese Anfangsbelegung in Form von Kommentaren und werden nun mit unserem eigentlichen Testprogramm beginnen.

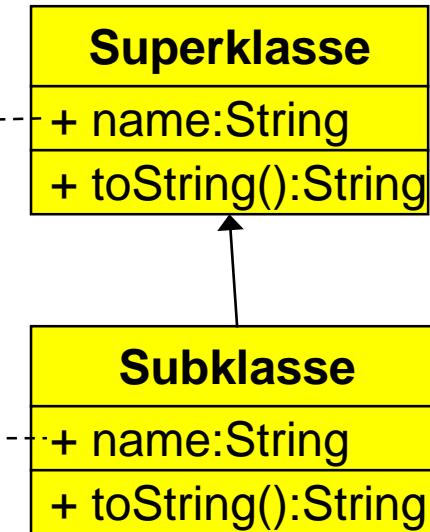
```
Superklasse vater      =new Superklasse();
Subklasse    sohn      =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```

name ist auf
„Super“ gesetzt.

name ist auf
„Sub“ gesetzt.



Wir erzeugen zwei neue Objekte **vater** und **sohn**, welche jeweils Super- und Subklasse instantiiieren.

Ferner erzeugen wir eine Referenz **gewandelt**, in der wir das **sohn**-Objekt als Instanz der Superklasse referenzieren.

Wir werden nun den Inhalt der Instanzvariablen auf verschiedenem Wege auf dem Bildschirm auszugeben versuchen.

```
Superklasse vater      =new Superklasse();
Subklasse   sohn       =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```

vater:Superklasse
name=„Super“

sohn:Subklasse
name=„Sub“

Superklasse
+ name:String
+ toString():String

Subklasse
+ name:String
+ toString():String

Wir versuchen zuerst, die durch **vater**, **sohn** und **gewandelt** referenzierten Objekte durch die Methode **toString** auf den Bildschirm zu bringen.

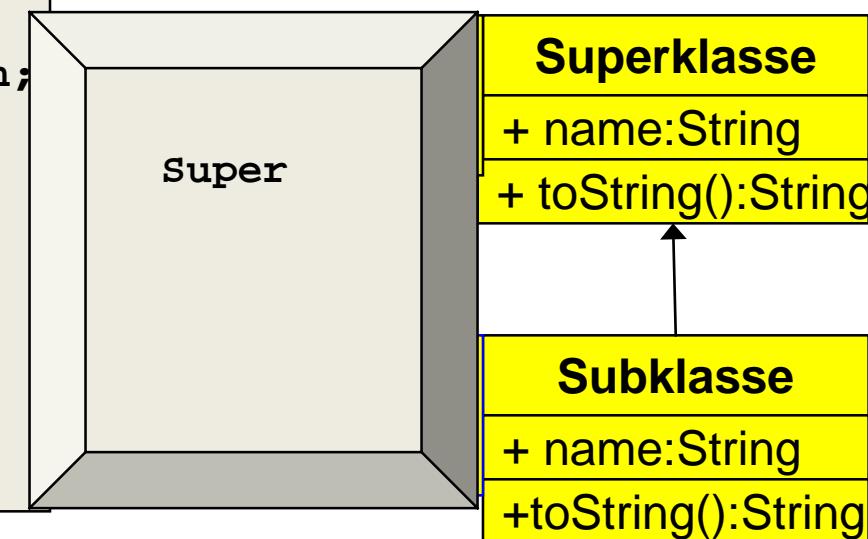
Im Falle des als **vater** bezeichneten Objektes ist das Ergebnis relativ einfach vorherzusehen.

Die Instanzmethode **toString** liefert den in der Variablen **name** gespeicherten Wert (also „**Super**“) zurück.

```
Superklasse vater      =new Superklasse();
Subklasse   sohn       =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```



Auch die Ausgabe für **sohn** ist einfach nachzuvollziehen. Die Methode **toString** liefert den Inhalt von **this.name** zurück.

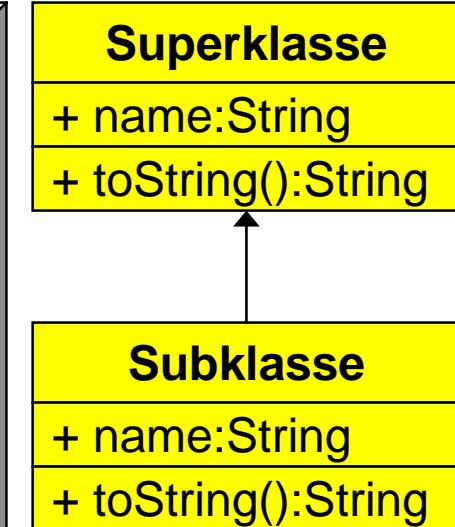
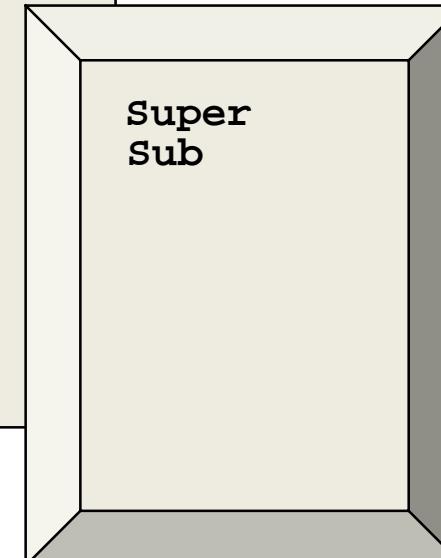
Hiermit ist natürlich eben jene Variable gemeint, die in derselben Klasse wie die Methode definiert wurde - also die der Subklasse.

Auf dem Bildschirm wird „Sub“ statt „Super“ ausgegeben.

```
Superklasse vater      =new Superklasse();
Subklasse    sohn      =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```



Das Ergebnis des Methodenaufrufes von `gewandelt.toString()` ist wieder auf das Prinzip des Polymorphismus zurückzuführen.

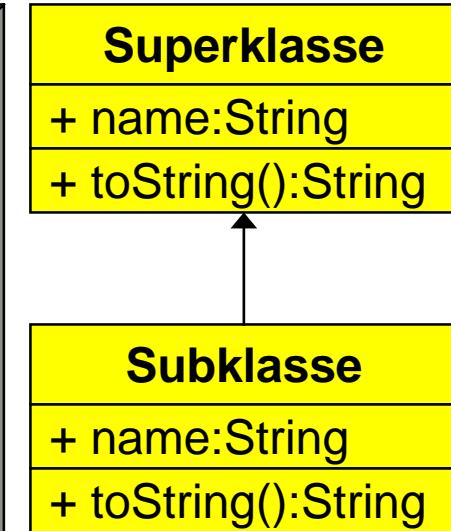
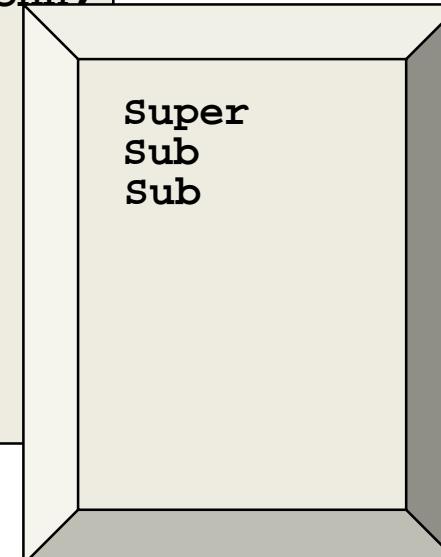
Obwohl das Objekt als Instanz der Superklasse angesehen wird, wird natürlich auch hier die **Methode der Subklasse aufgerufen**, welche die **originale `toString`-Methode überschreibt**.

Diese liefert dann erneut „Sub“ als Resultat.

```
Superklasse vater      =new Superklasse();
Subklasse   sohn       =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```



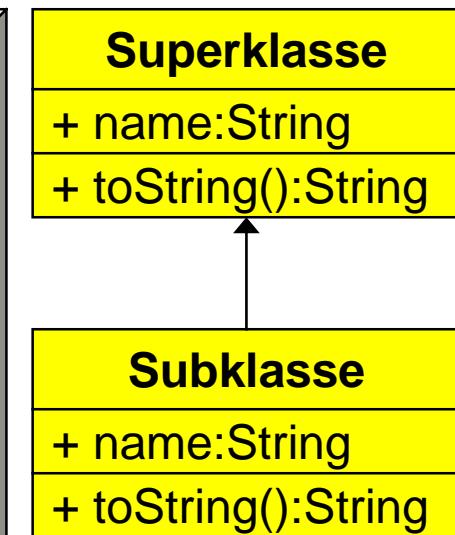
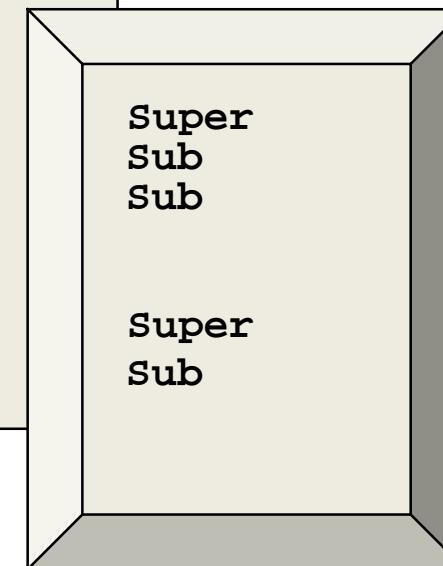
Wir werden nun den Inhalt der Variablen `name` direkt auf dem Bildschirm ausgeben - also ohne den Umweg über die `toString`-Methode.

Weder bei der Ausgabe von `vater.name` noch bei `sohn.name` erleben wir eine Überraschung. Der Mechanismus des Überschreibens scheint wie bei Methoden zu funktionieren.

```
Superklasse vater      =new Superklasse();
Subklasse   sohn      =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```



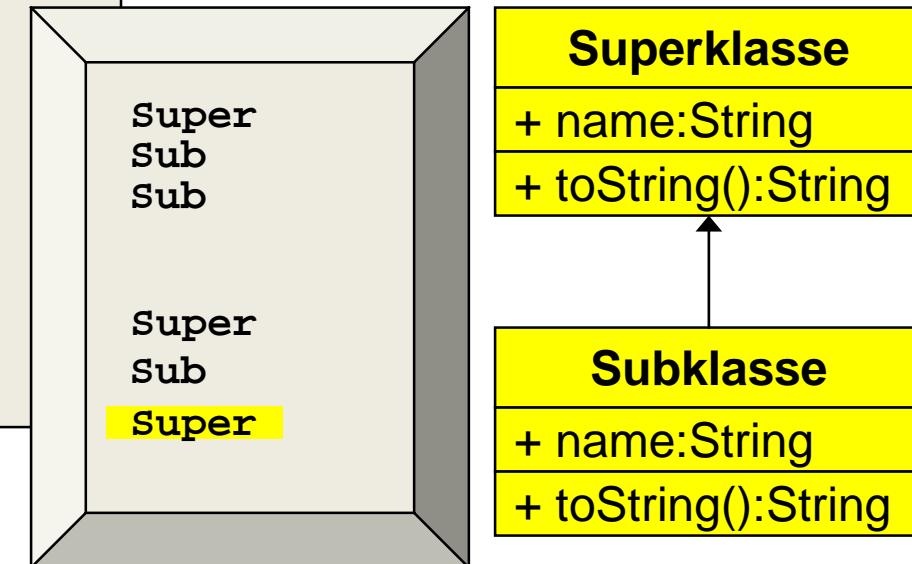
In der letzten Zeile jedoch erleben wir eine Überraschung: anstelle von „Sub“ wird „Super“ auf dem Bildschirm ausgegeben.

Der Grund hierfür liegt darin, dass Polymorphismus bei Variablen nicht anwendbar ist. Das **Objekt wird als Instanz der Superklasse betrachtet** - entsprechend erfolgt auch der Zugriff auf die Instanzvariable der Superklasse. Und in dieser ist nun einmal „Super“ abgespeichert...

```
Superklasse vater      =new Superklasse();
Subklasse   sohn      =new Subklasse();
Superklasse gewandelt=(Superklasse)sohn;

System.out.println(vater);
System.out.println(sohn);
System.out.println(gewandelt);

System.out.println(vater.name);
System.out.println(sohn.name);
System.out.println(gewandelt.name);
```



Fazit

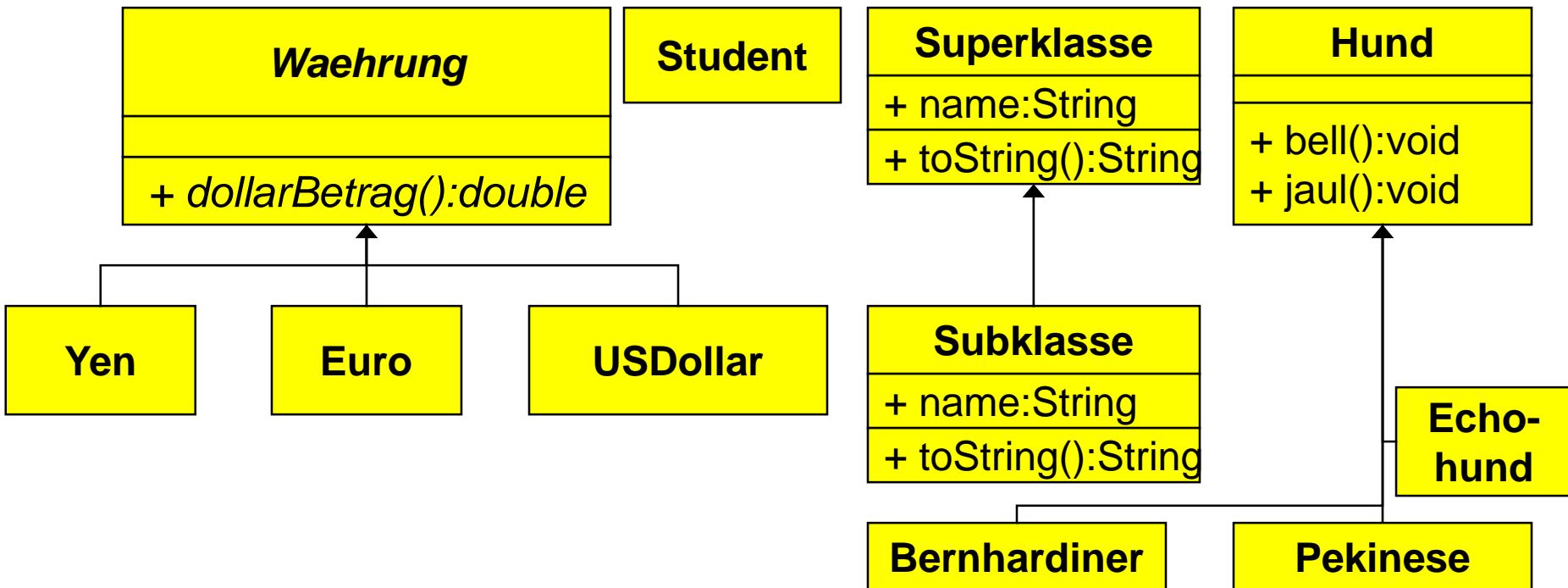
Auch Variablen können in Java überschrieben werden.

