

# Einführung in das Programmieren mit Java

Vortragende: Michaela Pum

# Grundlagen des Programmierens

»» Erste Schritte mit Java

# Was ist Programmieren

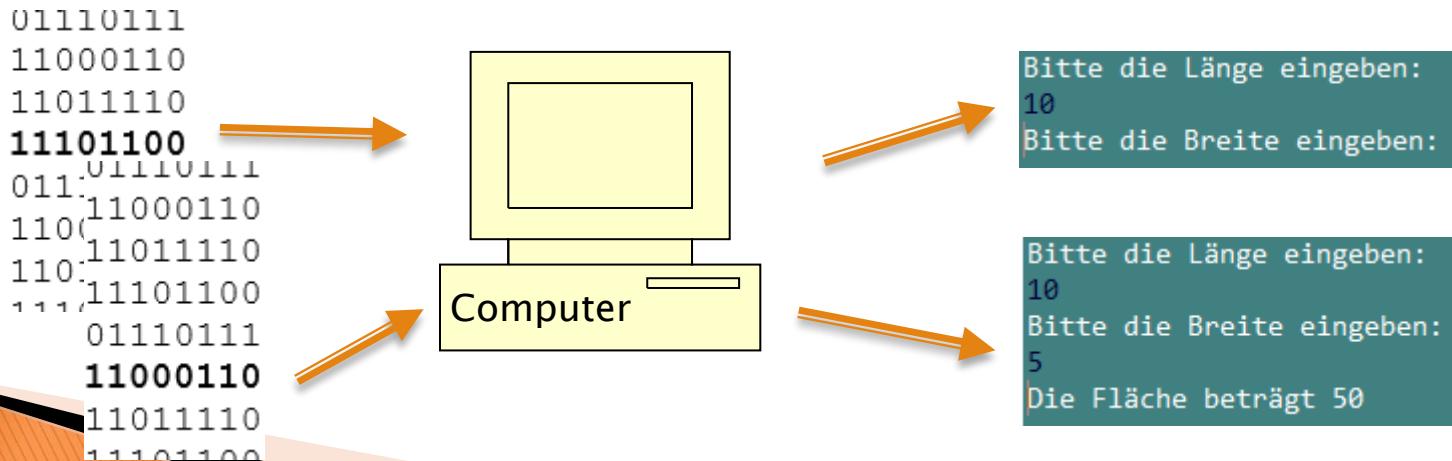
## ► Ein Computer

- ist ein Objekt das bestimmte Befehle versteht und ausführt

- versteht nur Befehle in "Maschinensprache"

## ► Ein Computerprogramm

- ist eine Abfolge von Befehlen, welche der Computer der Reihe nach interpretiert und ausführt



# Was ist Programmieren

- ▶ Mit einer Programmiersprache
  - lassen sich die Befehle verständlicher darstellen.
- ▶ Ein Compiler (Übersetzer)
  - übersetzt die Befehle von der Programmiersprache in die Maschinensprache

```
Scanner scanner = new Scanner(System.in);  
  
int l, b;  
System.out.println("Bitte die Länge eingeben:");  
l = scanner.nextInt();  
System.out.println("Bitte die Breite eingeben:");  
b = scanner.nextInt();  
scanner.nextLine();  
  
int flaeche = l * b;  
System.out.printf("Die Fläche beträgt %d \n", flaeche);  
scanner.close();
```



```
01110111  
11000110  
11011110  
11101100  
01110111  
11000110  
11011110  
11101100
```

# Datentypen

- ▶ Programme arbeiten mit verschiedenen Arten von Werten, z.B.
  - ganze Zahl: 10
  - Fließkommazahl: 3.14
  - Zeichenfolge: "Hallo Java"
  - Zeichen: 'a'
  - Wahrheitswert: true
- ▶ Der genaue Typ wird durch den **Datentyp** bestimmt
  - z.B. int oder boolean

# Tabelle: Primitive Datentypen

Typ	Inhalt	Größe
<b>boolean</b>	Wahrheitswert (true / false)	1 Bit
<b>char</b>	ein Unicode-Zeichen	16 Bit
<b>byte</b>	Ganzzahl	8 Bit
<b>short</b>	Ganzzahl	16 Bit
<b>int</b>	Ganzzahl	32 Bit
<b>long</b>	Ganzzahl	64 Bit
<b>float</b>	Fließkommazahl	32 Bit
<b>double</b>	Fließkommazahl	64 Bit

# Datentypen

Ganzzahl

Binär	Dezimal	Hex
00000000 00000000 00000000 00011011	27	0x00000001B
11111111 11111111 11111111 11100101	-27	0xFFFFFE5
10000000 00000000 00000000 00011011	-2147483621	0x80000001B

Vorzeichen

Fließkommazahl

Binär	Dezimal
00111110 10011001 10011001 10011010	0.3
10111110 10011001 10011001 10011010	-0.3

VZ

Exponent

Mantisse

# Variablen

- ▶ Behälter zur temporären Ablage von Daten
  - Datentyp (-> bestimmten Wertebereich)
  - zu einem Zeitpunkt genau einen Wert
  - Namen für den Zugriff
  - bestimmten Platz im Hauptspeicher
- ▶ Wert kann im Programmverlauf geändert werden



# Konstante

- ▶ Werte, die im Programmablauf nicht verändert werden können
  - Literale
    - Ganzzahlen: 10
    - Fließkommazahlen: 3.14
    - Zeichenfolgen: "Hallo C#"
    - Zeichen: 'a'
    - Wahrheitswert: true
  - mit Namen deklariert und zur Übersetzungszeit fixierte konstante Werte

```
final int standardPreis = 500;
```

# Tabelle: Wichtige Operatoren

	Bedeutung
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
=	Zuweisung
+ - * / % =	Abkürzung für Operation plus Zuweisung
++	Inkrement
--	Dekrement

	Bedeutung
==	gleich
!=	ungleich
<	kleiner
>	größer
<=	kleiner oder gleich
>=	größer oder gleich
&&	logisches UND
	logisches ODER
!	logisches NOT

# Logische Operatoren

Operation	Ausdruck a	Ausdruck b	Ergebnis
$!a$ (NOT)	false true		true false
$a \&& b$ (AND)	false false true true	false true false true	false false false true
$a    b$ (OR)	false false true true	false true false true	false true true true

# Operatoren – Prioritäten

Die grauen Klammern zeigen, wie die Operatoren standardmäßig priorisiert werden

## Priorität (hoch nach niedrig)

. () [] ++ -- new

! (Typ)

\* / %

+ -

< > <= >=

== !=

&& ||

= += -= \*= /= %=

a + b \* c  
a + (b \* c)

a + b < c + d  
(a + b) < (c + d)

a < 1 || a > 12  
(a < 1) || (a > 12)

b1 || b2 && b3  
b1 || (b2 && b3)

!b1 && b2  
(!b1) && b2

(int)a \* b  
((int)a) \* b

# Boole'sche Umwandlungen

! (Ausdruck)	Umgewandelt
!(a == b)	a != b
!(a != b)	a == b
!(a < b)	a >= b
!(a > b)	a <= b
!(a <= b)	a > b
!(a >= b)	a < b

## Vergleichsoperatoren

! (Ausdruck)	Umgewandelt
! (b1 && b2)	!b1    !b2
! (b1    b2)	!b1 && !b2

## Logische Operatoren

```
if( monat >= 1 && monat <= 12) { /*gültig*/ ... }  
if(! (monat >= 1 && monat <= 12)) { /*ungültig*/ ... }  
if( monat < 1 || monat > 12) { /*ungültig*/ ... }
```

# Java Programm

Variablen f. d.  
Benutzerinput

Berechnung  
(Operation)

Kommentar (gehört nicht zu  
den Programm-Befehlen)

```
// Programm zur Berechnung der Fläche eines Rechtecks
Scanner input = new Scanner(System.in);

int laenge, breite;
System.out.println("Bitte die Länge eingeben:");
laenge = input.nextInt();
System.out.println("Bitte die Breite eingeben:");
breite = input.nextInt();

int flaeche = laenge * breite;
System.out.printf("Die Fläche beträgt %d \n", flaeche);
input.close();
```

# Algorithmische Grundstrukturen

» Programmablauf steuern

# Algorithmus

- ▶ Ist eine Handlungsanweisung, bestehend aus elementaren Befehlen (Anweisungen)
  - Sequenz (nacheinander)
    - eine Anweisung wird nach einer anderen ausgeführt
  - Alternative
    - ein Teil des Algorithmus wird nur ausgeführt, wenn ...
  - Schleife (wiederholt ausführen)
    - ein Teil des Algorithmus wird wiederholt ausgeführt, und zwar solange ...