Das Prinzip des Polymorphismus ist auf diesen Fall allerdings nicht anwendbar!

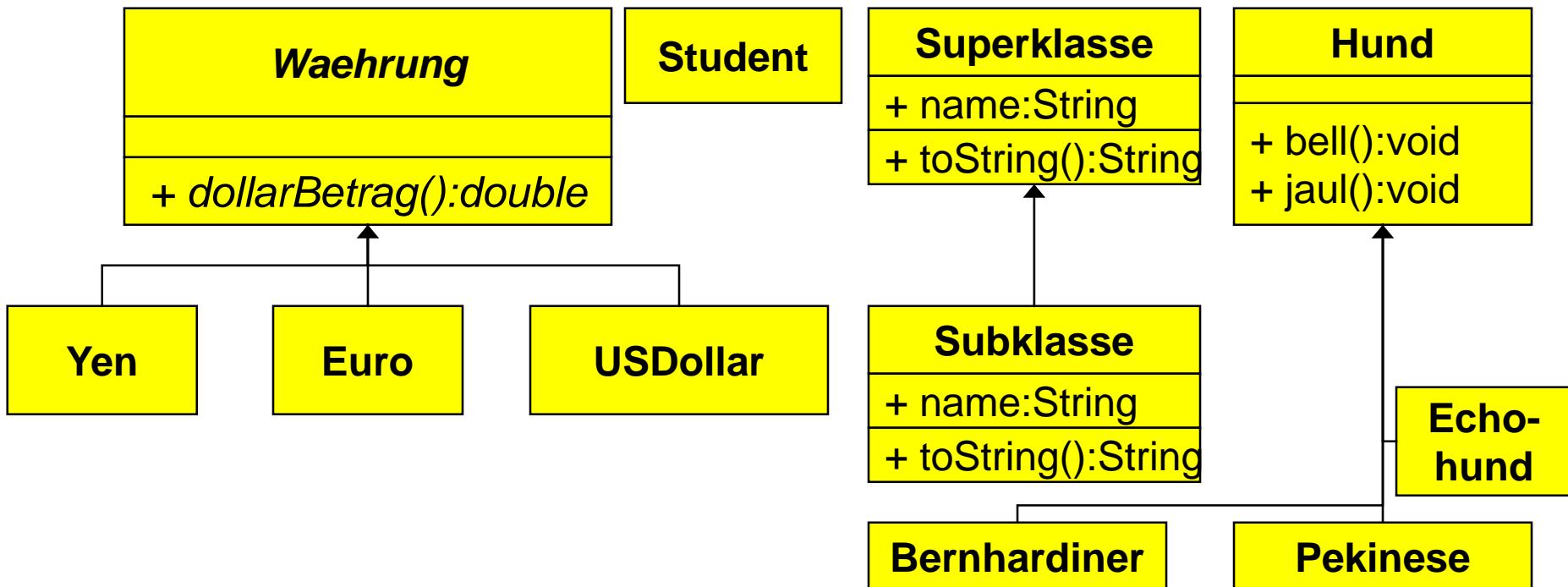
Ein Programmierer, der von dieser Möglichkeit also Gebrauch machen möchte, sollte sich immer des erhöhten Risikos bewusst sein, unbeabsichtigt Fehler in seinem Code zu produzieren.

Die Verwendung von Methoden zum Zugriff ([Datenkapselung](#)) ist immer noch die sicherste Methode!

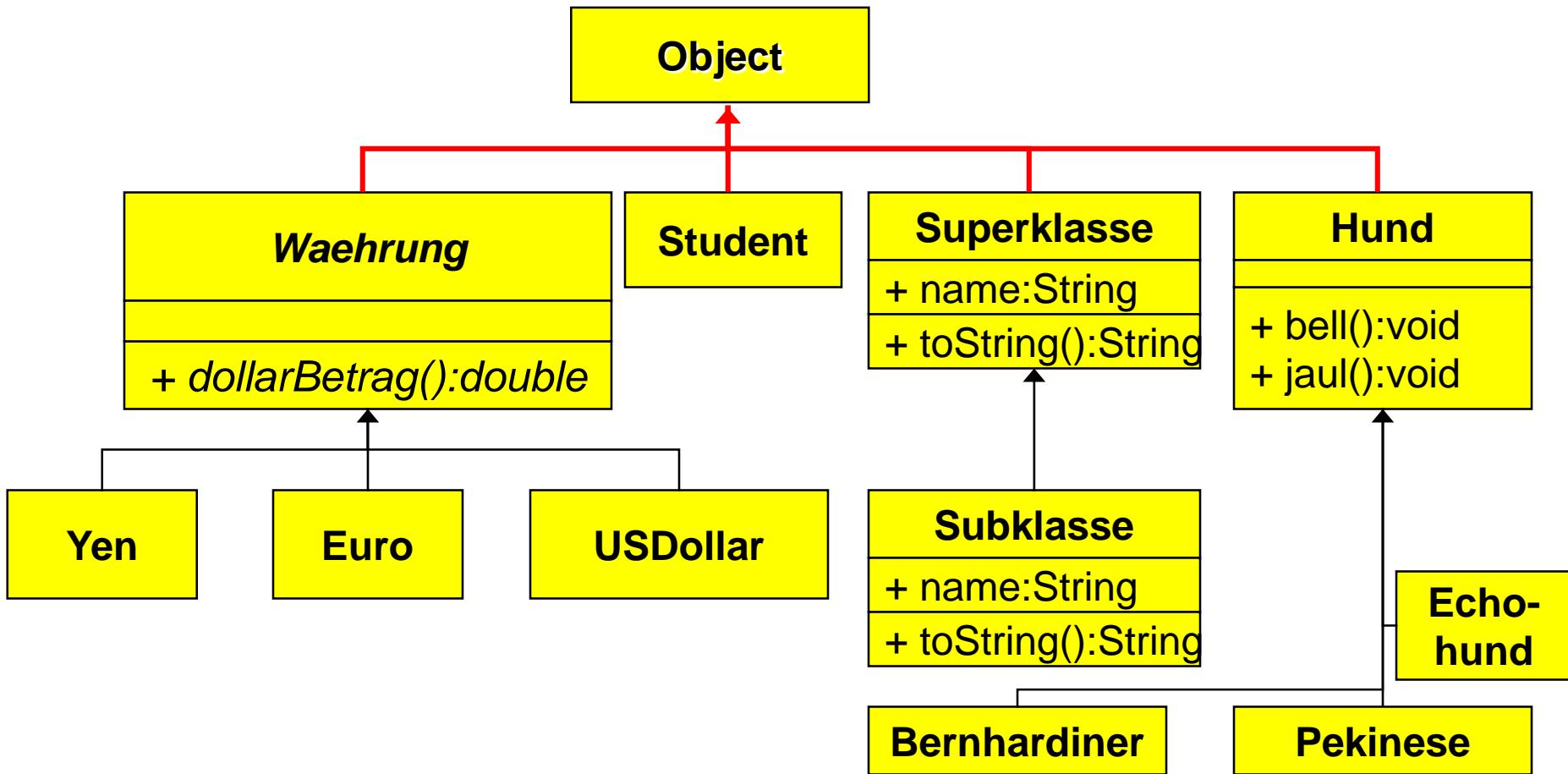
- Wir haben im Laufe der letzten Lektionen eine Vielzahl von verschiedenen Klassen geschaffen.
- Es stellt sich die Frage, ob diese verschiedenen Klassenhierarchien miteinander in Verbindung stehen.



- Es fällt an dieser Stelle natürlich schwer, eine Möglichkeit der Abstraktion zu finden. Welche Gemeinsamkeiten sollen etwa Studenten, Hunde und Zahlungsmittel aller Art besitzen?
- Tatsächlich gibt es jedoch eine Eigenschaft, die alle hier aufgeführten Klassen miteinander teilen: es lassen sich **Objekte** aus ihnen bilden.



- Diesem Umstand haben die Entwickler von Java Rechnung getragen und eine Klasse namens `Object` entwickelt. Jede wie auch immer gestaltete Klasse leitet sich von dieser ab.



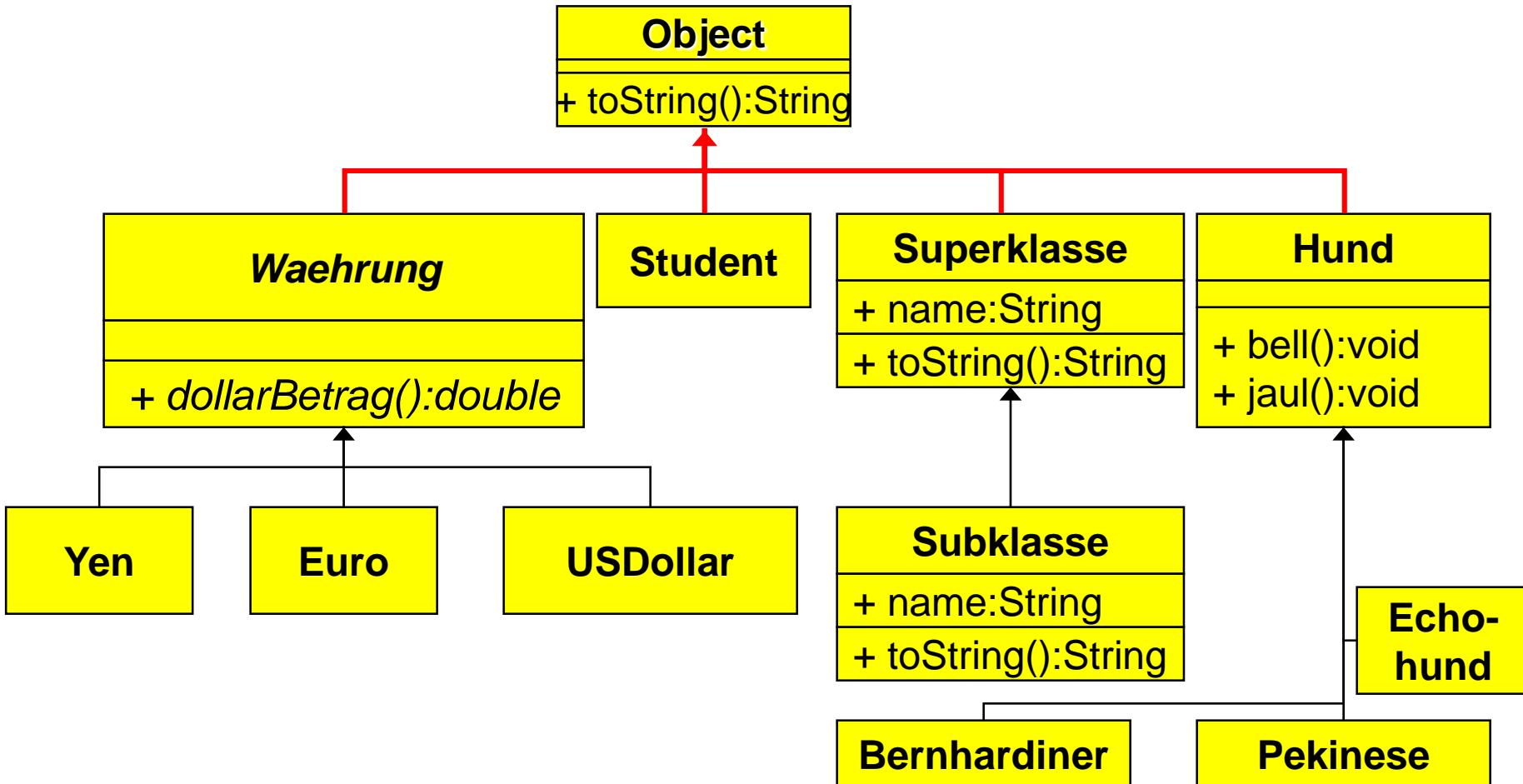
Welchen praktischen Sinn hat sie jedoch, die „**Mutter aller Klassen**“?

Um diesen Punkt zu verstehen, müssen wir begreifen, dass es gewisse Eigenschaften gibt, die alle Objekte miteinander teilen sollen. Hierbei handelt es sich hauptsächlich um die Existenz von gewissen Variablen und Methoden, die das System (aber auch der Benutzer) zum täglichen Umgang mit Objekten benötigt.

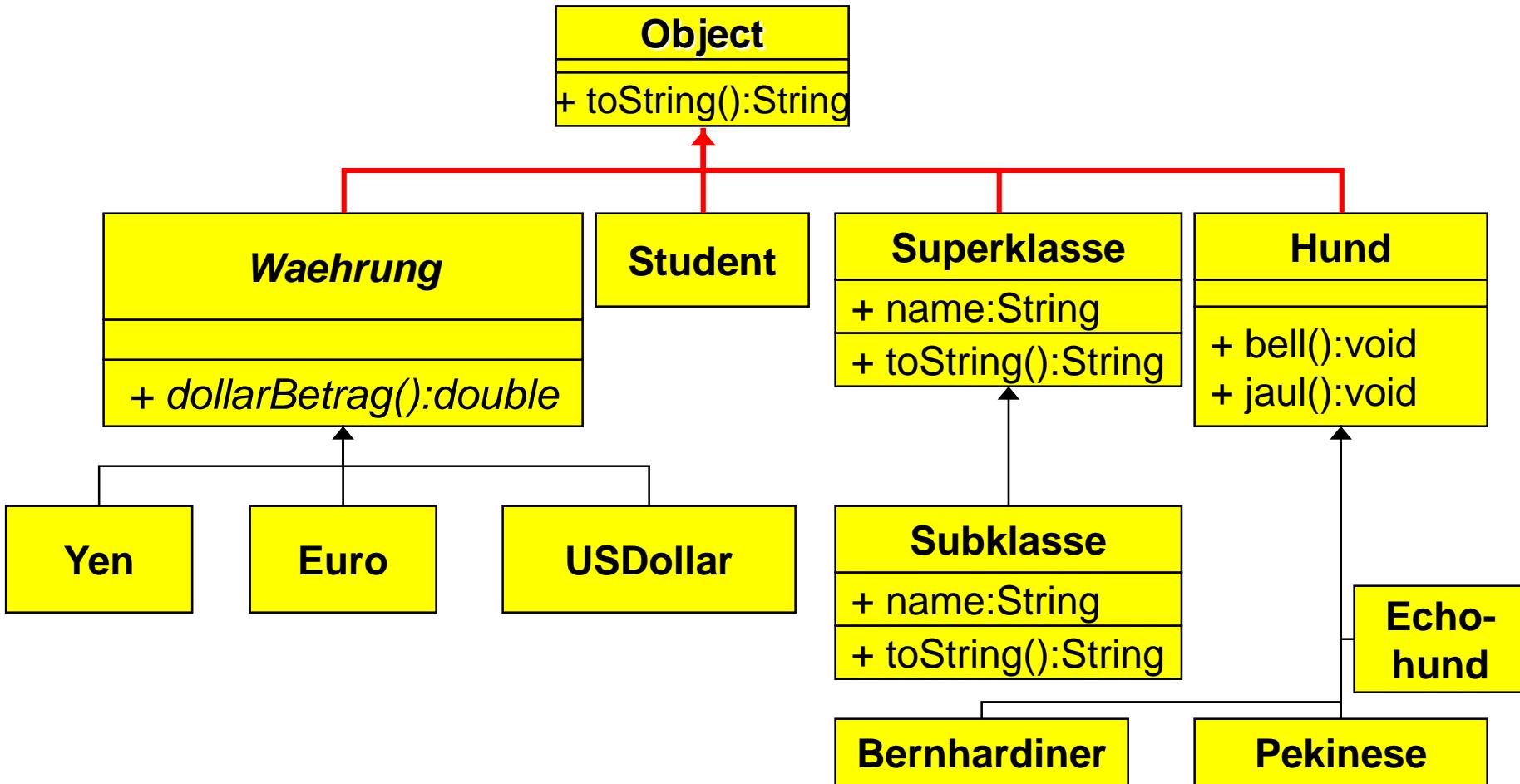
Ein gutes Beispiel hierfür ist die `toString`-Methode, mit der wir textuelle Darstellungen unserer Objekte erzeugen (etwa für unsere Bildschirmausgabe). Wenn wir Objekte mit dem Befehl `System.out.println` auf der Konsole ausgegeben haben, wurde von uns zu diesem Zweck immer eine `toString`-Methode geschaffen. Wir haben es als „Konvention“ betrachtet, die für die Sprache festgelegt wurde.

Der wahre Grund liegt jedoch in eben dieser Klasse **Object**.

Die Methode `toString` wurde bereits in der Klasse `Object` definiert. Aufgrund der Vererbung besitzt somit **jede** Klasse automatisch eine `toString`-Methode.



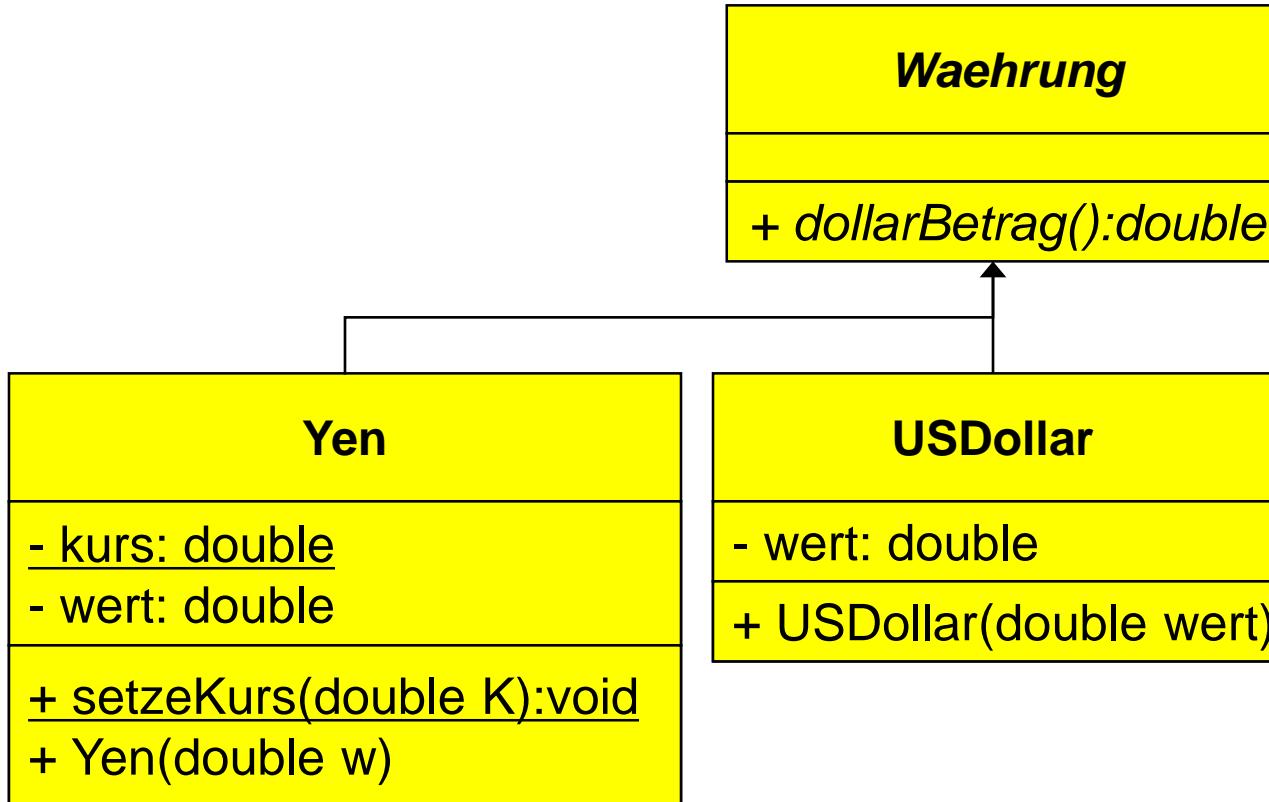
Wenn wir in einer Klasse also eine eigene Methode definieren, überschreiben wir hiermit lediglich die `toString`-Methode der Klasse `Object`.



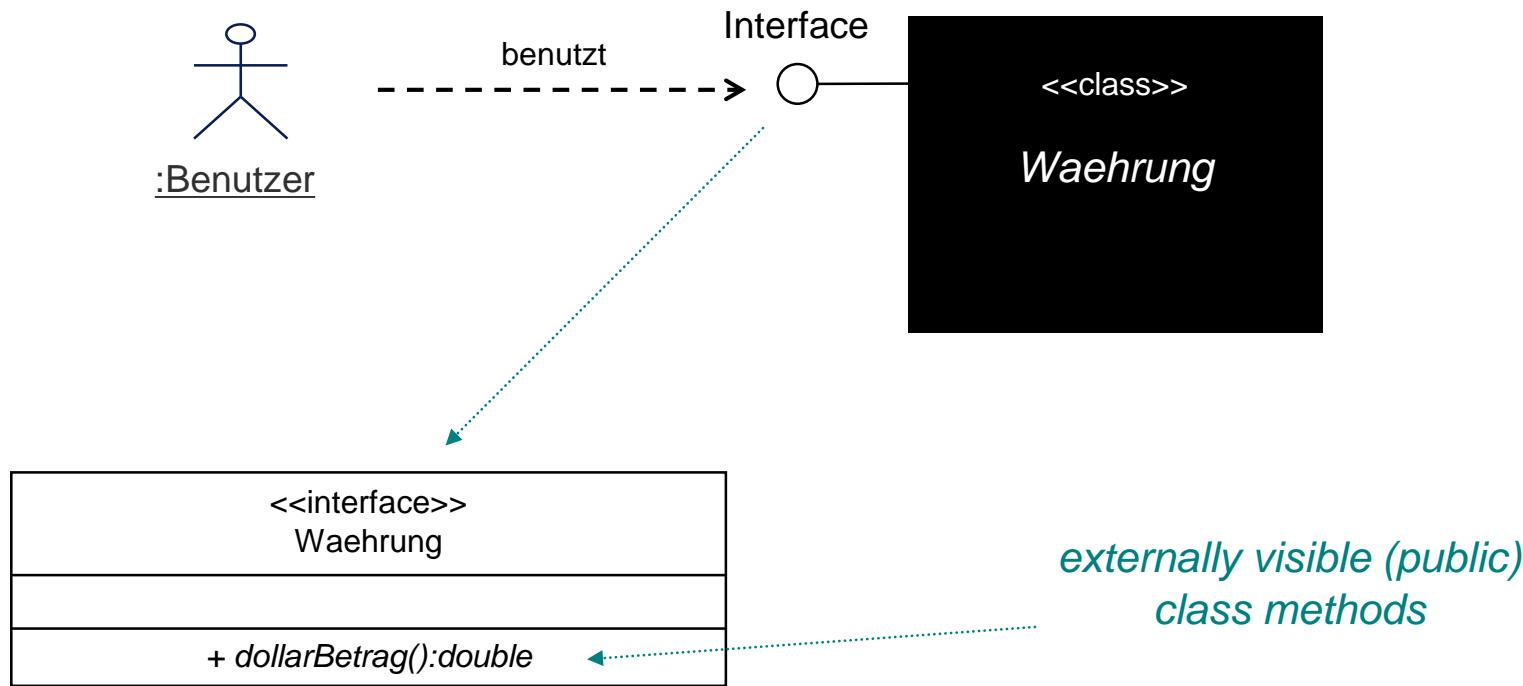
Inhalt

- Überschreiben von Methoden
- Typumwandlungen innerhalb einer Objekthierarchie
- Überschreiben von Variablen
- Die Klasse `java.lang.Object`
- **Interfaces**

Beispiel von letzter Vorlesung



Interface als allgemeines Konzept



- Alle konkreten Subklassen von Waehrung müssen eine Implementierung der Methode *dollarBetrag()* liefern
 - sie unterstützen (implementieren) damit die gleiche Schnittstelle
 - die entsprechende unterstützte Funktionalität kann einheitlich von außen aufgerufen werden ohne die internen Details der Klasse zu kennen

- Kann man in Java von mehreren Klassen erben?
- Mehrfachvererbung gibt es leider nicht in Java!
- Eine eingeschränkte Form existiert jedoch durch die Benutzung von Interfaces

Beispiel

Neben dem Barvermögen gehören unserer Hotelkette noch verschiedene andere Wertgegenstände, z.B. Grundstücke, Aktien, Firmenbeteiligungen etc.

Jeder dieser Wertgegenstände besitzt **völlig unterschiedliche Eigenschaften**.

Deshalb ist es **schwer für alle eine allgemeine Klasse zu definieren**.

Trotzdem soll der Gegenwert eines Wertgegenstands in Dollar (oder Yen, Euro,...) bestimmt werden können.

Dazu wollen wir unserer Klasse eine Schnittstelle vorgeben.

Alle unsere Klassen sollen eine Methode `wert()` besitzen, die den aktuellen Wert (z.B. unserer Immobilie) in einer beliebigen Währung zurückgibt.

Wir formulieren unsere Anforderung als ein sogenanntes **Interface**:

```
/** Ein beliebiger Wertgegenstand */
public interface Wertgegenstand {
    /** Gibt den Wert des Objekts als Waehrung zurueck */
    public Waehrung wert();
}
```

Alle unsere Klassen sollen eine Methode `wert()` besitzen, die den aktuellen Wert (z.B. unserer Immobilie) in einer beliebigen Währung zurückgibt.

Wir formulieren unsere Anforderung als ein sogenanntes **Interface**:

```
/** Ein beliebiger Wertgegenstand */
public interface Wertgegenstand {
    /** Gibt den Wert des Objekts als Waehrung zurueck */
    public Waehrung wert();
}
```

Interfaces sind im Gegensatz zu abstrakten Klassen keine Klassen im eigentlichen Sinne. Es existieren keine Kindklassen, die ein Interface mit Hilfe des Schlüsselwortes `extends` erben. Das entsprechende Wort für Interfaces heißt **implements**.

```
/** Ein Goldbarren (=Wertgegenstand) */
public class Goldbarren implements Wertgegenstand {
    /**Wie viel ist Gold heutzutage eigentlich wert? */
    public static double preisProGrammInDollar=60;

    /**Das Gewicht des Barrens */
    private double gewicht;

    /** Konstruktor - das Gewicht ist in Gramm anzugeben*/
    public Goldbarren(double gewichtInGramm){
        gewicht = gewichtInGramm;
    }

    /**Implementierung des Interfaces*/
    public Währung wert() {
        return new US Dollar(gewicht*preisProGrammInDollar);
    }
}
```

Beispiel

Beispiel

Die Klasse **Goldbarren** setzt das Interface **Wertgegenstand** in einer Klasse um und macht dies dem Compiler durch die Worte **implements Wertgegenstand** klar.

Um nun dabei eine gültige Klassendefinition zu erzeugen **müssen** wir eine entsprechende Methode **wert** definieren (sonst erhalten wir eine Fehlermeldung beim Übersetzen).

Hierbei errechnen wir den Wert unseres Barrens in Dollar aus dem Gewicht und geben diesen als **USDollar**-Objekt zurück.

Dieses Vorgehen ist erlaubt, da **USDollar** Subklasse von **Waehrung** ist.

Beispiel

```
/** Berechne den Gesamtwert einer Menge von Wertgegenstaenden */
public static Waehrung gesamtwert(Wertgegenstand[ ] objekte){
    double summe=0;
    for (Wertgegenstand w : objekte){
        summe += w.wert().dollarBetrag();
    }
    return new USDollar(summe);
}
```

In der Schleife werden die verschiedenen Geldbeträge über die Methode `wert()` ausgelesen.

Weil das Resultat dieser Methode jeweils ein Währungsobjekt ist, muss der Dollarbetrag noch über die Methode `dollarBetrag()` ausgelesen werden.

- Sofern die Objekte das Interface implementieren, können sie als Wertgegenstand aufgefasst werden.
- **Unterschied** zu abstrakten Klassen: **Die Methoden im Interface müssen alle abstrakt sein.**
- Damit bieten sich Interfaces als Schnittstellenvorgabe an.
- Im Gegensatz zu normalen Klassen **erlauben Interfaces** jedoch die Möglichkeit von **Mehrfachvererbung**.
- Eine Klasse darf nämlich zwar nur eine Superklasse besitzen; sie darf aber beliebig viele Interfaces implementieren!

- Die Klasse **String** dient zur Darstellung von Zeichenketten
- Alle Zeichenketten-Literale in Java Programmen (z.B. "abc") sind als Instanzen dieser Klasse realisiert
- Werte vom Typ String sind konstant, ihre Werte können nach ihrer Erzeugung nicht mehr verändert werden
- Es gibt verschiedene äquivalente Varianten zur Erzeugung von Strings:

```
String s1 = "abc";
```

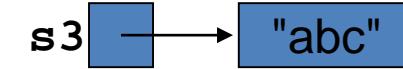
// Variante 1



```
String s2 = new String("abc"); // Variante 2
```



```
char[] data = {'a', 'b', 'c'}; // Variante 3
String s3 = new String(data);
```



```
byte[] b = {97, 98, 99}; // Variante 4
String s4 = new String(b);
```



```
String s5 = new String(s4); // Variante 5
```



- Die Klasse **String** beinhaltet Methoden zum
 - Zugriff auf einzelne Zeichen der Zeichenkette
 - Vergleich von Zeichenketten
 - Suchen von Teil-Zeichenketten
 - Herausgreifen von Teil-Zeichenketten
 - Wandeln von Groß- in Kleinbuchstaben und umgekehrt
- Strings kann man mit dem Operator + konatenieren (aneinanderhängen)
- Andere Objekte können in Strings umgewandelt werden
- Alle "Veränderungen" an einem String laufen so ab, dass jeweils ein neues String-Objekt geliefert wird!