# Sequenz



```
Scanner input =  
    new Scanner(System.in);  
int laenge, breite;  
System.out.println("Länge:");  
laenge = input.nextInt();  
System.out.println("Breite:");  
breite = input.nextInt();
```

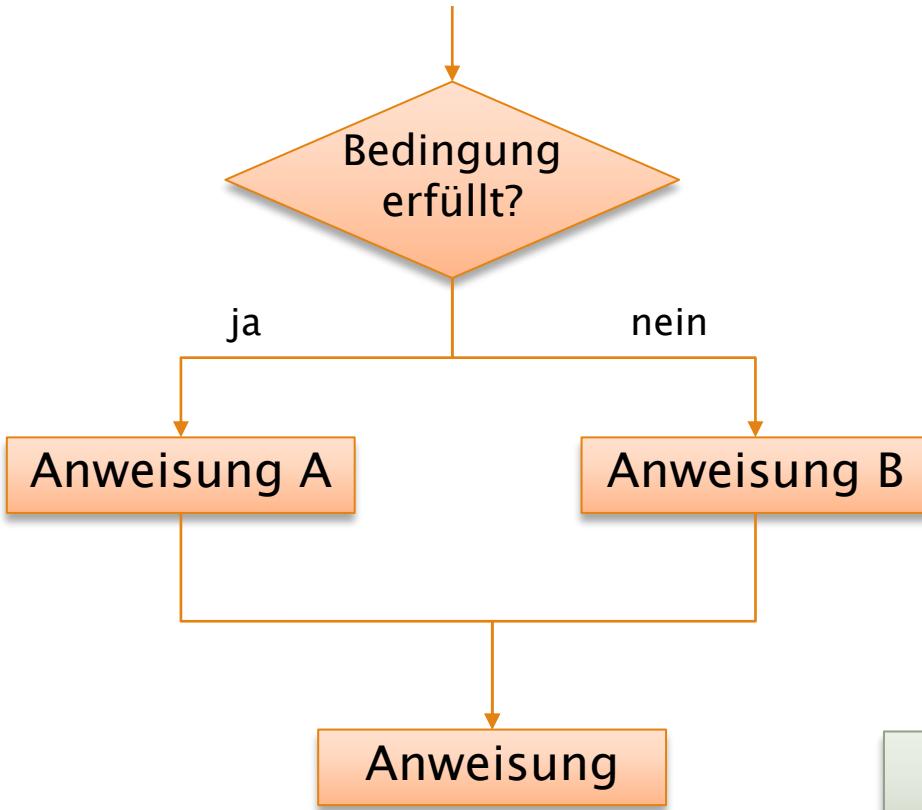
Beispiel Fläche berechnen: Aus- und Eingaben müssen in der richtigen Reihenfolge sein!

# Verzweigung (Alternative)

```
if (Boole'scher Ausdruck)
    Anweisung oder Anweisungsblock
    if (Boole'scher Ausdruck)
        Anweisung oder Anweisungsblock
    else
        Anweisung oder Anweisungsblock
```

```
int i = ...;
if (i != 0) {
    .......; // Anweisung(en) für den True-Fall
}
else {
    .......; // Anweisung(en) für den False-Fall
}
```

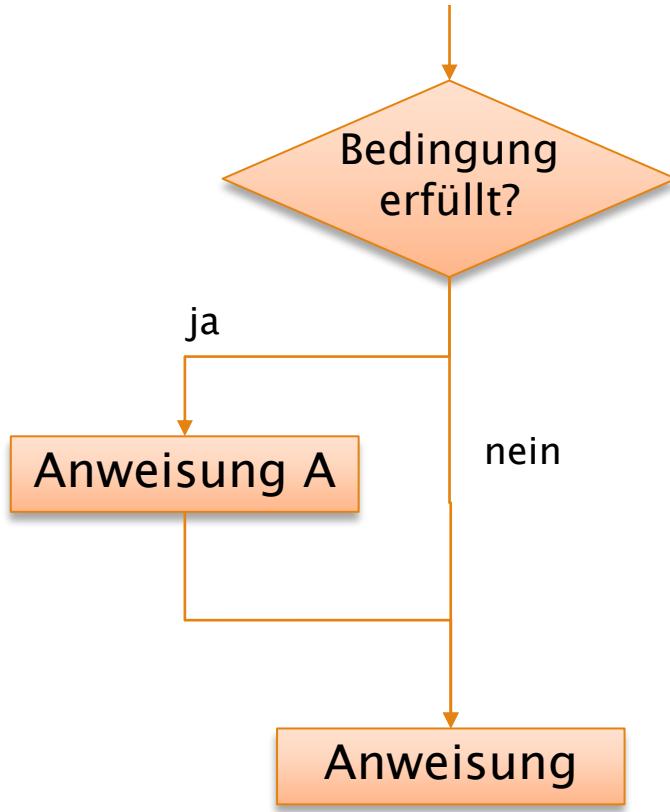
# Verzweigung (Alternative)



```
int preis;  
char student = ...;  
// wenn es ein Student ist  
if (student == 'j') {  
    // Studenten-Preis  
    preis = 400;  
}  
else {  
    // sonst: Normal-Preis  
    preis = 500;  
}
```

Beispiel Fitnesscenter: Testen ob der Kunde Student ist

# Verzweigung (Alternative)



```
int preis;  
...  
int jahre = ...;  
// für Stammkunden  
if (jahre > 3){  
    // Rabatt abziehen  
    preis -= 50;  
}
```

Beispiel Fitnesscenter: Testen ob es ein Stammkunde ist

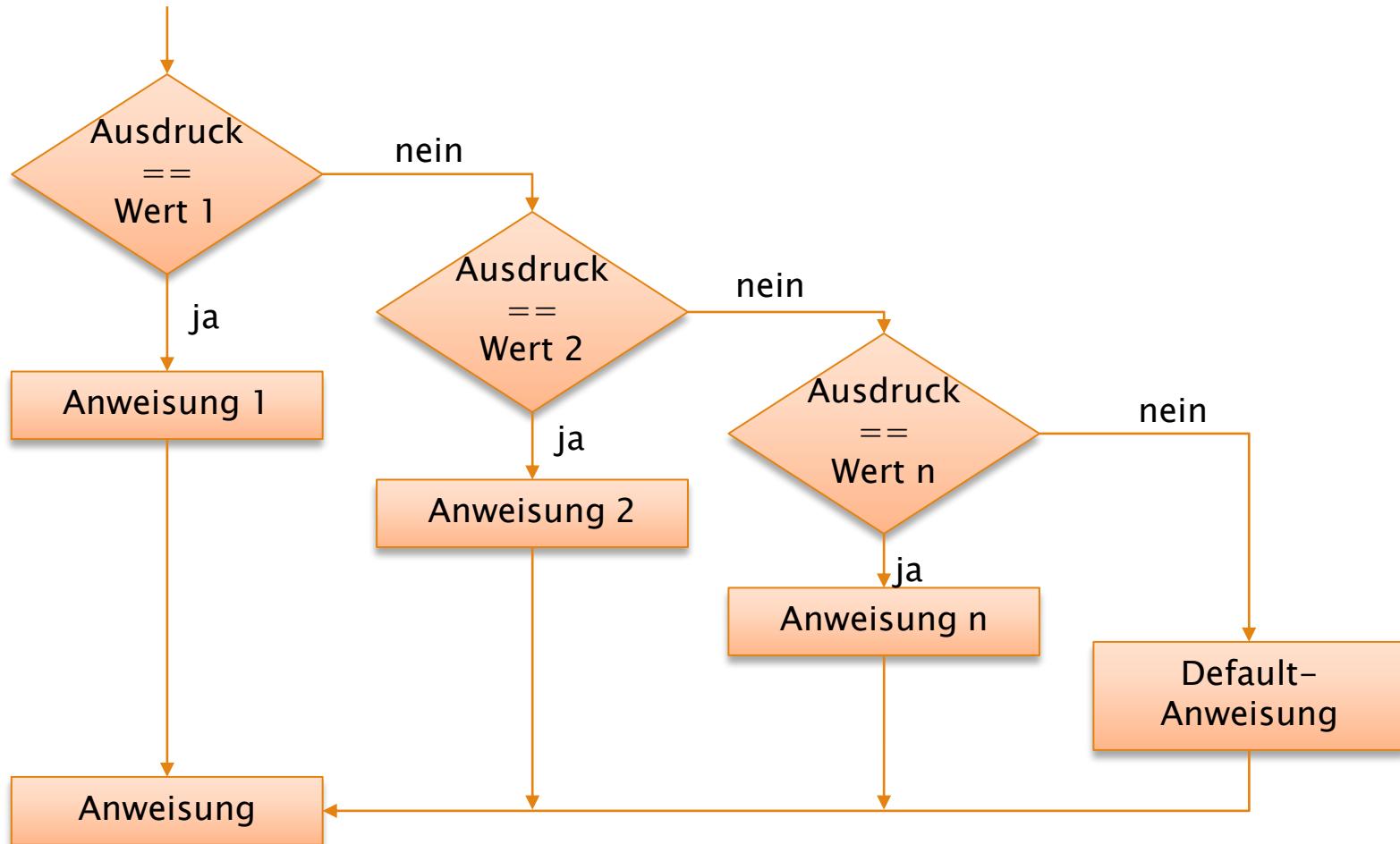
# Fallunterscheidung

```
switch (Ausdruck) {  
    case Konstante1:  
    case Konstante2:  
        Anweisung(en)  
        break;  
    case Konstante3:  
        Anweisung(en)  
        break;  
    default:  
        Anweisung(en)  
        break;  
}
```

```
int i = ...;  
switch(i){  
    case 1:  
    case 2:  
        ....;  
        break;  
    case 3:  
        ....;  
        break;  
    default:  
        ....;  
        break;  
}
```

- für ganzzahlige Ausdrücke und Zeichenfolgen
- break am Ende eines Blocks nicht vergessen!