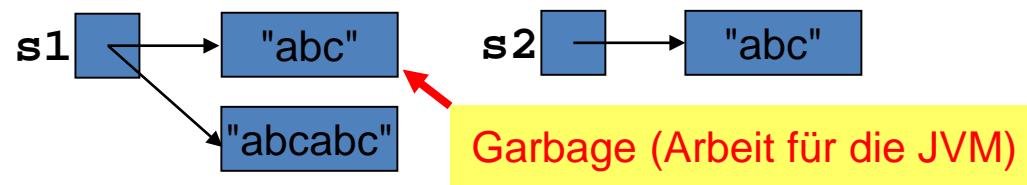
Beispiel

vorher:



Anweisung: **s1 = s1 + s2**

nachher:



Beispiele für String-Methoden

```
public class StringTest {  
    public static void main (String[] args) {  
        String s1 = "Weihnachten";  
        String s2 = "Veihnachten";  
        String s3 = "Xeihnachten";  
        String s4 = "WEIHNACHTEN";  
        System.out.println(s1);  
        System.out.println(s1.charAt(4));  
        System.out.println(s1.compareTo(s1));  
        System.out.println(s1.compareTo(s2));  
        System.out.println(s1.compareTo(s3));  
        System.out.println(s1.endsWith("ten"));  
        System.out.println(s1.equals(s2));  
        System.out.println(s1.equalsIgnoreCase(s4));  
        System.out.println(s1.indexOf("n"));  
        System.out.println(s1.indexOf("ach"));  
        System.out.println(s1.length());  
        System.out.println(s1.replace('e','E'));  
        System.out.println(s1.startsWith("Weih"));  
        System.out.println(s1.substring(3));  
        System.out.println(s1.substring(3,7));  
        System.out.println(s1.toLowerCase());  
        System.out.println(s1.toUpperCase());  
        System.out.println(String.valueOf(1.5e2));  
    }  
}
```

Ausgaben

Weihnachten
n
0
1
-1
true
false
true
4
5
11
WEihnachtEn
true
hnachten
hnac
weihnachten
WEIHNACHTEN
150.0

- ...stellt veränderbare Zeichenketten dar
- Beispiel

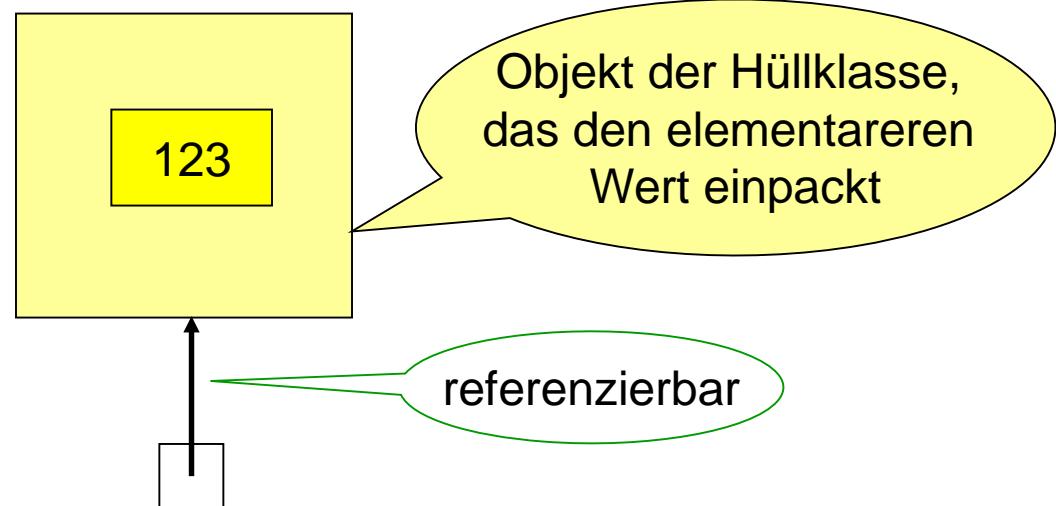
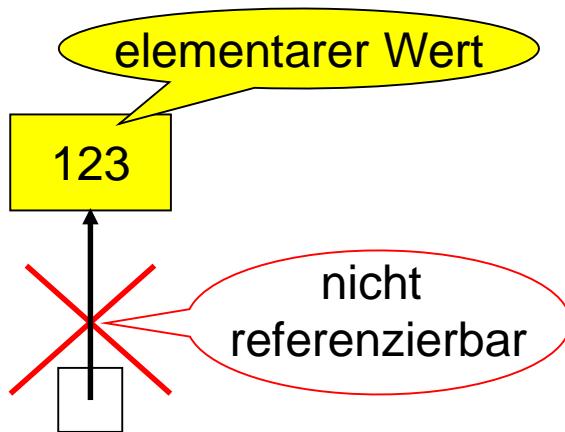
```
StringBuffer x = new StringBuffer();  
x.append("A").append(4).append("-Seite");
```

- Im wesentlichen stellt die Klasse **StringBuffer** die Methoden **append** und **insert** zur Verfügung, die auf dem **StringBuffer**-Objekt selbst arbeiten (es muss also kein neuer String erzeugt werden). Die Methoden sind für fast alle elementaren Datentypen überladen.
- Die Anweisung

```
x = new StringBuffer("start").append("s").insert(4,"le");
```

sorgt also dafür, dass **x** den Wert "**starlets**" enthält.

- **Frage:** Können wir ein Feld anlegen, das in seinen Komponenten Werte von unterschiedlichen elementaren Datentypen speichern kann? → **Nein!**
- Andererseits könnten in einem Feld vom Typ `Object[]` Referenzen auf Objekte beliebiger Klassen gespeichert werden.
- Um Werte der elementaren Datentypen genauso handhaben zu können, muss man diese in Objekte "einpacken".
- Dazu stellt Java so genannte **Hüllklassen (wrapper classes)** zur Verfügung.

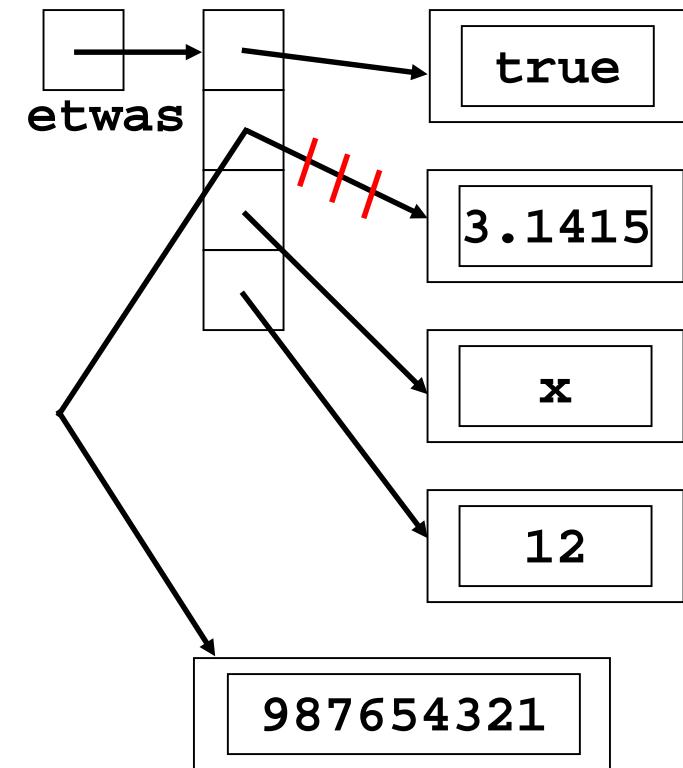


- Für elementare Datentypen existieren Hüllklassen mit ähnlichem Aufbau
 - **Byte, Short, Integer, Long, Float, Double**
 - **Boolean**
 - **Character**

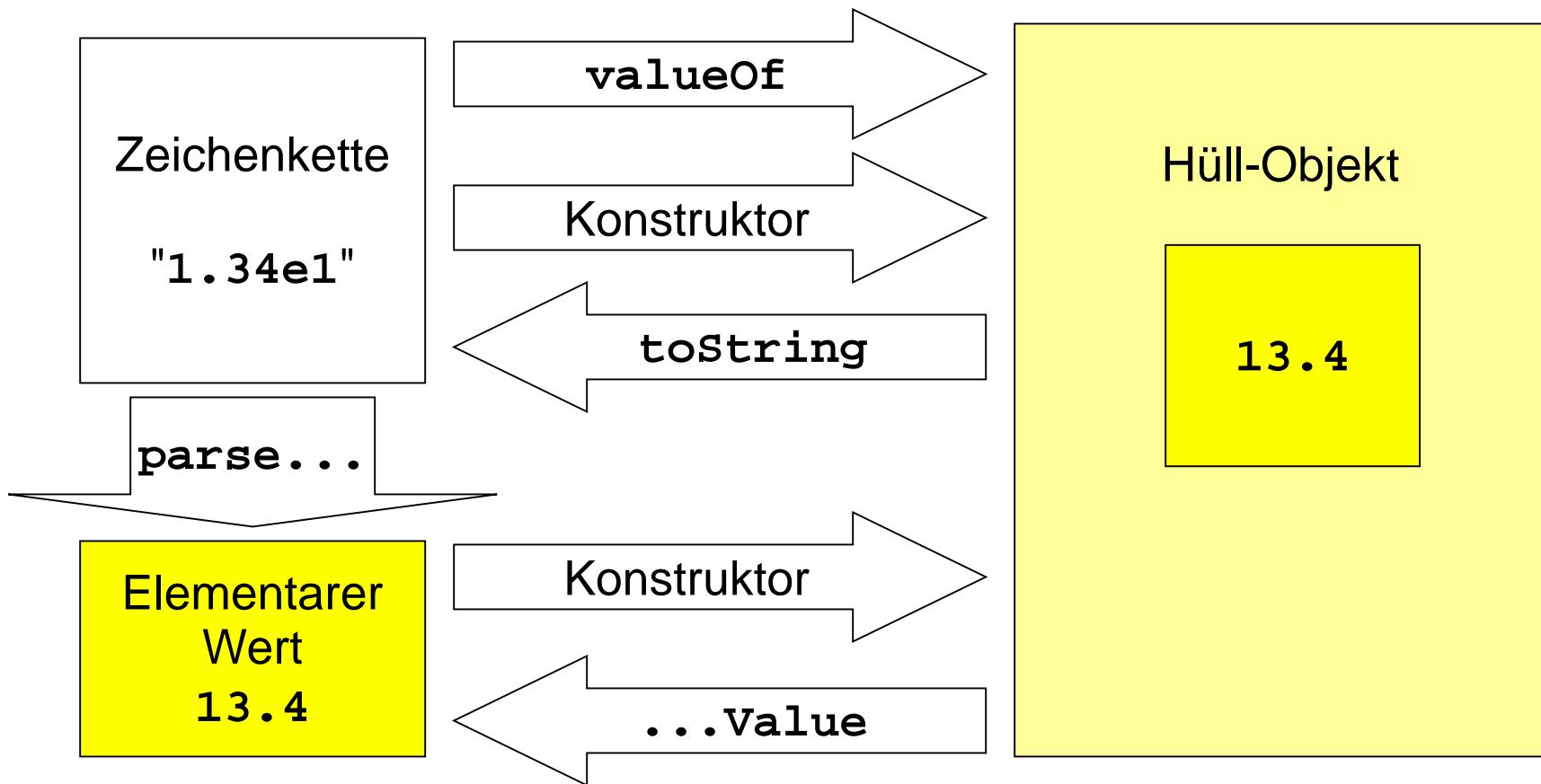


elementarer Datentyp	Wrapper-Klasse	Konstruktoren
byte	Byte	Byte(byte b) Byte(String s)
short	Short	Short(short s) Short(String s)
int	Integer	Integer(int i) Integer(String s)
long	Long	Long(long l) Long(String s)
float	Float	Float(float f) Float(String s)
double	Double	Double(double d) Double(String s)
boolean	Boolean	Boolean(boolean b) Boolean(String s)
char	Character	Character(char c)

```
public class WrapperBeispiel {  
    public static void main (String[] args) {  
        Object[] etwas = new Object[4];  
        etwas[0] = new Boolean(true);  
        etwas[1] = new Double(3.1415);  
        etwas[2] = new Character('x');  
        etwas[3] = new Integer(12);  
        for (int i=0; i<4; i++)  
            System.out.println(etwas[i]);  
        etwas[1] = new Long(987654321);  
        for (int i=0; i<4; i++)  
            System.out.println(etwas[i]);  
    }  
}
```



- Aufbau am Beispiel **Double** als Hüllklasse zu **double**
 - je zwei Konstruktoren
 - **Double (double d)**
 - **Double (String s)**
 - Klassenmethode **Double valueOf(String s) { ... }**
wandelt einen String in ein **Double**-Objekt
 - Instanzmethode **double doubleValue() { ... }**
wandelt ein **Double**-Objekt in einen **double**-Wert
 - Klassenmethode **double parseDouble(String s) { ... }**
wandelt einen String in einen **double**-Wert
 - **double x =**
Double.valueOf("12.345").doubleValue();
entspricht also
double x = Double.parseDouble("12.345");



- C++ bietet Klassen an: Effektiv **structs**, welche zusätzlich Methoden enthalten können
- Anders als in Java wird die Verwendung von Klassen nicht erzwungen
 - d.h. Methoden können auch außerhalb von Klassen auftreten
- Bei Klassen, Attributen und Methoden gilt auch das Prinzip der **Vorwärtsdeklaration!**
 - Problem: Zyklische Aufrufe sind unmöglich. Methodenordnung ist schwierig
 - Lösung: Prinzipielle separate Deklaration in Header-Dateien

- Prinzip der **Vorwärtsdeklaration**

- Problem: Zyklische Aufrufe sind unmöglich. Methodenordnung ist schwierig.
- Beispiel: Inkrementiert und verdoppelt Wert **n**-Mal.
Enthält zyklische Abhängigkeit.

```
int doubleValue(int a, int i) {  
    return increment(2*a, i);  
}  
int increment(int a, int i){  
    if (i == 0) { return a; }  
    return doubleValue(a+1, i-1);  
}  
int main() {  
    int i = 5;  
    return increment(1, i);  
}
```

- Compilerausgabe:

```
main.cpp: In function 'int doubleValue(int, int)':  
main.cpp:2:27: error: 'increment' was not declared in this scope  
        return increment(2*a,i);  
                           ^
```

- Lösung: Separate Vorwärtsdeklaration

Vorwärtsdeklaration

Deklariert increment(int,int)
vor der Implementierung.

Deklarationen werden
normalerweise in
separaten **header-**
Dateien (.h) abgelegt die
mit **#include**
eingebunden werden (z.B.
#include <string.h>
bindet die string.h
header-Datei aus der
C-Standard-Library ein)

```
int increment(int, int);  
  
int doubleValue(int a, int i) {  
    return increment(2*a, i);  
}  
  
int increment(int a, int i){  
    if (i == 0) { return a; }  
    return doubleValue(a+1, i-1);  
}  
  
int main() {  
    int n = 5;  
    return increment(1, i);  
}
```

**Referenzierung von
deklarerter Funktion**

doubleValue(int,int) kann
increment(int,int) referenzieren,
da es bereits deklariert worden
ist.

**Auflösung der separaten
Implementierung**

Als Teil des Kompilievorgangs
ordnet der Linker die
Implementierung von
increment(int,int) der Deklaration
zu.

ACHTUNG: Das geht auch Datei-
übergreifend!

- Enthalten Deklarationen von Funktionen, Datenstrukturen und Klassen
 - Bei Klassen: Attributen und Methoden
- **private / public** Modifizierer können für ganze Deklarations-Blöcke verwendet werden

File myfile.h

```
// class declaration
class MyClass {
    private:
        int x;
    public:
        int getX();
        void setX(int x);
};
```

- Enthalten die Implementierungen der Methoden
- Können auf die Variablen aus dem Header zugreifen
- :: Notation zeigt Zugehörigkeit zur Klasse an

```
File container.cpp
-----
#include "myclass.h"

int MyClass::getX() { <----->
    return x;
}

void MyClass::setX(int x) {
    (*this).x = x;
}
```

Die **getX** Methode von
der Klasse **MyClass**

Achtung:
Anders als in Java ist **this**
ein Pointer auf das Objekt,
nicht das Objekt selbst!

- Objekte können direkt deklariert und angesprochen werden
 - Anders als in Java, werden hier die entsprechenden Variablen ähnlich wie Variablen einfacher Datentypen behandelt
 - Die Variablen sind KEINE Referenzen oder Pointer; sie beziehen sich auf das Objekt selbst!
 - Members werden per Punktnotation angesprochen

```
#include <iostream>
#include "myclass.h"

int main() {
    MyClass myclass = MyClass();
    myclass.setX(5);
    return 0;
}
```

Standardkonstruktor wird für die Instanziierung verwendet

- Der **->** Operator

- Einfacher Zugriff auf Members von Objekt- oder struct-Pointern
- Für jedes Objekt/struct **x** mit Pointer **ptr = &x** gilt:

ptr->y; ist äquivalent zu **(*ptr).y;**

```
#include <iostream>
#include "myclass.h"

int main() {
    MyClass myclass = MyClass();
    myclass.setX(5);
    MyClass *ptr = &myclass;
    cout << ptr->getX();
    // equivalent to: cout << (*ptr).getX();
    return 0;
}
```

- Jedes Programm wird in drei Speicherbereiche (Segmente) aufgeteilt:
 - *text segment* (oder *code segment*): Programminstruktionen und Konstanten
 - *stack segment*: Lokale Variablen und Parameter bei Funktionen
 - *heap segment*: Globale Variablen und dynamisch allokierte Speicher (im Code)
- Direkte Speichermanipulation mit Pointern ist möglich → **Mögliche Speicherfehler!**
- Explizite **dynamische Allokation** von Speicher mit
 - `void * malloc(int size)`: Allokiert `size` zusammenhängende Bytes. Gibt einen Pointer auf die Adresse des ersten Bytes zurück.
 - Größe von erforderlichem Speicher kann mit `sizeof(<Datentyp>)` ermittelt werden
 - `void free(void * ptr)`: Deallokiert den Speicherblock, der bei `ptr` beginnt.
 - Aufräumen von ungenutztem Speicher nicht vergessen!
 - Keine garbage collection (wie in Java)!

- Direkte Objekt-Deklaration geschieht auf dem Stack.
Objekte sind nur für ihren Code-Block instanziert.
 - Danach zeigen Pointer auf die Objekte ins Nirvana!
- Allokation auf dem Heap mit **malloc()** und **free()**
- **Oder einfacher:** mit **new** und **delete**
 - **new:** Allokiert Speicher, führt den Konstruktor aus und liefert einen Pointer zurück
 - Beispiel: **MyClass *ptr = new MyClass();**
 - **delete:** Deallokiert Speicher des Objekts, führt den Destruktor aus
 - Beispiel: **delete(ptr);**

Aufgelistete Werte

- Manchmal möchte man eine Variable definieren, die nur eine gewisse Menge an aufgelisteten Werten halten soll
- Beispiele:
 - `dayOfWeek: SUNDAY, MONDAY, TUESDAY, ...`
 - `month: JAN, FEB, MAR, APR, ...`
 - `gender: MALE, FEMALE`
 - `title: MR, MRS, MS, DR, PROF`
- Erinnerung: Konstanten werden in ALLCAPS geschrieben
- Was ist der eigentliche Typ dieser Konstanten?

- In der Vergangenheit wurden solche Auflistungen mit Integers realisiert:

```
public final int SPRING = 0;  
public final int SUMMER = 1;  
public final int FALL = 2;  
public final int WINTER = 3;
```

- Nervig und fehleranfällig

```
int season = WINTER; ...; season = season + 1;  
int now = WINTER; ... ; month = now;
```

- Besser: enum-Typ nutzen

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```

- Ein **enum** ist eigentlich ein neuer Klassentyp
 - Kann auch optional als innere Klasse (s. später) deklariert werden
 - Liefert Typsicherheitsprüfungen zur *Compile-Zeit*
 - Jeder deklarierte Wert ist eine eigene Instanz der enum-Klasse
 - Enums sind implizit **public**, **static** und **final**
 - Enums können mit **equals** oder **==** verglichen werden
 - Enums erben von **java.lang.Enum** und implementieren **java.lang.Comparable** (d.h., Enums sind sortierbar)
 - Enums überschreiben **toString()** und bieten **valueOf()**
 - Beispiele:

```
Season season = Season.WINTER;  
  
System.out.println(season);           // schreibt WINTER  
  
season = Season.valueOf("SPRING");    // setzt season zu Season.SPRING
```

Beispiel: Enum und Switch

```
public void tellItLikeItIs(DayOfWeek day) {  
    switch (day) {  
        case MONDAY:  
            System.out.println("Mondays are bad.");  
            break;  
        case FRIDAY:  
            System.out.println("Fridays are better.");  
            break;  
        case SATURDAY:  
        case SUNDAY:  
            System.out.println("Weekends are best.");  
            break;  
        default:  
            System.out.println("Midweek days are so-so.");  
            break;  
    }  
}
```

Quelle: <http://docs.oracle.com/javase/tutorial/java/javaOO/enum.html>

- Enums bieten Typsicherheit zur Compile-Zeit
 - `int` enums bieten überhaupt keine Typsicherheit, z.B. `int season = 43;`
- Enums haben lesbare Namen für ihre Zustände
 - Bei `int` enums muss man die Zugehörigkeit der Konstanten mit Präfixen darstellen (Beispiel: `seasonWINTER` oder `S_WINTER`).
- Enums sind robust
 - Hinzufügen, Entfernen oder Umordnen von Konstanten erfordert Rekompilierung
- Weil Enums Objekte sind, können sie auch zu Collections (s. später) hinzugefügt werden
- Weil Enum-Typen Klassen sind, können sie auch Attribute und Methoden enthalten

Enum Methoden

- **String `toString()`**
 - gibt den Namen des Enum-Objektes als Konstante zurück
- **boolean `equals(Object other)`**
 - vergleicht die enum-Konstanten
- **int `compareTo(E o)`**
 - gibt eine Ordnung zum Sortieren zurück
(-1 für kleiner, 0 für gleich, 1 für größer)
- **static enum-type `valueOf(String s)`**
 - gibt das Enum-Objekt mit dem Namen `s` zurück
- **static enum-type[] `values()`**
 - gibt ein Array der Konstanten Enum-Objekte zurück

- Arrays
 - Eigenschaften
 - Lineare Folge fester Größe
 - length-Attribut
 - Kann primitive Datentypen enthalten oder Referenzen auf Objekte
 - Nur Objekte eines Typs
 - Vorteile
 - Sehr effizient
 - Typüberprüfung zur Compile-Zeit
 - Nachteile
 - Feste Größe
 - Nur ein Typ
 - Wenig Methoden

→ Container

- Ein Container ist ein Objekt das eine Sammlung von Referenzen auf andere Objekte enthält
- Ein Gruppierungsmechanismus mit komplexen Methoden zum Verwalten und Ändern der Sammlung
- Zwei unterschiedliche Konzepte von Containern

1. Collection

- Enthält einzelne Objekte
- z.B. in Java: Set, ArrayList, LinkedList

2. Map

- Enthält Schlüssel-Wert Paare (**key-value pairs**)
- Jeder Schlüssel wird auf einen Wert abgebildet
- Jeder Schlüssel darf nur einmal vorkommen
- z.B. in Java: HashMap, HashTable

- Eigenschaften
 - Können verschiedene Typen beinhalten
 - Keine feste Größe, automatische Größenanpassung
 - Enthalten nur Referenzen auf Objekte
 - Keine primitiven Typen
 - Package `java.util`
- Beispiele von Methoden
 - `add(Object o)`
 - `remove(int index)`
 - `size()`
 - `get(int index)`
 - `put(Object key, Object value) // für Map`

```
import java.util.ArrayList;  
  
// Constructs an empty list with the specified  
// initial capacity:  
ArrayList songs = new ArrayList(20);
```

```
// Add a Song:  
songs.add(new Song());
```

```
// Set the artist:
```

```
// (the cast is needed, to make clear it is a Song,  
// you could do movies inside ☺ but ...)  
((Song) songs.get(0)).setArtist("Abba");
```

ArrayList.get(i) returns a reference to an Object. The type cast (Song) is needed to tell the compiler that it is actually an instance of the Song class. This way, we can then call the method setArtist of Song.

```
/**  
 * This method returns a String description of all Songs  
 * @return String - a String containing all Song names  
 */  
  
public String getAllSongs() {  
    int i = 0;  
    String result = "";  
  
    while (i < songs.size()) {  
        result = result + ((Song) songs.get(i)).toString() + "\n";  
        i++;  
    }  
  
    if (i == 0)  
        return ("Keine Lieder vorhanden!");  
    else  
        return result;  
}
```

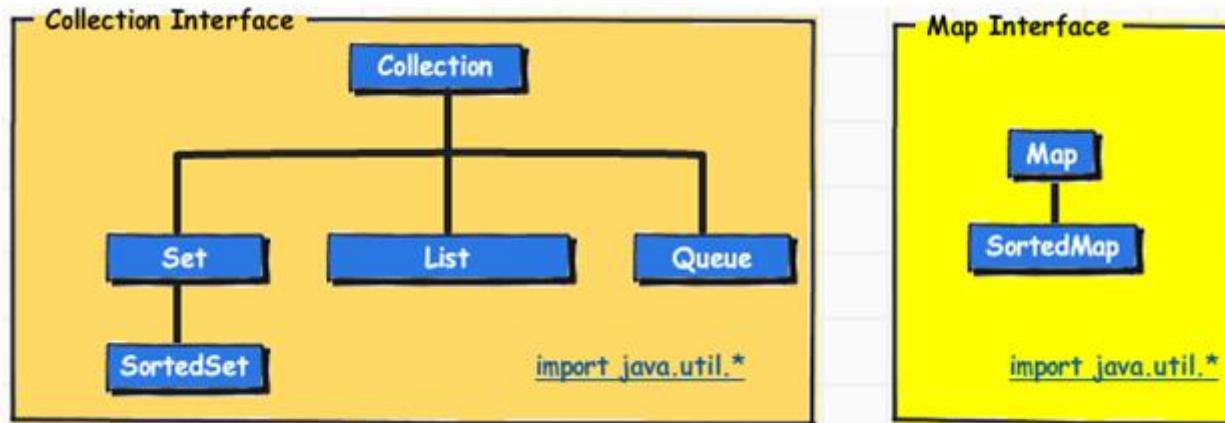
- Iteratoren: Objekte die die folgenden zwei Methoden bieten um einen Container zu durchlaufen

```
boolean hasNext()  
// Returns true if the iteration has more elements  
  
Object next()  
// Returns the next element in the iteration
```

- Wozu Iteratoren?
 - Höhere Abstraktion
 - Generischer Code
→ Änderung von z.B. `ArrayList` in `LinkedList` oder `Set` möglich
 - Alle drei Klassen bieten `iterator()` Methode an

```
public String getAllSongs(ArrayList songs) {  
    String result = "";  
  
    Iterator e = songs.iterator();  
  
    while(e.hasNext())  
        result = result + ((Song) e.next()).toString() + "\n";  
  
    if (result.equals(""))  
        return ("Keine Lieder vorhanden");  
    else  
        return result;  
}
```

- Statt mit konkreten Klassen wird meistens mit Interfaces gearbeitet
z.B. `List songs = new ArrayList(20);`
`List` ist ein Interface in `java.util`
`ArrayList` ist eine konkrete Klasse die das `List`-Interface implementiert
`LinkedList` ist eine andere Klasse die das `List`-Interface implementiert
- Die Verwendung generischer Interfaces erleichtert den Austausch von den konkreten Klassen, z.B. von `ArrayList` zu `LinkedList`
- Wichtige Interfaces: `Collection`, `List`, `Set`, `Queue`, `Map`



```
import java.util.ArrayList;  
  
// Constructs an empty list with the specified  
// initial capacity:  
ArrayList songs = new ArrayList(20);
```

```
// Add a Song:  
songs.add(new Song());
```

```
// Set the artist:
```

```
// (the cast is needed, to make clear it is a Song,  
// you could do movies inside ☺ but ...)  
((Song) songs.get(0)).setArtist("Abba");
```

ArrayList.get(i) returns a reference to an Object. The type cast (Song) is needed to tell the compiler that it is actually an instance of the Song class. This way, we can then call the method setArtist of Song.

```
/**  
 * This method returns a String description of all Songs  
 * @return String - a String containing all Song names  
 */  
  
public String getAllSongs() {  
    int i = 0;  
    String result = "";  
  
    while (i < songs.size()) {  
        result = result + ((Song) songs.get(i)).toString() + "\n";  
        i++;  
    }  
  
    if (i == 0)  
        return ("Keine Lieder vorhanden!");  
    else  
        return result;  
}
```



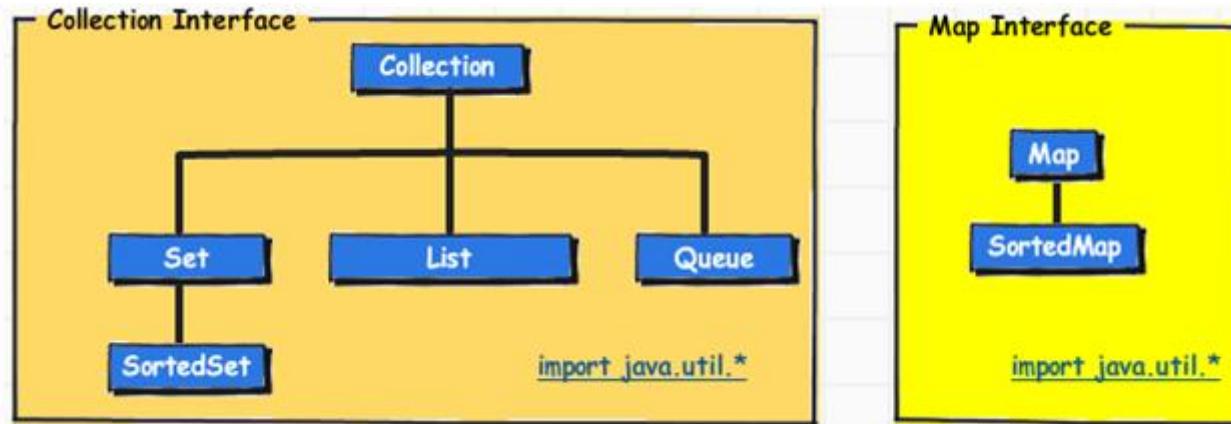
- Iteratoren: Objekte die die folgenden zwei Methoden bieten um einen Container zu durchlaufen