# Fallunterscheidung



# Wiederholung (Iteration)

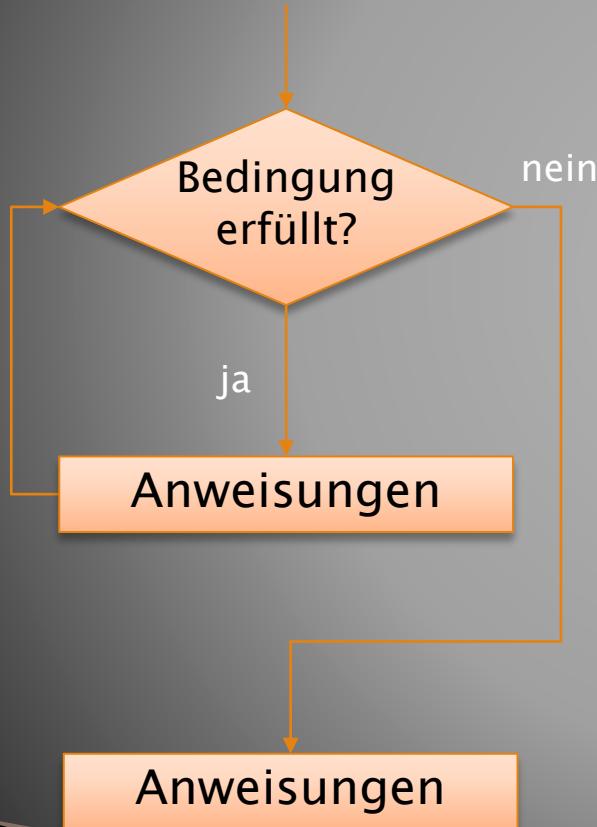
```
while (Boole'scher Ausdruck)
      Anweisung oder Anweisungsblock
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt

```
int i;
...
while(i < 9) {
    // Block wird ausgeführt solange i < 9 ist
    ....;
    i = ....;
    ....;
}
```

# Wiederholung (Iteration)

## ► while-Schleife



Bedingung steht im Schleifenkopf

```
int zahl = ...;
int ziffSum = 0;
// solange zahl ungleich 0
while (zahl != 0) {
    // berechnen der ziffer
    int ziffer = zahl % 10;
    // ... zur Summe hinzuzählen
    ziffSum += ziffer;
    // ziffer entfernen
    zahl /= 10;
}
```

Beispiel Ziffernsumme: letzte Ziffer dazuzählen solange die Zahl ungleich 0 ist

# Wiederholung (Iteration)

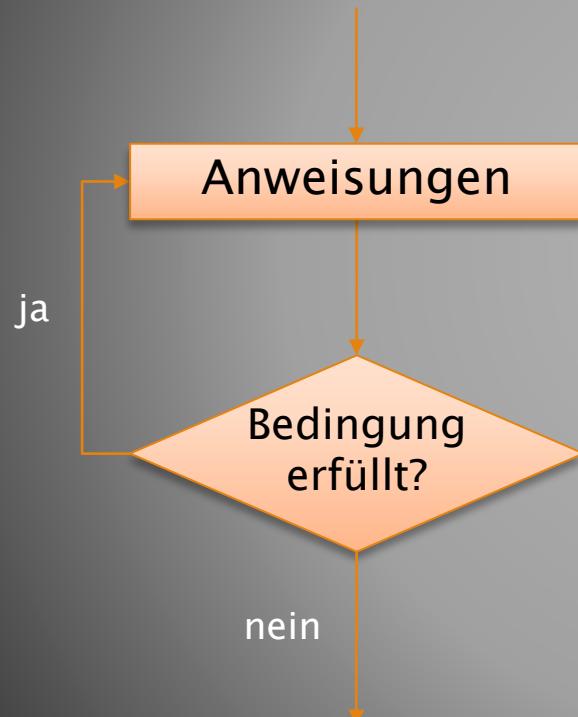
```
do
    Anweisung oder Anweisungsblock
while (Boole'scher Ausdruck);
```

- Anweisung bzw. Block wird 1 bis n Mal ausgeführt

```
int i;
...
do {
    ....;
    i = ...;
    ....;
} while(i < 9); // solange i < 9 ist, gibt es noch eine Wiederholung
```

# Wiederholung (do-while)

- do-while Schleife



```
Scanner input = new  
Scanner(System.in);
```

```
int monat;
```

```
do {  
    System.out.println(  
        "welches Monat (1-12)?");  
    monat = input.nextInt();  
} while (monat < 1 || monat > 12);
```

```
System.out.printf(  
    "Das Monat ist %d\n", monat);
```

Bedingung steht  
im Fuß der Schleife

Beispiel Benutzereingabe: Wiederholen  
solange der Wert ungültig ist

# Wiederholung (for)

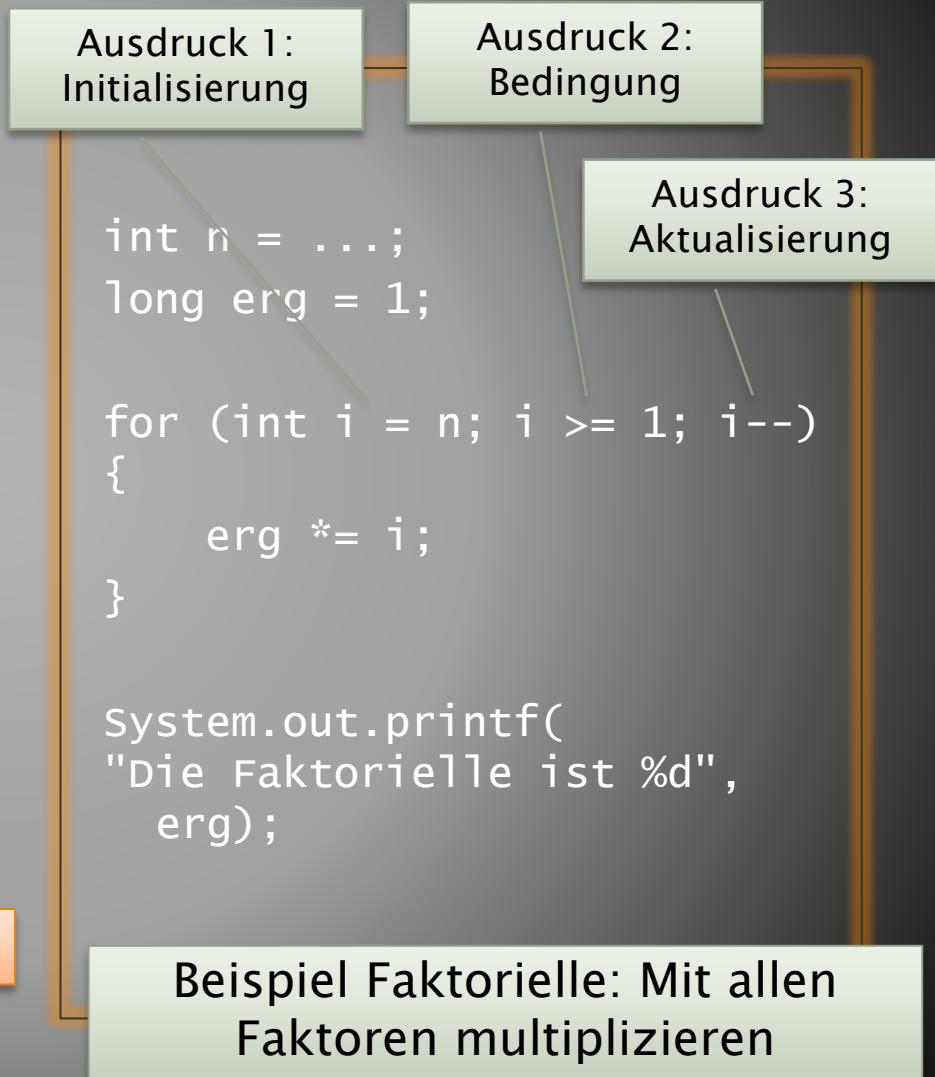
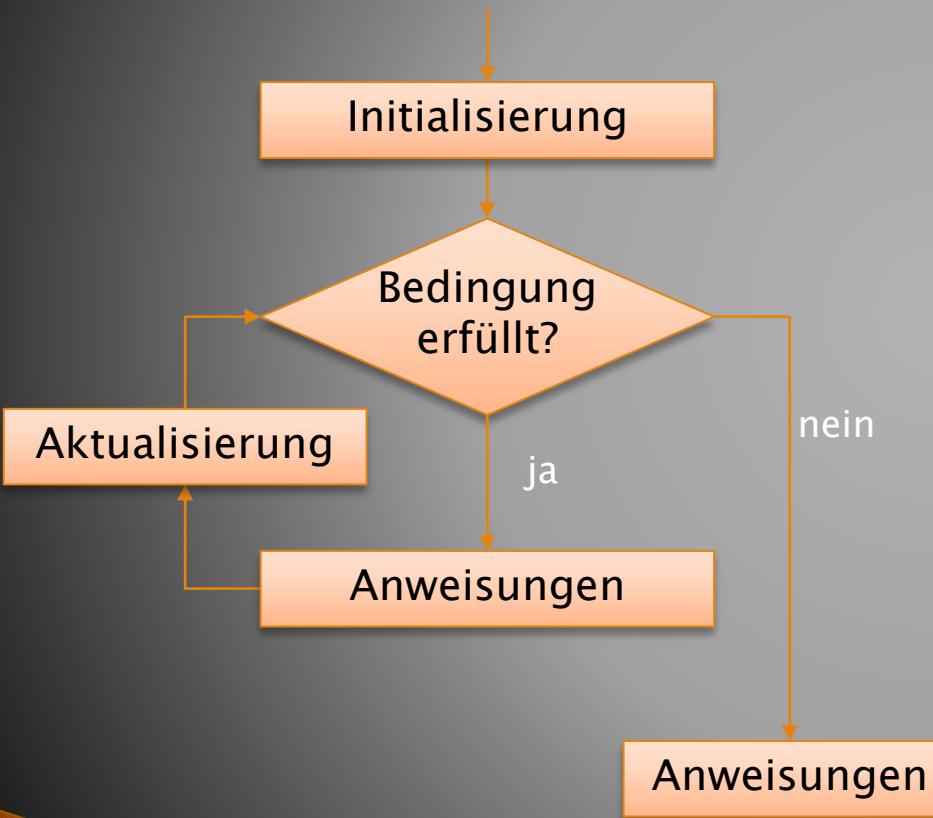
```
for (<init>;<bedingung>;<aktualisierung>)
    Anweisung oder Anweisungsblock
```

- Anweisung bzw. Block wird 0 bis n Mal ausgeführt
- die Variable i gilt nur in der for-Schleife

```
for (int i=1; i<=10; i++) {
    // Block wird ausgeführt solange i <= 10 ist
    ....;
    ....;
}
```

# Wiederholung (for-Schleife)

## for-Schleife



# Kontrollstrukturen – break

```
int i;  
...  
  
switch(i){  
    case 1: ....; break;  
    case 2: ....;  
            ....; break;  
    case 3: ....; break;  
    default: ....;  
}  
...  
...
```

erlaubt im switch und  
in allen Schleifen

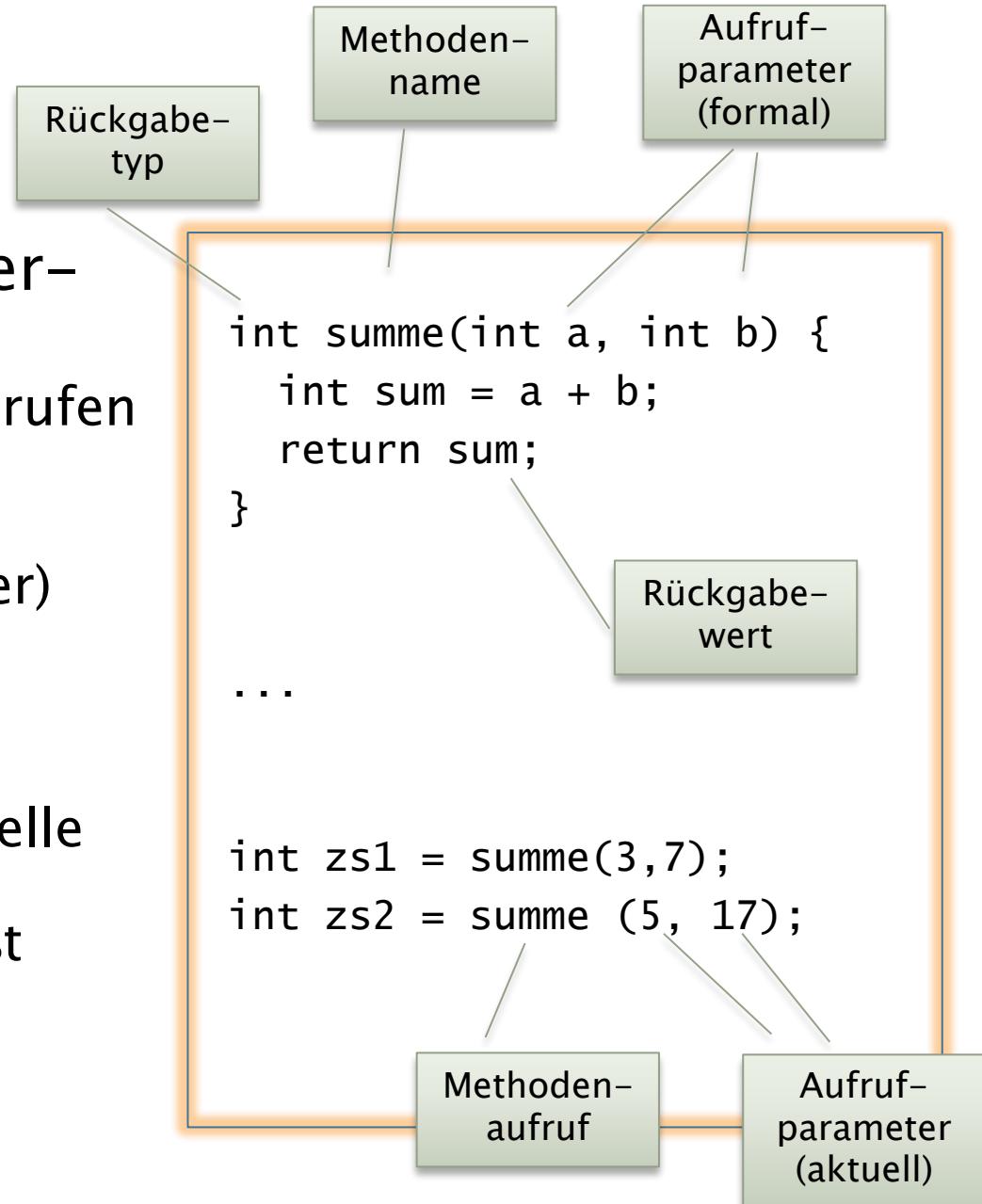
# Kontrollstrukturen – continue

```
int x = 0;  
  
do {  
  
    if (++x < 10)  
        continue; ←  
  
    System.out.println("x ist " + x);  
} while (x < 10); ←  
  
...
```

erlaubt in Schleifen

# Methode

- ▶ Definiert einen Unter-Algorithmus
  - kann mehrfach aufgerufen werden
  - kann beim Aufruf Argumente (Parameter) erhalten
  - kann einen Wert zurückliefern
  - Die formale Schnittstelle (Rückgabetyp, Name, Parametertypen) heißt **(Methoden-)Signatur**



# Methoden überladen (overload)

## ► mehrere Methoden

- haben denselben Namen
- aber Unterschiede in der Parameterliste:
  - Anzahl der Parameter
  - Typen der Parameter an einer Position