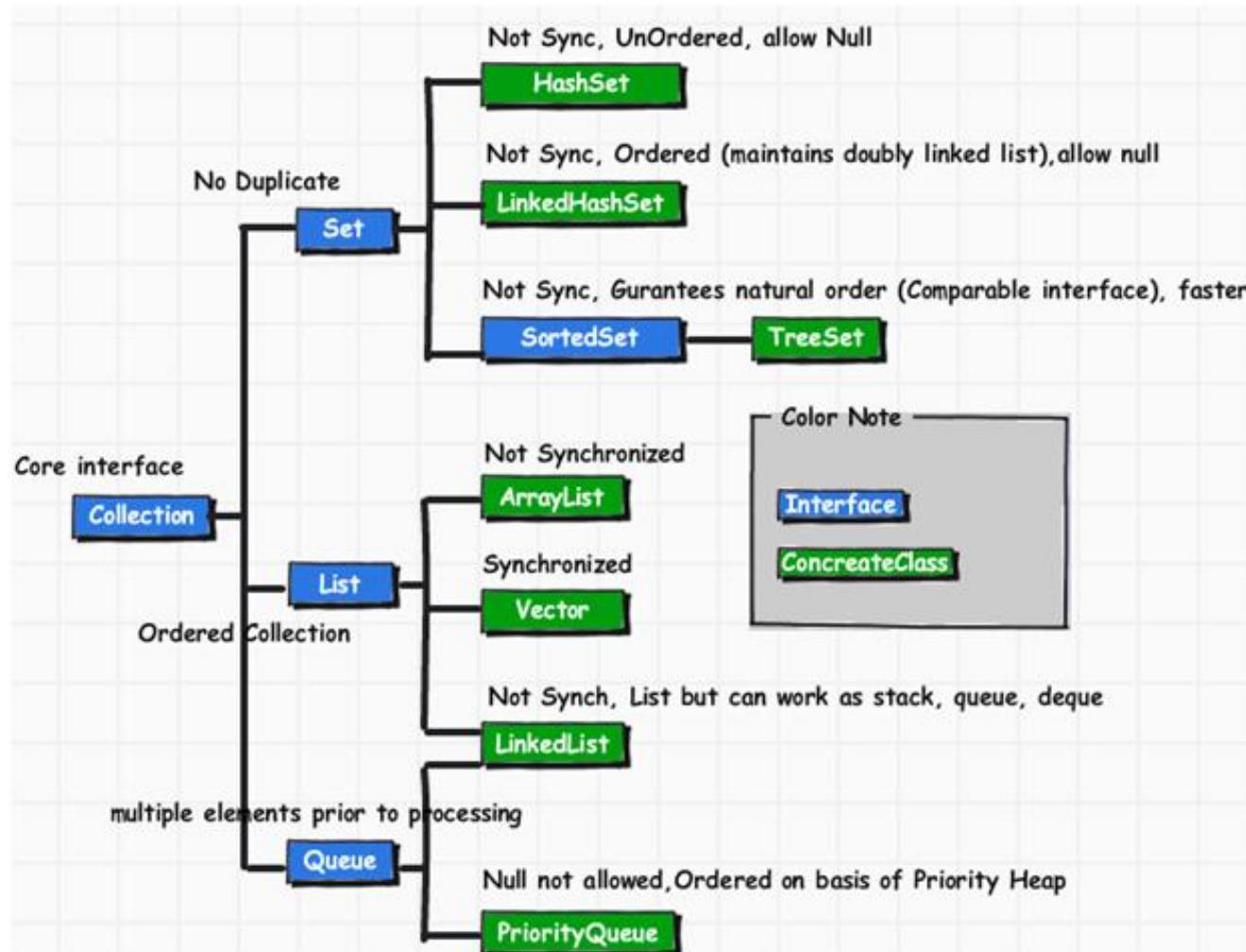
```
boolean hasNext()  
// Returns true if the iteration has more elements  
  
Object next()  
// Returns the next element in the iteration
```

- Wozu Iteratoren?
 - Höhere Abstraktion
 - Generischer Code
→ Änderung von z.B. `ArrayList` in `LinkedList` oder `Set` möglich

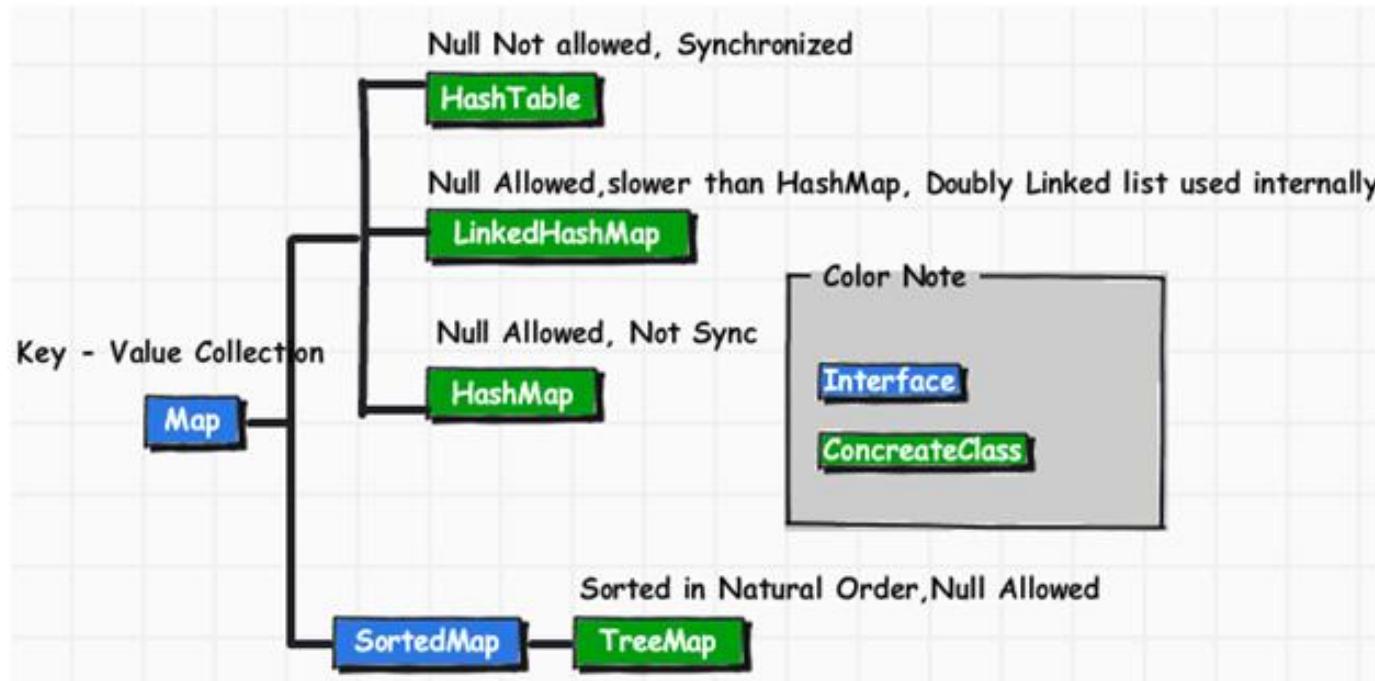
```
public String getAllSongs(ArrayList songs) {  
    String result = "";  
  
    Iterator e = songs.iterator();  
  
    while(e.hasNext())  
        result = result + ((Song) e.next()).toString() + "\n";  
  
    if (result.equals(""))  
        return ("Keine Lieder vorhanden");  
    else  
        return result;  
}
```

- Statt mit konkreten Klassen wird meistens mit Interfaces gearbeitet
z.B. `List songs = new ArrayList(20);`
`List` ist ein Interface in `java.util`
`ArrayList` ist eine konkrete Klasse die das `List`-Interface implementiert
`LinkedList` ist eine andere Klasse die das `List`-Interface implementiert
- Die Verwendung generischer Interfaces erleichtert den Austausch von den konkreten Klassen, z.B. von `ArrayList` zu `LinkedList`
- Wichtige Interfaces: `Collection`, `List`, `Set`, `Queue`, `Map`





Quelle: <http://www.jitendrzaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/>



Quelle: <http://www.jitendrzaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/>

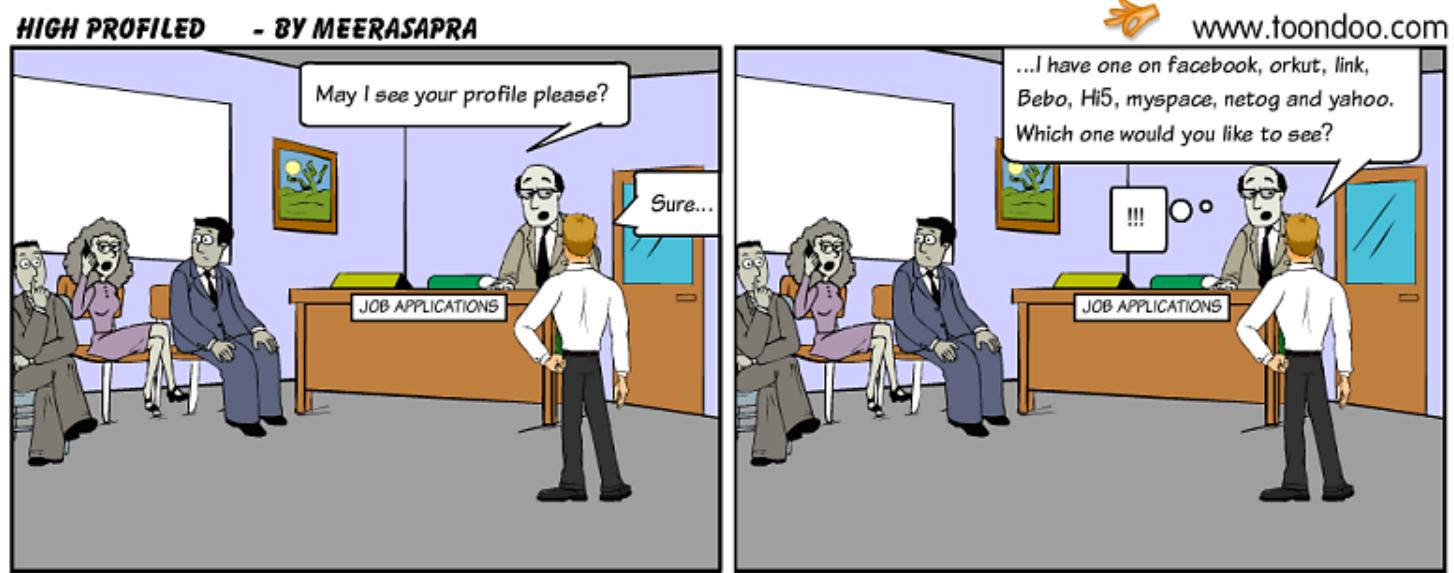
- Nachteile der Standard-Container (d.h. ohne Verwendung von Generics - s. später)
 - Weniger effizient als Arrays
 - Keine Typüberprüfung; manuelle Typumwandlungen (casts) notwendig
- Bsp. für unterschiedliche Performance (Anzahl für die Implementierung benötigter elementarer Operationen wie addieren, subtrahieren,...)
 - klassisches Array → Get:172; Iteration:516; Insert: na; Remove: na
 - ArrayList → Get:281; Iteration:1375; Insert:328; Remove:30484
 - LinkedList → Get:5828; Iteration:1047; Insert:109; Remove:16
- Wichtige funktionale Unterschiede

	HashSet	LinkedHashSet	TreeSet	ArrayList	Vector	LinkedList	HashTable	LinkedHashMap	HashMap	TreeMap
Null	✓	✓	✗	✓	✓	✓	✗	✓	✓	✓
Duplicate	✗	✗	✗	✓	✓	✓	✗	✗	✗	✗
Sorted Result	✗	✗	✓	✗	✗	✗	✗	✗	✗	✓
Retrieval order same as insertion	✗	✓	✗	✓	✓	✓	✗	✓	✗	✗

Quelle: <http://www.jitendrzaa.com/blog/java/complete-java-collection-tutorial-for-the-beginner/>

- C++ bietet viele Container-Klassen als Teil der Standard Template Library an
- Alle C++ Container sind Templates
 - Verwendung sehr ähnlich zu Java Generics (s. später)
- Beispielcontainer

C++ Container	Java Collection	Beschreibung
Vector	ArrayList	Array mit veränderbarer Länge
List	LinkedList	Doppelt verkettete Liste
Set	SortedSet	Sortierte und iterierbare Menge in der jedes Element maximal ein mal vorkommt
Map	SortedMap	Sortierte und iterierbare Tupelsammlung mit Schlüssel/Wert-Tupeln



Variable Typisierung

GENERICS

- **Hintergrund:** Alte Versionen der Java Collections (vor Java 1.5) waren *Objekt*-basiert und erforderten das Benutzen von hässlichen Typecasts
 - Typ der Elemente einer Collection kann nicht spezifiziert werden
 - Beim **Zugriff** braucht man immer **Typecast** zur spezifischen Klasse
Beispiel: `(Song) songs.get(0).setArtist("Abba");`
- **Java Generics**
 - Ermöglichen mehr Sicherheit und Lesbarkeit
 - Vor allem für allgemeine Datenstrukturen nützlich
- **Generische Programmierung** = Programmierung von Klassen und Methoden mit parametrisierten Typen

- Generische Klassen und Methoden sind ein Werkzeug zur Erstellung von wiederverwendbaren objektorientierten Typen und Bibliotheken
- Volles Verständnis erfordert Einarbeitung in Typ-Theorie
 - Insbesondere die Prinzipien der *Kovarianz* und *Kontravarianz*
- In dieser Vorlesung illustrieren wir das Prinzip hinter generischer Typisierung mit Hilfe von einigen Beispielen

- Beispiel einer **generischen Klasse**

```
class Paar<T> {  
    public T first;  
    public T second;  
    public Paar(T f, T s) { first = f; second = s; }  
    public Paar() { first = null; second = null; }  
}
```

Die Klasse **Paar** ist parametrisiert nach Typ **T**

- Die Klasse wird mit der Substitution des Typs T durch einen konkreten Typ instantiiert. Beispiel: **Paar<String>**
- Eine instantiierte generische Klasse ist wie eine ganz normale Klasse benutzbar (fast):