```
int summe(int a, int b) {  
    return a + b;  
}  
int summe(int a, int b, int c) {  
    return a + b + c;  
}  
...  
int s1 = summe(20, 12);  
int s2 = summe(13, 17, 25);
```

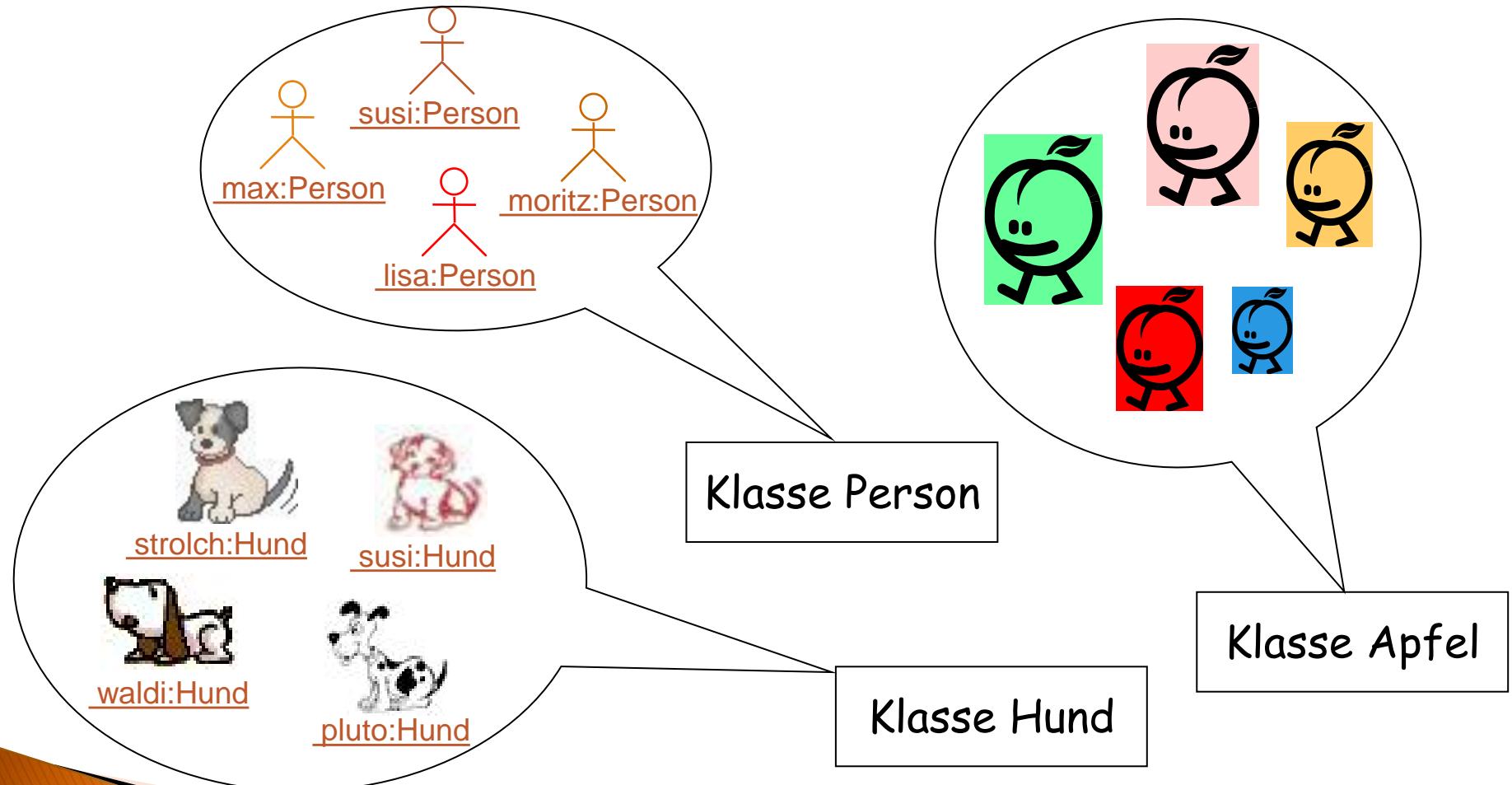
Überladungen der  
Methode "summe"

Compiler unterscheidet je  
nach Anzahl/Typen der  
aktuellen Parameter

# Objektorientierte Konzepte

» Klassen und Objekte  
definieren und verwenden

# Klassen und Objekte



# Klassen und Objekte

**Klasse  
(Class)**



**Typ**

**Objekt  
(Object)**



**Wert**

- Datentyp
- Enthält Attribute und Methoden
- Schablone für Objekte

- Exemplar ("Instanz") einer Klasse ("Wert")

# Klassen und Objekte

**Klasse**



**String**

**Objekt**



**"Herr Nilsson"**

# Klassen und Objekte

**Klasse**



**Person**

**Objekt**



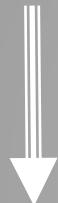
**Name: Susi**

**Alter: 12**

**Hobby: Reiten**

# Klassen und Objekte

**Klasse**



**Hund**

**Objekt**



**Name: Struppi**  
**MarkenNr:**  
**W4711**

# Klassen und Objekte

- ▶ Eine **Klasse** beschreibt eine Menge von Objekten mit gleicher Struktur (Attributen), gleichem Verhalten (Methoden)
- ▶ Die **Attribute** (Felder) einer Klasse definieren den Status ihrer Objekte
- ▶ Die **Methoden** einer Klasse definieren die Fähigkeiten ihrer Objekte
- ▶ Jede **Klasse** hat einen Namen
- ▶ Jede **Klasse** muss ausführlich dokumentiert werden

# Klassen und Objekte

Es soll eine Klasse erstellt werden,  
welche eine Uhrzeit als Stunde und  
Minute kapselt.

Die Werte für Stunde und Minute  
müssen gesetzt werden können.  
Eine beliebige Anzahl von Minuten soll  
dazugezählt werden können

# Klassendefinition

Klassename

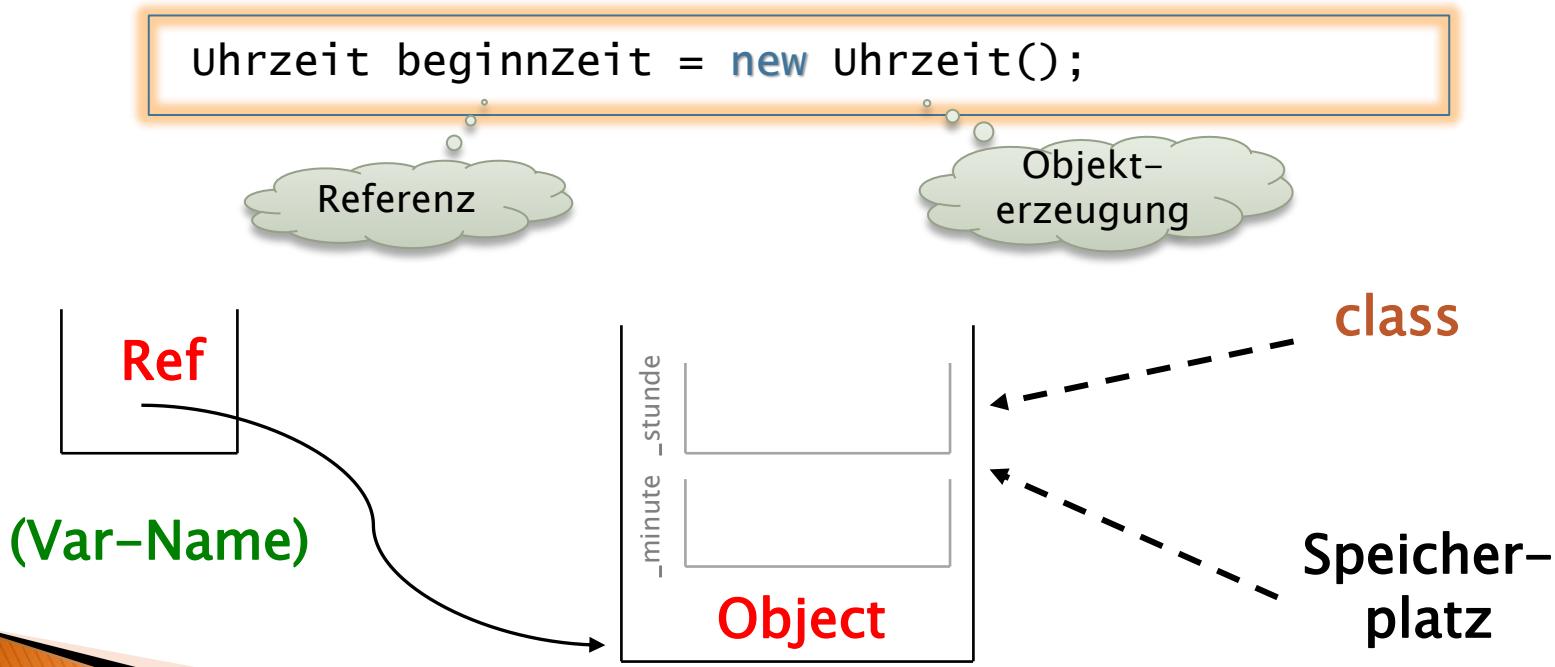
Attribute (oft "private"), legen fest welche Daten die Objekte haben

```
public class Uhrzeit {  
    // Attribute für die Stunde und die Minute  
    private int _stunde, _minute;  
    // eine Uhrzeit setzen  
    public void setzen(int stunde, int minute) {  
        _stunde = stunde;  
        _minute = minute;  
    }  
    // anzeigen  
    public void anzeigen() {  
        System.out.printf("%02d:%02d\n", _stunde, _minute);  
    }  
    // Minuten dazuzählen  
    public int zeitDazu(int minuten) { ... }  
}
```

Methoden (oft "public"), bestimmen welche Aktionen für die Objekte ausgeführt werden können

# Instanziierung von Objekten

- ▶ Erzeugung von Objekten (Instanziierung) erfolgt in Java ausschließlich **dynamisch**
- ▶ Weitere Verwendung immer über Referenz

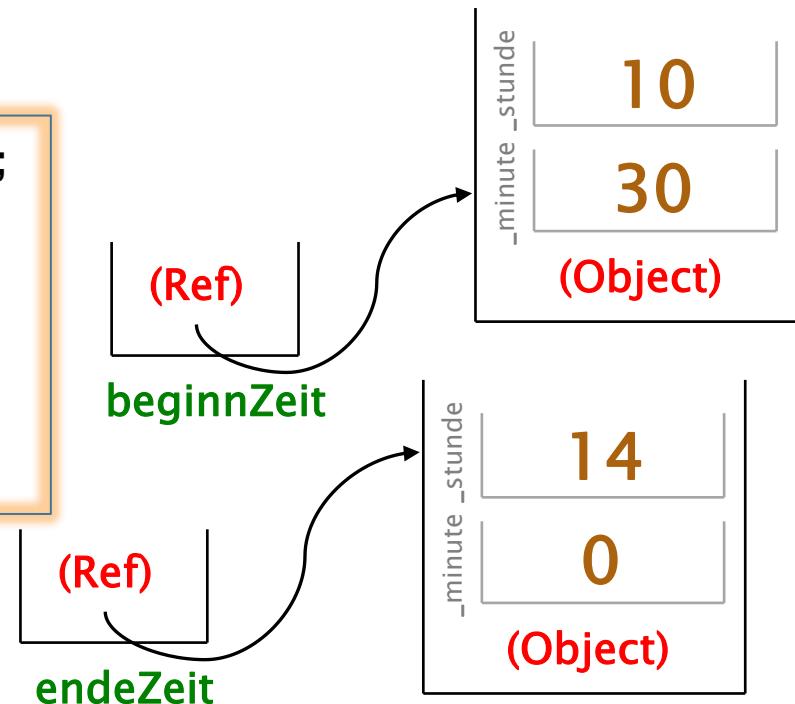


# Verwenden von Objekten

## ► Aufrufen der Methoden

- erfolgt immer über eine Referenz
- mit dem Operator für den Memberzugriff `.`

```
Uhrzeit beginnZeit = new Uhrzeit();
beginnZeit.setzen(10, 30);
beginnZeit.anzeigen();
Uhrzeit endezeit = new Uhrzeit();
endezeit.setzen(14, 0);
```



# Zugriffsattribute

Zugriffsart	Wer darf zugreifen
<b>private</b>	nur die Klasse selbst
<b>package</b>	alle im selben Package; <i>wird nicht angegeben</i>
<b>protected</b>	die Klasse, abgeleitete Klassen und alle im selben Package
<b>public</b>	jeder

# Konstruktor



- ▶ In Java muss jedes neue Objekt initialisiert werden
  - dazu dient ein Konstruktor
- ▶ Ein Konstruktor
  - ist eine Methode, die beim Erzeugen des Objekts **automatisch aufgerufen** wird
  - heißt so wie die Klasse
  - hat **keinen Rückgabetyp**
  - kann **überladen** werden

# Konstruktor

```
public class Uhrzeit {  
    // Attribute für die Stunde und die Minute  
    private int _stunde, _minute;  
    // Default-Konstruktor  
    public Uhrzeit() {  
    }  
    // Konstruktor mit Werten für Stunde und Minute  
    public Uhrzeit(int stunde, int minute) {  
        _stunde = stunde;  
        _minute = minute;  
    }  
}
```

uhrzeit ende = new Uhrzeit(14, 0);  
uhrzeit beginn = new Uhrzeit();  
beginn.setzen(10,30);

# Konstruktor

## ► Defaultkonstruktor

- wird **automatisch** erzeugt, wenn die Klasse keinen Konstruktor enthält
- hat **keine Parameter**
- enthält **keine Anweisungen**
- **initialisiert** alle Attribute
  - mit **0**, **null** oder **false**
  - bzw. mit den Werten der Feldinitialisierung

## ► Defaultkonstruktor wird nicht erzeugt

- wenn die Klasse **irgend einen Konstruktor** enthält

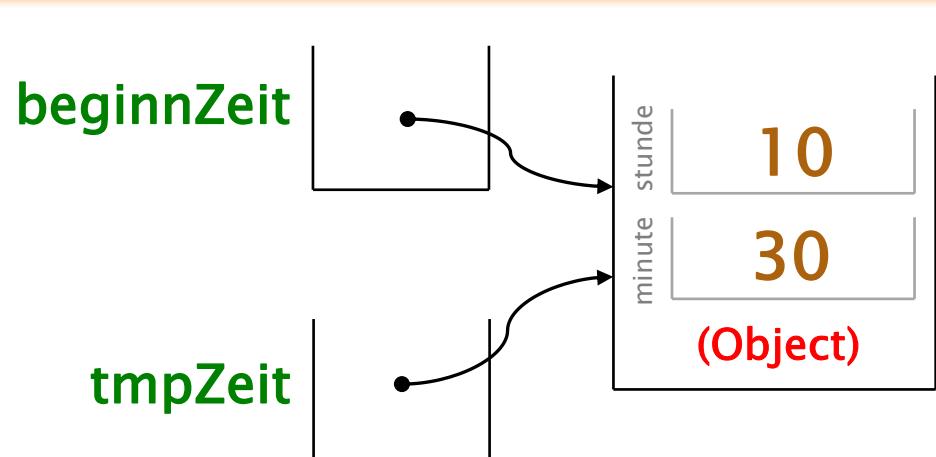
# Werttypen und Referenztypen

- ▶ Kopieren einer primitiven Variable
  - Kopiert den Wert
- ▶ Kopieren einer Variable einer Klasse
  - Kopiert die Referenz, nicht das Objekt

```
int zahl1, tmpzahl1;  
zahl1 = 10;  
tmpzahl1 = zahl1;
```



```
Uhrzeit beginnZeit, tmpZeit;  
beginnZeit = new Uhrzeit(10, 30);  
tmpZeit = beginnZeit;
```



# Instanz- vs. Statische Member

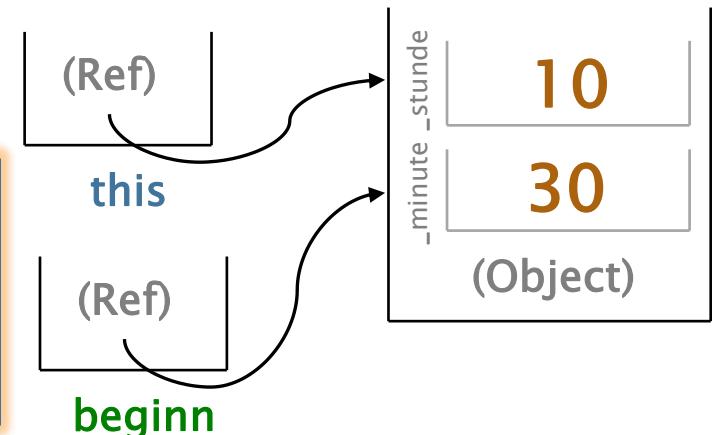
	Instanzmember	Statische Member
Zugehörigkeit	sind an ein Objekt gebunden	gehören direkt zur Klasse
Verwendung	es muss zuvor ein Objekt erzeugt worden sein	es ist kein Objekt erforderlich
Zugriff	nur über eine Referenz	ohne Referenz, über den Klassennamen
Kennzeichnung	keine (ist der Normalfall)	mit Keyword <b>static</b>
Einsatz	<ul style="list-style-type: none"><li>○ mehrere Exemplare der Klasse mit jeweils eigenen Werten;</li><li>○ fortgeschrittene OOP-Techniken nutzen (z.B. Vererbung)</li></ul>	<ul style="list-style-type: none"><li>○ Utility-Methoden, die ohne vorherige Instanziierung verwendbar sein sollen;</li><li>○ "Globale" Methoden</li></ul>

# Instanz-Member

- ▶ Instanz-Attribut
  - "Gewöhnliches" Attribut: existiert 1x pro Objekt
- ▶ Instanz-Methode
  - kann auch die Instanz-Attribute lesen und ändern
  - dazu wird beim Aufruf eine versteckte Referenz mitgegeben: **this**

```
Uhrzeit beginn = new Uhrzeit();
beginn.setzen(10, 30);
```

```
public void setzen(int stunde, int minute){
    this._stunde = stunde;
    this._minute = minute;
}
```



# Statische Member

- ▶ werden als **static** gekennzeichnet
- ▶ Statische Attribute
  - existieren genau 1x für die Klasse
  - existieren unabhängig von Instanzen der Klasse
- ▶ Statische Methoden
  - können diese Attribute und andere statische Methoden verwenden
  - können keine Instanzmember verwenden (**kein this**)
- ▶ Zugriff von außen
  - erfolgt über den Klassennamen

# Statische Member

Das Instanzattribut "id"  
existiert 1x pro Objekt

```
public class CountClass {  
    private static int objectCount;  
    private int id;  
    public CountClass(int id) {  
        this.id = id;  
        objectCount++;  
    }  
    public int getId() {  
        return id;  
    }  
    public static int getObjectCount(){  
        return objectCount;  
    }  
}
```

In static Methoden kann nur auf  
static Elemente zugegriffen werden

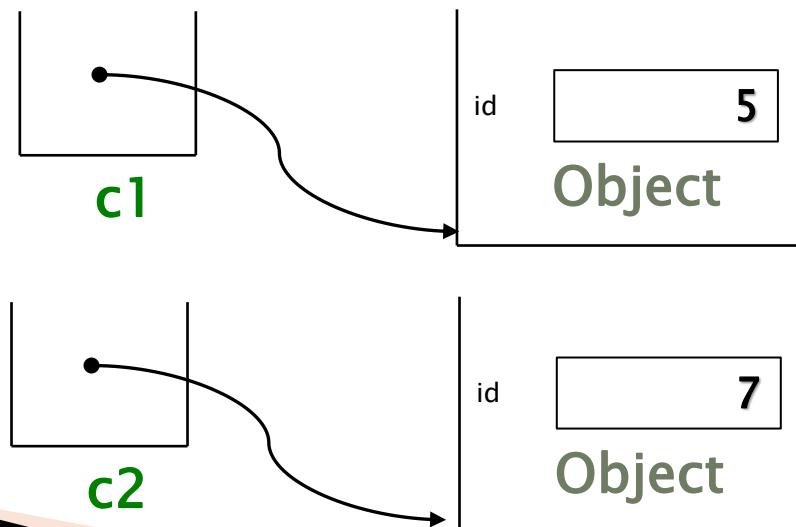
Das static Attribut "objectCount" gibt  
es nur einmal für die ganze Klasse!