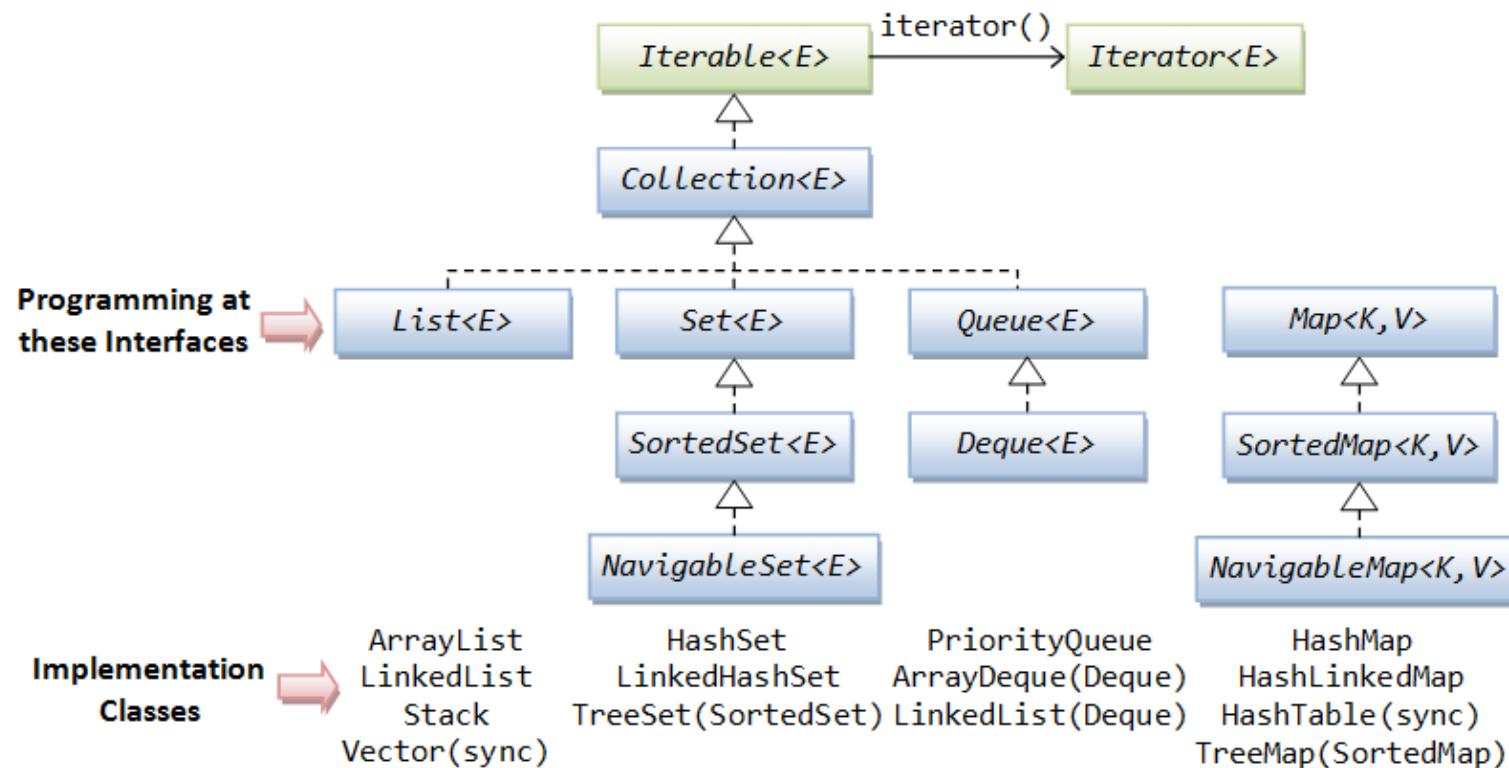
```
Paar<String> pair = new Paar<String>("1", "2");
```

- Mehrere Typ-Parametrisierungen:

```
class Paar<T, U> {  
    public T first;  
    public U second;  
    public Paar (T x, U y) { first = x; second = y; }  
    public Paar () { first = null; second = null; }  
}
```

- Zur Instantiierung
Paar<String, Number>

- Generics werden am häufigsten verwendet um mit typisierten Containern zu arbeiten
- Ab Java 5 (v1.5) werden typisierte Container-Klassen unterstützt



Quelle: http://www.ntu.edu.sg/home/ehchua/programming/java/J5c_Collection.html

Beispiel: Typisierte ArrayList

```
import java.util.ArrayList;  
  
// Constructs an empty list with the specified  
// initial capacity:  
ArrayList<Song> songs = new ArrayList<Song>(20);
```

```
// Add a Song:  
songs.add(new Song());
```

```
// Set the artist:  
songs.get(0).setArtist("Abba");
```

songs is of type ArrayList<Song> and therefore songs.get(i) returns a reference to a Song object. No type cast is needed!

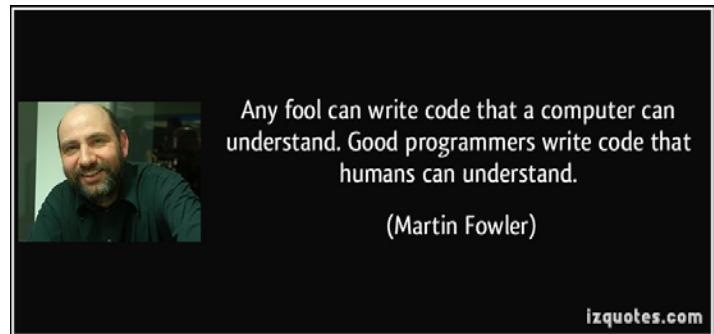
- Unterstützen *statisch typisierte* Datenstrukturen
 - *Frühe Compile-Zeit Erkennung* von Typverletzungen
 - Beispiel: Man kann keinen **String** in **ArrayList<Song>** einfügen
 - Versteckt automatisch generierte Typumwandlungen
 - Keine Typecasts notwendig
- Generische Typen sind Fabriken für *Compile-Zeit-Abkapselungen* generierter Klassen und Methoden
- *Oberfächlich betrachtet* ähnlich wie C++ Templates

- **Generische Methoden** können in normalen und generischen Klassen definiert werden

```
class Algorithms { // irgendeine Utility-Klasse
    public static <T> T getMiddle(T[] a) {
        return a[a.length / 2];
    }
    ...
}
```

- Beim Aufruf der generischen Methode kann der Typ spezifiziert werden
- String s = Algorithms.<String>getMiddle(names);
- Meist wird der Typ automatisch vom Compiler abgeleitet:
- String s = Algorithms.getMiddle(names);

- Wozu Code kommentieren?



- Problem beim Dokumentieren
 - Aktualisierung der Dokumentation
 - Code und Dokumentation oft in separaten Dateien
- Lösung: javadoc
 - Code und Dokumentation in einer Datei
 - javadoc tags z.B. @param, @return
 - Teil der JDK Installation
 - Generiert HTML Dokumentation

- Klassen, Methoden und Variablen
- Beispiel

```
/** A class comment */
public class MediaDataBase {
    /** A variable comment */
    public int i;

    /** A method comment */
    public void addSong(Song song) {
        ...
    }
}
```

- Beispiele

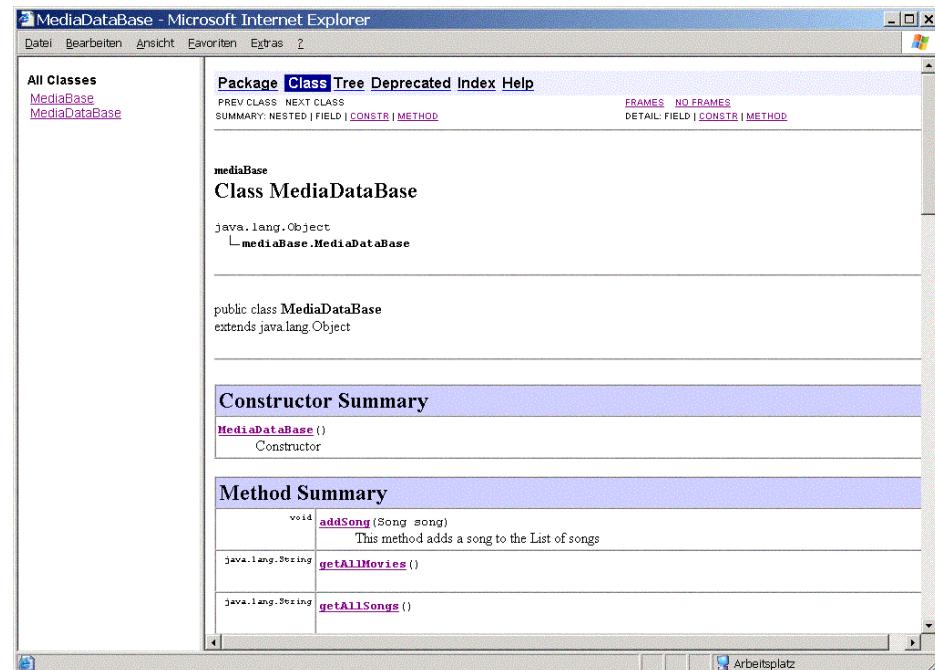
```
@author author-information
@author version-information
@param parameter-name description
@return description
@throws fully-qualified-class-name description
```

```
/**
 * This method adds a song to the List of songs
 * @param song the Song to add
 * @return void
 */
public void addSong(Song song) {
    int i;
    songs.add(song);
}
```

- Zum Erzeugen der HTML-Dokumentation von mediaBase:

```
javac MediaDataBase -d doc mediaBase mediaBase.ui  
mediaBase.model
```

- Siehe javadoc -help



- Klassen können deklariert werden
 - innerhalb einer anderen Klasse (**geschachtelte / innere Klasse**)
 - außerhalb einer Klasse, aber in eigenständiger Datei (**bevorzugt!**)
 - außerhalb einer Klasse, aber in der gleichen Datei

```
public class <MAINCLASS> {  
    // Geschachtelte Klasse  
    public static class <KLASSENNAME> {  
        // hier die einzubindenden Variablen angeben  
    }  
    // Das Hauptprogramm kann diese geschachtelten Klassen nutzen  
    public static void main(String[] args) {  
        // hier steht das eigentliche Hauptprogramm  
    }  
}
```

```
public class BeispielGeschachtelteKlasse {  
    public static class Person {  
        public String vorname;  
        public String nachname;  
        public int alter;  
        public double gehalt;  
        public String abteilung;  
        public int personalnummer;  
    }  
    public static void main(String[] args) {  
        Person otto;  
        otto = new Person();  
        otto.vorname = "Otto";  
        otto.nachname = "Meier";  
        otto.alter = 22;  
        otto.gehalt = 2345.0;  
        otto.abteilung = "F&E";  
        otto.personalnummer = 1234;  
        System.out.println(otto.alter);  
        System.out.println(otto.personalnummer);  
        // u.s.w.  
    }  
}
```

```
public class Hallo {  
    public static void main ( String[] args ) {  
        System.out.println("Hallo!");  
    }  
}
```

- Compilieren

Compiliere ...

Programm compiliert.

- Interpretieren

Programm wird ausgeführt ...

Hallo!

Programm beendet

```
public class Hallo {  
    public static void main ( String[] args ) {  
        System.outprintln("Hallo!");  
    }  
}
```



- Compilieren

- Fehlermeldungen

Tipp: Beheben Sie stets die oberste Fehlermeldung zuerst. Klicken Sie auf eine Fehlermeldung, um zur Zeile zu springen.

Haloo.java (3,11): cannot find symbol
Methode 'outprintln(java.lang.String)' in Klasse 'java.lang.System'

- Interpretieren
 - **Nicht möglich!**

```
public class Feld
{
    public static void main (String [ ] args)
    {
        int[ ][ ] feld = new int[5][ ];
        System.out.print(feld[0][0]);
    }
}
```

- Compilieren

Compiliere ...

Programm compiliert.

- Interpretieren

Programm wird ausgeführt ...

Exception in thread "main" java.lang.NullPointerException

at Feld.main(Feld.java:9)

Es sind Laufzeitfehler aufgetreten.

- Prinzipiell: Eine Ausnahmesituation
- Konkret in Java: Eine Instanz der Klasse `java.lang.Exception`
 - extends `Throwable`
- Auszug API:

[Overview](#) [Package](#) [Class](#) [Use](#) [Tree](#) [Deprecated](#) [Index](#) [Help](#)

[PREV CLASS](#) [NEXT CLASS](#)

SUMMARY: NESTED | FIELD | [CONSTR](#) | [METHOD](#)

java.lang

Class Exception

[java.lang.Object](#)
└ [java.lang.Throwable](#)
 └ [java.lang.Exception](#)

All Implemented Interfaces:

[Serializable](#)

- Manchmal ist es schwer, im Vornherein zu erkennen, welche Probleme mit den eigenen Programmen auftreten können
- Instanzen der Klasse `java.lang.Exception` helfen uns also beim Verstehen von Problemen in unserem Programm, statt es einfach „abstürzen“ zu lassen.
- Exceptions können durch den Entwickler abgefangen und damit **behandelt** werden. Dann kann das Programm wie vorgesehen weiter ausgeführt werden.
 - Dieses Behandeln von Ausnahmen nennt man **Fangen (catching)**

```
Programm wird ausgeführt ...
Exception in thread "main"
java.lang.NullPointerException
    at Feld.main(Feld.java:9)
```

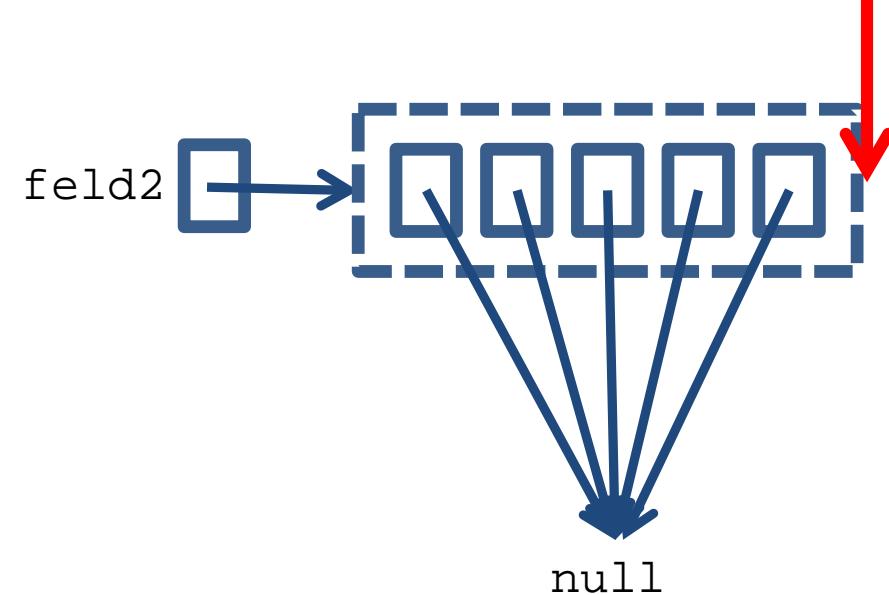
Es sind Laufzeitfehler
aufgetreten.

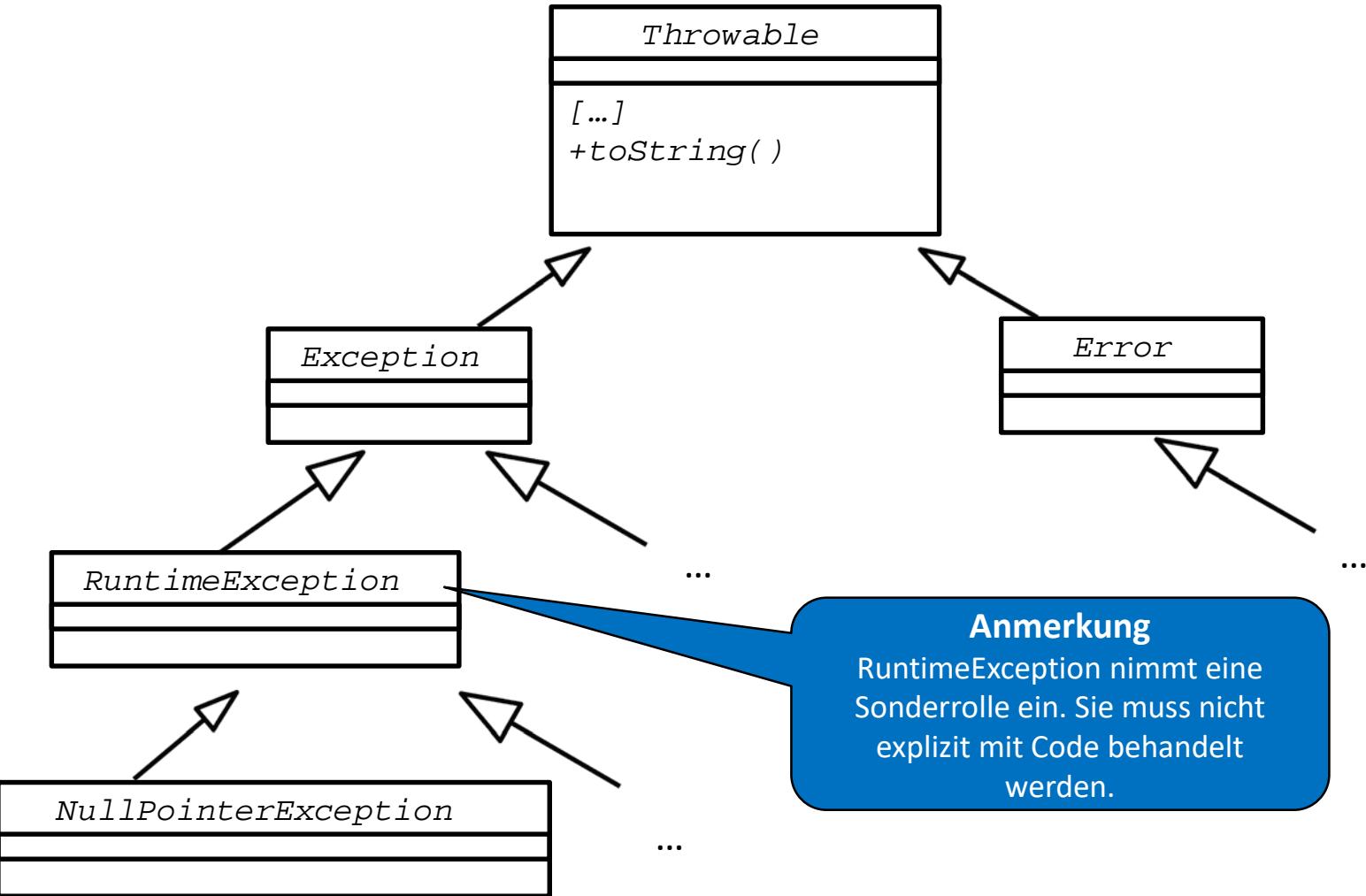
ArrayIndexOutOfBoundsException

Code

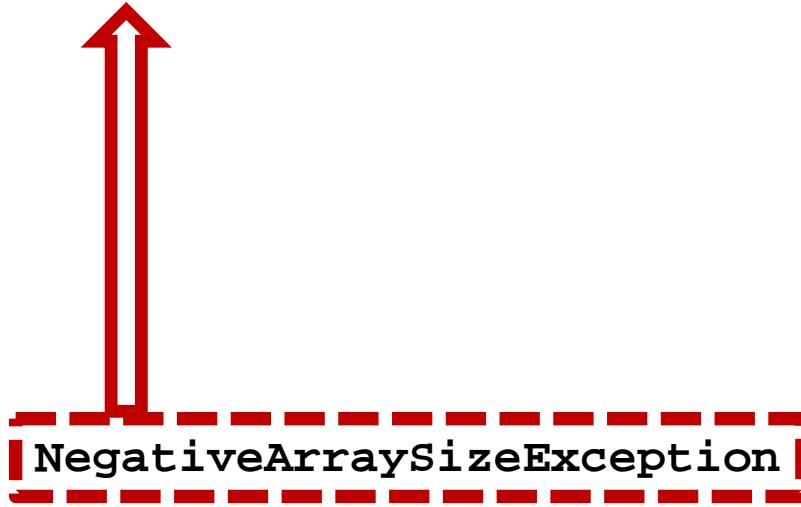
```
int[] feld2 = new int[5];
System.out.print(feld2[5]);
```

Speicher





```
int[] zfeld;  
  
Scanner in = new Scanner(System.in);  
  
System.out.print("Wie lang soll das Feld sein? ");  
  
zfeld = new int[in.nextInt()];  
  
.  
.  
.
```



Beispiel: Fangen einer Exception (2)

```
int[] zfeld;  
  
Scanner in = new Scanner(System.in);  
System.out.print("Wie lang soll das Feld sein? ");  
  
try {  
    zfeld = new int[in.nextInt()];  
}  
catch (NegativeArraySizeException e) {  
    System.out.print("Bitte nur positive Werte! Wie lang soll das Feld sein? ");  
    zfeld = new int[in.nextInt()];  
}  
.  
.  
.
```

```
int[] zfeld;  
  
Scanner in = new Scanner(System.in);  
  
System.out.print("Wie lang soll das Feld sein? ");  
  
try {  
  
    zfeld = new int[in.nextInt()];  
  
    for (int i = 0; i < zfeld.length; i++) {  
  
        zfeld[i] = i * i;  
  
    }  
  
    System.out.print(" Welches El. soll ausgegeben werden? ");  
  
    System.out.println(zfeld[in.nextInt()]);  
}  
catch (NegativeArraySizeException nase) {  
  
    System.out.print("Nur positive Werte erlaubt.");  
}  
catch (Exception e) {  
  
    System.out.print("Eine Ausnahme ist aufgetreten.");  
}  
finally {  
  
    System.out.println(" Bitte beim nächsten Programmstart beachten. Programm wird  
beendet...");  
}
```

```
int[] zfeld;  
  
Scanner in = new Scanner(System.in);  
System.out.print("Wie lang soll das Feld sein? ");  
  
try {  
  
    zfeld = new int[in.nextInt()];  
    for (int i = 0; i < zfeld.length; i++) {  
        zfeld[i] = i * i;  
    }  
    System.out.print(" Welches El. soll ausgegeben?  
    System.out.println(zfeld[in.nextInt()]);  
} catch (NegativeArraySizeException nase) {  
    System.out.print("Nur positive Werte erlaubt.");  
} catch (Exception e) {  
    System.out.print("Eine Ausnahme ist aufgetreten.");  
} finally {  
    System.out.println("Beim nächsten Programmstart  
    beendet...");  
}
```

Potenzieller Fehler 0

Negative Eingabe =>
NegativeArraySizeException

Potenzieller Fehler 1

Negative/zu Große Eingabe =>
ArrayIndexOutOfBoundsException

Catch Block 0

Wird zuerst überprüft. Eine Exception,
die hier gefangen wird, löst späteres
Catch nicht aus.

Catch Block 1

Wird danach überprüft. Hier eine
allgemeinere Exception, die u.a.
ArrayIndexOutOfBoundsException mit
fängt.

Finally

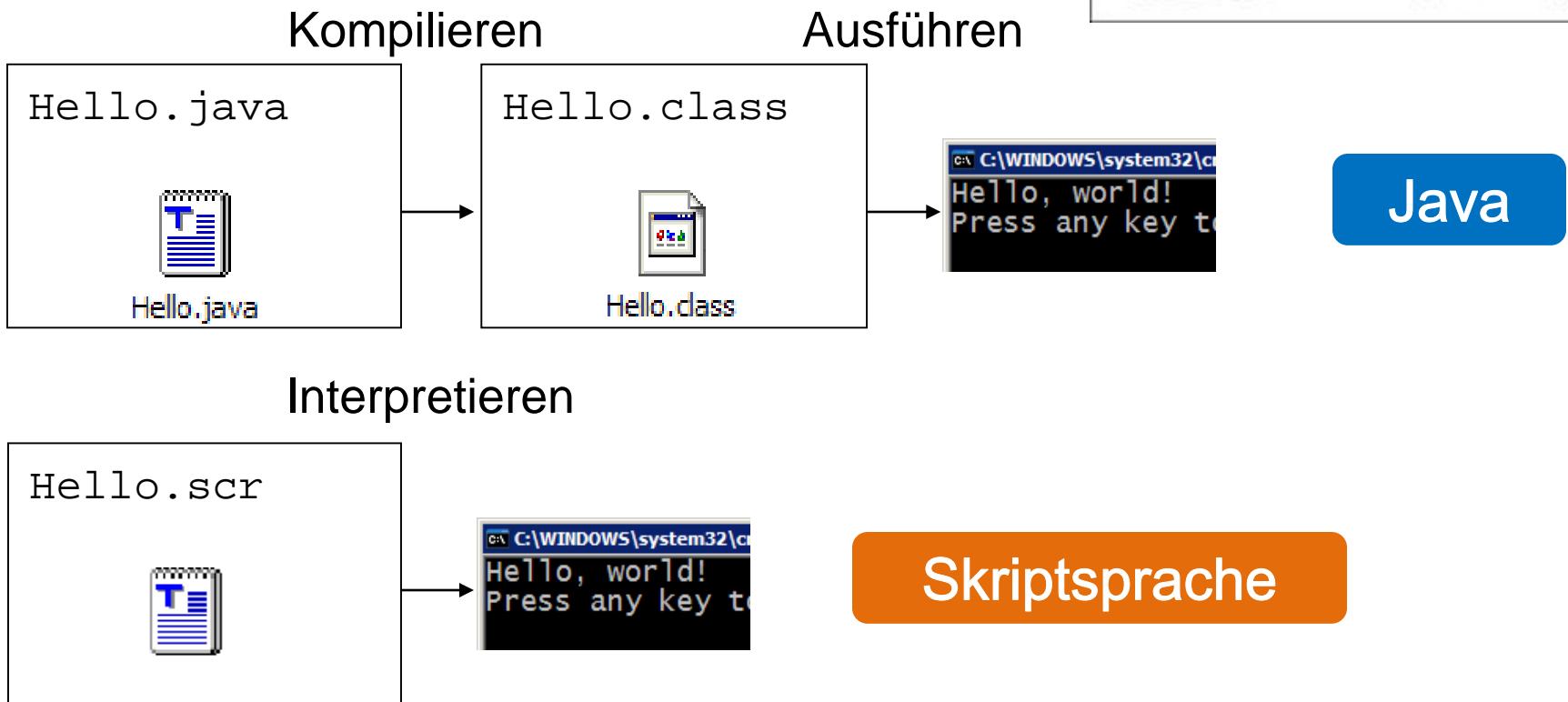
Wird immer am Ende ausgeführt.
Kann auch ohne Exceptions nötig sein,
um Verhalten am Ende zu garantieren.
Beispiel: Geöffnete Dateien schließen.

Zusammenfassung

- **Exceptions** helfen beim Aufdecken von Lücken und Problemen im eigenen Programm
- **Exceptions** können durch den Nutzer/die Nutzerin abgefangen werden (catching)
- Beim Auffangen ist die Vererbungslinie zu beachten, um die spezifisch-gewollten **Exceptions** geeignet abzufangen

Was ist ein Skript?

- Keine Kompilierung



Kompilieren vs. Interpretieren

Java

Kompilieren (javac Hello.java)

Hello.java

```
public class Hello {  
    public static void main(String[] args) {  
        System.out.println("Hello");  
        thisDoesntMakeSense;  
    }  
}
```



HelloWorld.java:4: error: not a
statement
thisDoesntMakeSense;
^
1 error

Interpretieren

Skriptsprache

Hello.scr

```
print "Hello"  
thisDoesntMakeSense
```



Hello
Error: thisDoesntMakeSense

Der Python-Interpreter

```
% python
Python 2.7.5 (default, Mar 9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 3+3
6
>>> Control + D
```

Ausgang

Eingabeauswertung

oder

Python 2

```
Summe.py
print 3+3
```

Ausgabe

Python 3

```
Summe.py
print(3+3)
```

Ausgabe

```
% python Summe.py
```

6

- Iteratoren werden benutzt um Collections in Schleifen zu durchlaufen.
- `hasNext()` überprüft auf ein nächstes Element
- `next()` gibt das nächste Element zurück

```
public String getAllSongs(ArrayList<String> songs) {  
    String result = "";  
  
    Iterator e = songs.iterator();  
    while (e.hasNext()) {  
        String song = e.next;  
        result = result + song + "\n";  
    }  
    if (result.equals(""))  
        return ("Keine Lieder vorhanden");  
    else  
        return result;  
}
```

- Vereinfachte Notation für die Schleife
- Iterator wird durch einen Doppelpunkt (:) ersetzt
- Schleife wird zu einer for-Schleife

```
String result = "";
```

```
Iterator e = songs.iterator();
```

```
while (e.hasNext()) {
```

```
    String song = e.next;
```

```
    result = result + song + "\n";
```

```
}
```

```
String result = "";
```

```
for (String song : songs) {
```

```
    result = result + song + "\n";
```

```
}
```

- Funktioniert mit Collections und Arrays

- Fundamentale Programmierstile
- Logische Prinzipien hinter der Organisation einer Programmiersprache
- Schließen einander nicht aus
- Einige typische Paradigmen:
 - Imperativ
 - Deklarativ
 - Prozedural
 - Objektorientiert
 - Ereignisorientiert
 - Logisch
 - Funktional

- Imperative Programmierung:
 - Abfolge von Befehlen, die der Computer auszuführen hat
 - Compiler arbeitet Befehle ab, übersetzt in Maschinenbefehle
 - Vergleichbar mit: Möbelbau-Anleitung
 - Sprachen: Assembler, C, Java, ...
- Deklarative Programmierung:
 - Menge von Definitionen, Relationen und Regeln
 - Ergeben einen Bedingen Raum über die Eingabewerte
 - Befehle sind implizit aus den Relationen und Regeln extrahierbar
 - Vergleichbar mit: Mathematischen Formeln
 - Sprachen: SQL, Prolog, ...

- Imperativ
 - Zerlege Anweisungsfolgen in Teilfolgen
 - Anweisungsfolgen sind nach Außen durch Schnittstellen definiert
 - Teilfolgen können sich untereinander aufrufen
 - Parameter erlauben das Weiterreichen von Informationen zwischen Anweisungsfolgen
 - Spezialfall: Rückgabewert ermöglicht die Definition von Prozeduren als Funktion
- Sprachen:
 - Fortan
 - C
 - COBOL
 - ...

- Objekte instanziiieren Objekt-Templates (Klassen)
 - Haben Eigenschaften (Attribute) und können Aktionen ausführen (Methoden)
- Objekte werden häufig zur Modularisierung des Codes verwendet.
- Methoden werden häufig nach dem prozeduralen Paradigma definiert.
- Sprachen:
 - Java
 - C#
 - C++
 - ...

- Sammlung von Ereignisbehandlungs routinen
- Routinen werden automatisch aufgerufen, wenn das passende Ereignis auftritt. Können andere Ereignisse auslösen.
- Gut mit objektorientierten Sprachen vereinbar
- Typischer Anwendungsfall: GUI-Programmierung
- Sprachen:
 - Visual Basic
 - Pascal
 - ...

- Deklarativ
 - Beruht auf mathematischer Logik
 - Menge von Axiomen
 - Interpreter berechnet Lösungsaussage einer Anfrage aus den Axiomen
- Beispiel in natürlicher Sprache:

Fakten:

Lucia ist die Mutter von Minna.

Lucia ist die Mutter von Klaus.

Minna ist die Mutter von Nadine.

Regel:

Falls X ist die Mutter von Y und Y ist die Mutter von Z Dann X ist die Großmutter von Z.

Frage/Ziel:

Wer ist die Großmutter von Nadine?

Antwort des Computers, Folgerung aus den Fakten und Regeln:

Lucia

- Bekannteste Sprache: Prolog