```
CountClass c1, c2;  
c1 = new CountClass(5);  
c2 = new CountClass(7);  
int id1, id2, count;  
id1 = c1.getId(); // == 5  
id2 = c2.getId(); // == 7  
count = CountClass.  
    getObjectCount(); // == 2
```

Der Zugriff von außen erfolgt über  
den Klassennamen

# Instanz- vs. statische Member

```
public void test() {  
    CountClass c1 = new CountClass(5), c2 = new CountClass(7);  
    int id1 = c1.getId(), id2 = c2.getId();  
    int count = CountClass.getObjectCount();  
}
```



0 1 2

CountClass.  
objectCount

Speicherplatz für statisches Feld  
liegt außerhalb der Objekte

# Array

» Zusammenhängender  
Datenblock

# Array

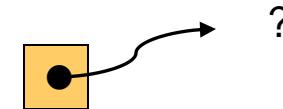
- ▶ Zusammenhängender Block von Werten desselben Typs
  - Anzahl der Elemente: **length**
  - Zugriff auf Elemente: per **Index** (beginnt bei 0)
- ▶ Synonyme: Vektor, Datenfeld, Tabelle

```
Random zufall = new Random(); // zufallszahlen-Generator
int[] zahlen = new int[3];    // Array für 3 Zahlen anlegen
// für jedes Element im Array
for (int i = 0; i < zahlen.length; i++) {
    // an diesem Index eine zufällige Zahl eintragen
    zahlen[i] = zufall.nextInt(100);
}
```

# Array - Vektor

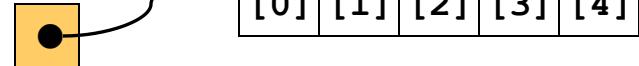
```
int[] zahlen;
```

Deklaration



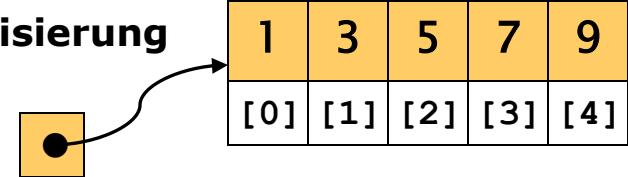
```
zahlen = new int[5];
```

Erzeugung



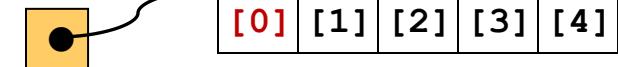
```
zahlen = new int[] { 1, 3, 5, 7, 9 };  
int[] zahlen = { 1, 3, 5, 7, 9 };
```

Initialisierung



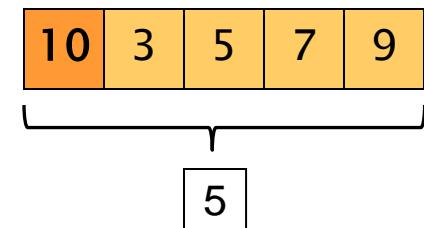
```
zahlen[0] = 10;
```

Zugriff per Index



```
int len = zahlen.length;
```

Anzahl der  
Elemente



# Wiederholung (for als for-each)

Ausdruck 1:  
Laufvariable

```
...
// passendes Array anlegen
int[] zahlen = new int[anzahl];
```

Ausdruck 2: Container  
mit mehreren Elementen,  
z.B. Array

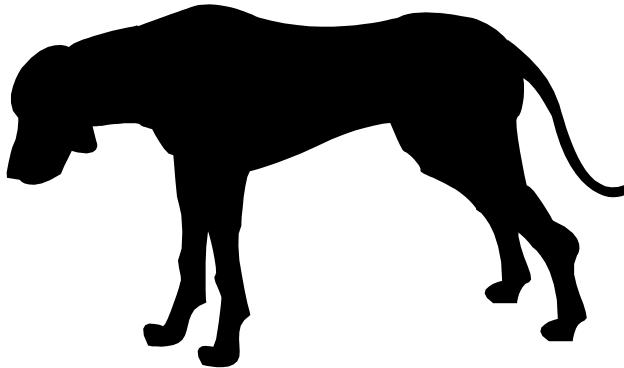
```
...
// alle zahlen ausgeben, mit for-each
for (int zahl : zahlen) {
    System.out.print(zahl + " ");
}
System.out.println();
```

# Vererbung

» Spezialisierung von Klassen

# Vererbung - Ableitung

- Die abgeleitete Klasse ist eine Spezialisierung einer (Basis-)Klasse



Hund

- Basisklasse
  - Grundattribute und -methoden von Hunden

Dackel ist abgeleitet  
von Hund



Dackel

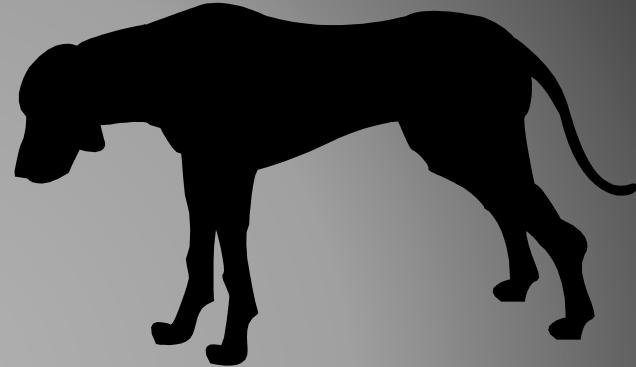
- Wie Hund, aber:
  - einige Dinge anders
  - zusätzliche Funktionalität

# Vererbung - Ableitung

public class Hund



public class Dackel extends Hund



Dackel wird von  
Hund abgeleitet

# Vererbung

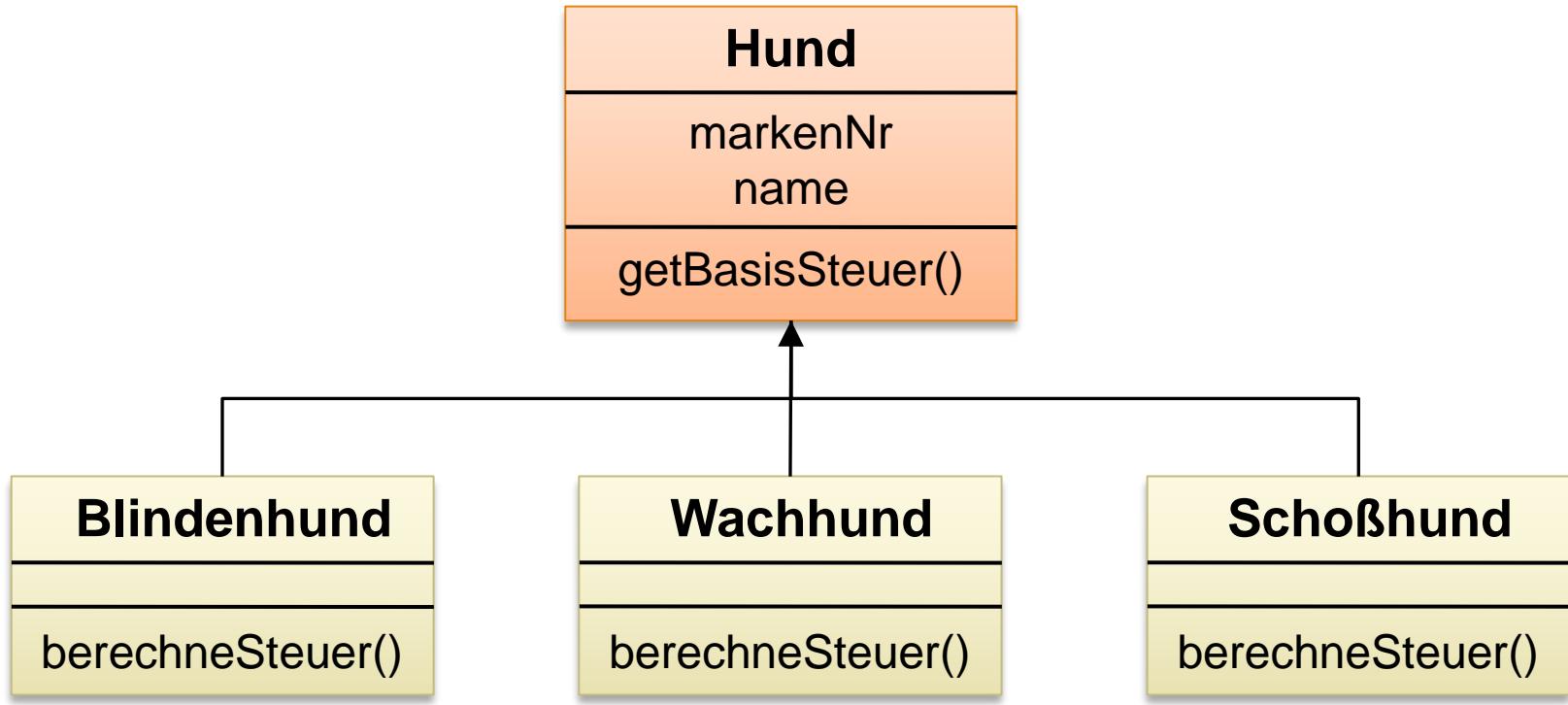
- ▶ Beispiel "Hundesteuer berechnen"

*Die Hundesteuer beträgt 100 EUR.  
Wachhunde zahlen nur die Hälfte.  
Und Blindenhunde sind überhaupt  
von der Steuer befreit.*

Berechnet werden soll die Steuer für  
Blindenhunde, Wachhunde und Schoßhunde

# Vererbung

- ▶ Beispiel "Hundesteuer berechnen"



Blindenhund, Wachhund und Schoßhund sind Spezialisierungen von Hund, mit jeweils eigener Berechnungsmethode

# Vererbung

```
class Hund {  
  
    private String _name, _markenNr;  
  
    // Hundesteuer berechnen  
    public int getBasisSteuer() {  
        // "Normale" Steuer  
        return 100;  
    }  
    public String getName(){  
        return _name;  
    }  
    public String getMarkenNr(){  
        return _markenNr;  
    }  
}
```

Basisklasse

```
class Wachhund extends Hund {  
  
    private String _einsatzort;  
  
    // Hundesteuer für Wachhunde  
    public int berechneSteuer() {  
        // zahlen nur die Hälfte  
        return  
            super.getBasisSteuer()/2;  
    }  
    public String getEinsatzort(){  
        return _einsatzort;  
    }  
}
```

abgeleitete  
Klasse

# Vererbung - Konstruktor

- ▶ Beim Erzeugen eines Objekts einer abgeleiteten Kasse wird immer
  - zuerst der **Konstruktor der Basisklasse** aufgerufen,
  - dann der **Konstruktor der abgeleiteten Kasse**

```
class Hund{  
    ...  
}
```

```
wachhund rex = new Wachhund();
```

```
class Wachhund extends Hund {  
    ...  
}
```



1. Konstruktor von Hund
2. Konstruktor von Wachhund

# Vererbung: super-Aufruf

- Falls der Basisklassen-Konstruktor Parameter hat
  - werden die Parameter über einen **expliziten super-Aufruf** übergeben
  - Dieser Aufruf muss als **1. Anweisung** in einem Konstruktor stehen

```
class Hund {  
    public Hund(String name) {  
        ...  
    }  
}
```

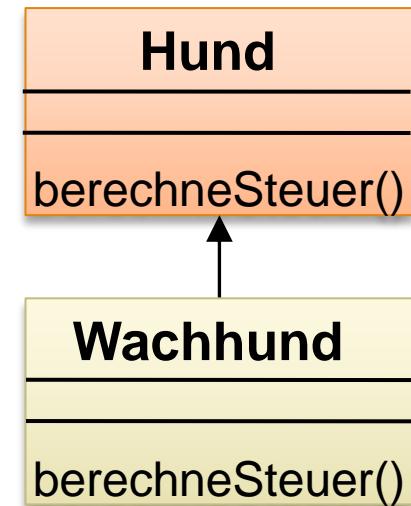
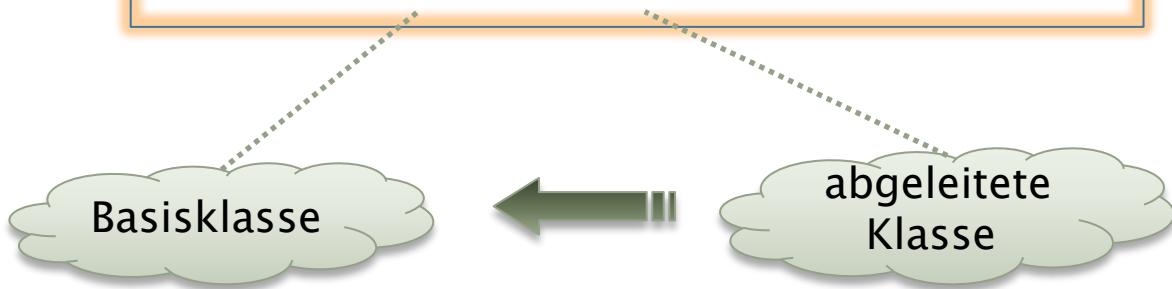
```
class Wachhund extends Hund {  
    public Wachhund(String n) {  
        super(n);  
        ...  
    }  
}
```

```
wachhund rex =  
new wachhund ("Rex");
```

# Vererbung - Typkompatibilität

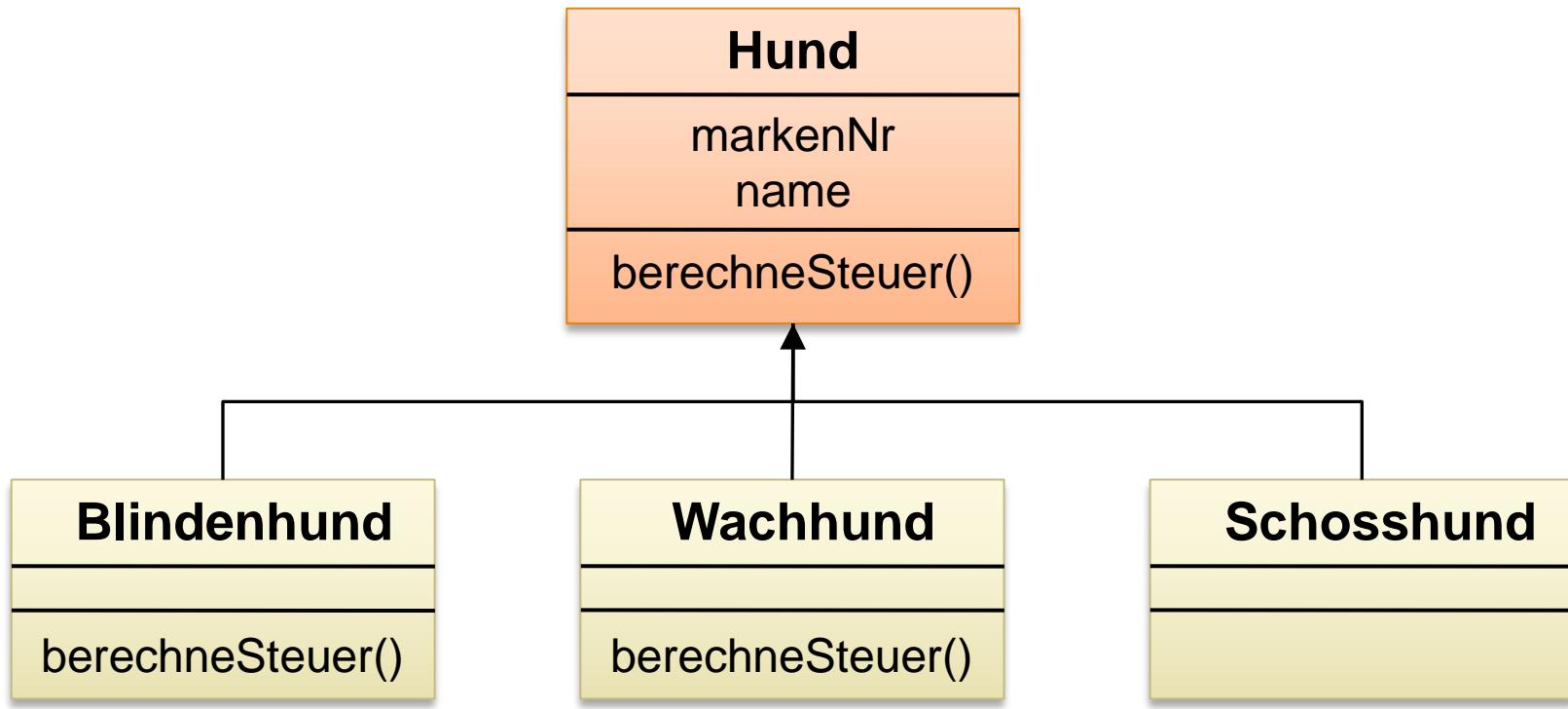
- ▶ Referenz der Basisklasse
  - kann auf Objekt einer abgeleiteten Klasse verweisen
- ▶ umgekehrte Zuweisung
  - ist implizit nicht möglich

```
wachhund rex = new Wachhund ("Rex");  
Hund einHund = rex;
```



# Vererbung – Polymorphismus

- ▶ Beispiel "Hundesteuer berechnen (2)"



Blindenhund, Wachhund und Schoßhund sind Spezialisierungen von Hund, falls erforderlich mit eigener Berechnungsmethode

# Vererbung – Polymorphismus

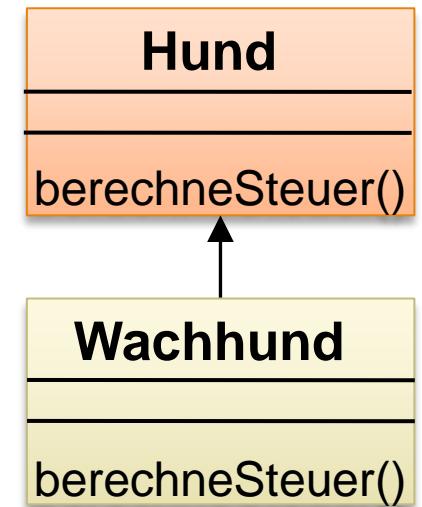
```
wachhund rex = new Wachhund ("Rex");  
Hund einHund = rex;  
int steuer = einHund.berechneSteuer();
```

Referenz  
auf  
Hund



```
int berechneSteuer()  
{  
    return 100/2;  
}
```

```
int berechneSteuer()  
{  
    return 100;  
}
```



Welche Methode  
wird aufgerufen?

# Vererbung – Polymorphismus

## ► Polymorphe Methode

- Methode der abgeleiteten Klasse **überschreibt** eine Basisklassen-Methode mit der gleichen Signatur
- Zur Laufzeit wird die **zum aktuellen Objekt** gehörende Methode aufgerufen (=dynamisches Binden, "**late binding**")
- Das Objekt kann in **mehreren Gestalten** auftreten (polymorph)

# Vererbung – Polymorphismus

- ▶ in Java sind Instanzmethoden automatisch polymorph
  - keine spezielle Kennzeichnung erforderlich
  - abgeleitete Klassen überschreiben die Methode, indem sie eine Methode mit derselben Signatur definieren
  - Implementierung der Basisklasse
    - kann über **super** aufgerufen werden
- ▶ Nicht polymorph sind
  - private Methoden
  - static Methoden
  - mit final gekennzeichnete Methoden

# Vererbung – Polymorphismus

## ► Die Annotation **@Override**

- kennzeichnet eine Methode als Überschreibung
- schützt vor Fehlern durch nicht übereinstimmende Signaturen

```
class Hund {  
    public int berechneSteuer() {  
        return 100;  
    }  
}
```

```
class Wachhund {  
    @Override  
    public int berechneSteuer() {  
        return super.berechneSteuer() / 2;  
    }  
}
```