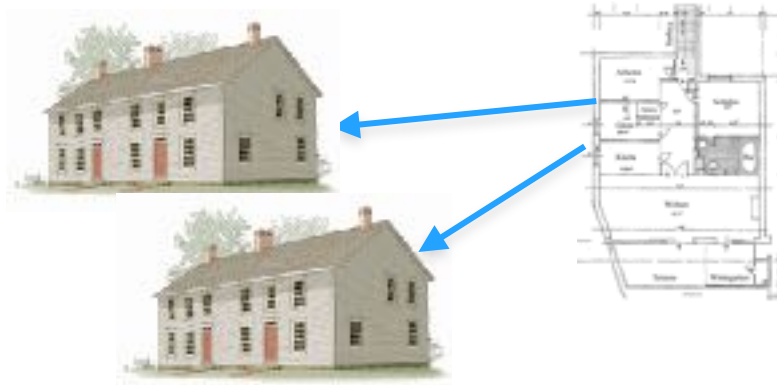


Objektorientiertes Programmieren



SoSe 2018

Oliver Wiese

statisch

Was ist eine Klasse?

Eine Klasse ist ein Bauplan, um Objekte einer bestimmten Sorte zu erzeugen.

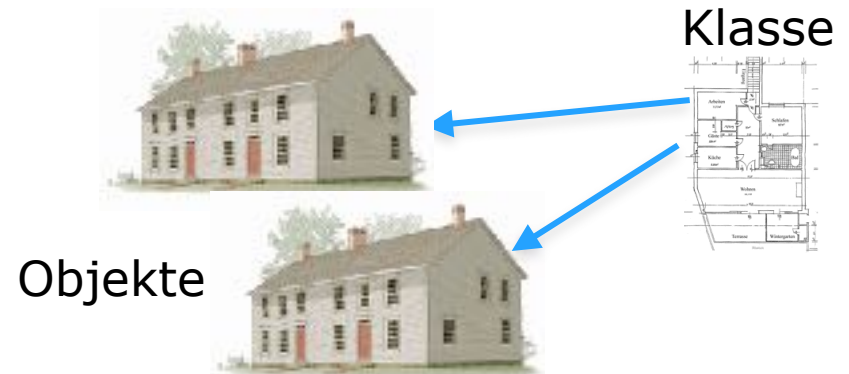


dynamisch

Was ist ein Objekt ?

Ein Objekt ist ein Softwarebündel aus **Variablen** und mit diesen Variablen zusammenhängenden **Methoden**.

Ein Objekt ist eine konkrete Ausprägung bzw. eine Instanz einer Klasse.



Beispiel:

Die Haus-Klasse

Eigenschaften:

Etagen
Wohnfläche
Nutzfläche
Adresse

Operationen:

sanieren
renovieren
verkaufen

Klassendefinition

Ein konkretes Haus Objekt

Zustand:

zwei Etagen
100 m² Wohnfläche
200 m² Nutzfläche
Takustr. 20

Operationen:

kann saniert werden
kann renoviert werden
kann verkauft werden

Klasse-Definition

Attribute:

- *Eigenschaft₁*
- *Eigenschaft₂*
- • • •

Verhalten:

- *Methode₁*
- *Methode₂*
- *Methode₃*
- • • •

Beispiel: Katze-Klasse

Attribute :

- Name
- Besitzer
- Farbe
- hungrig

• • • • •

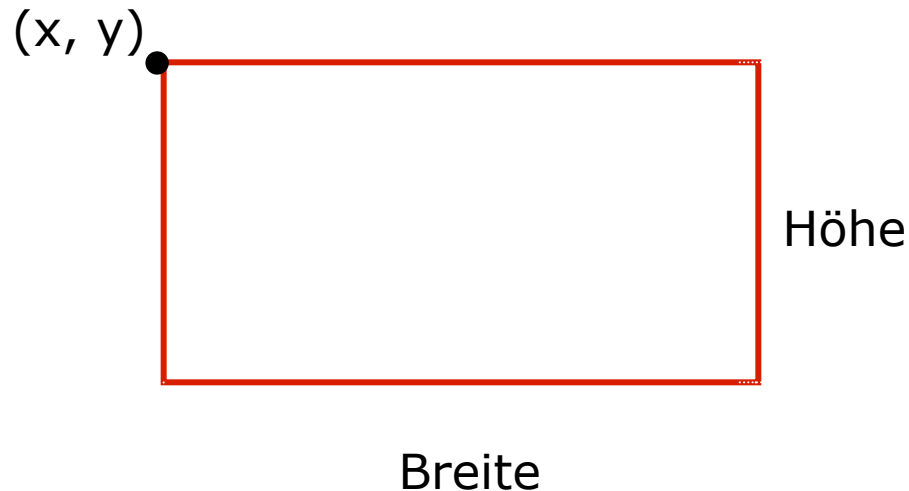
Verhalten:

- isst
- läuft
- kratzt
- schläft

• • •

Modellierung von Rechteck-Objekten

Eigenschaften



Operationen

- berechne Fläche
- berechne Umfang
- verkleinern
- vergrößern
- verschieben
- klonen

Modellierung von Rechteck-Objekten

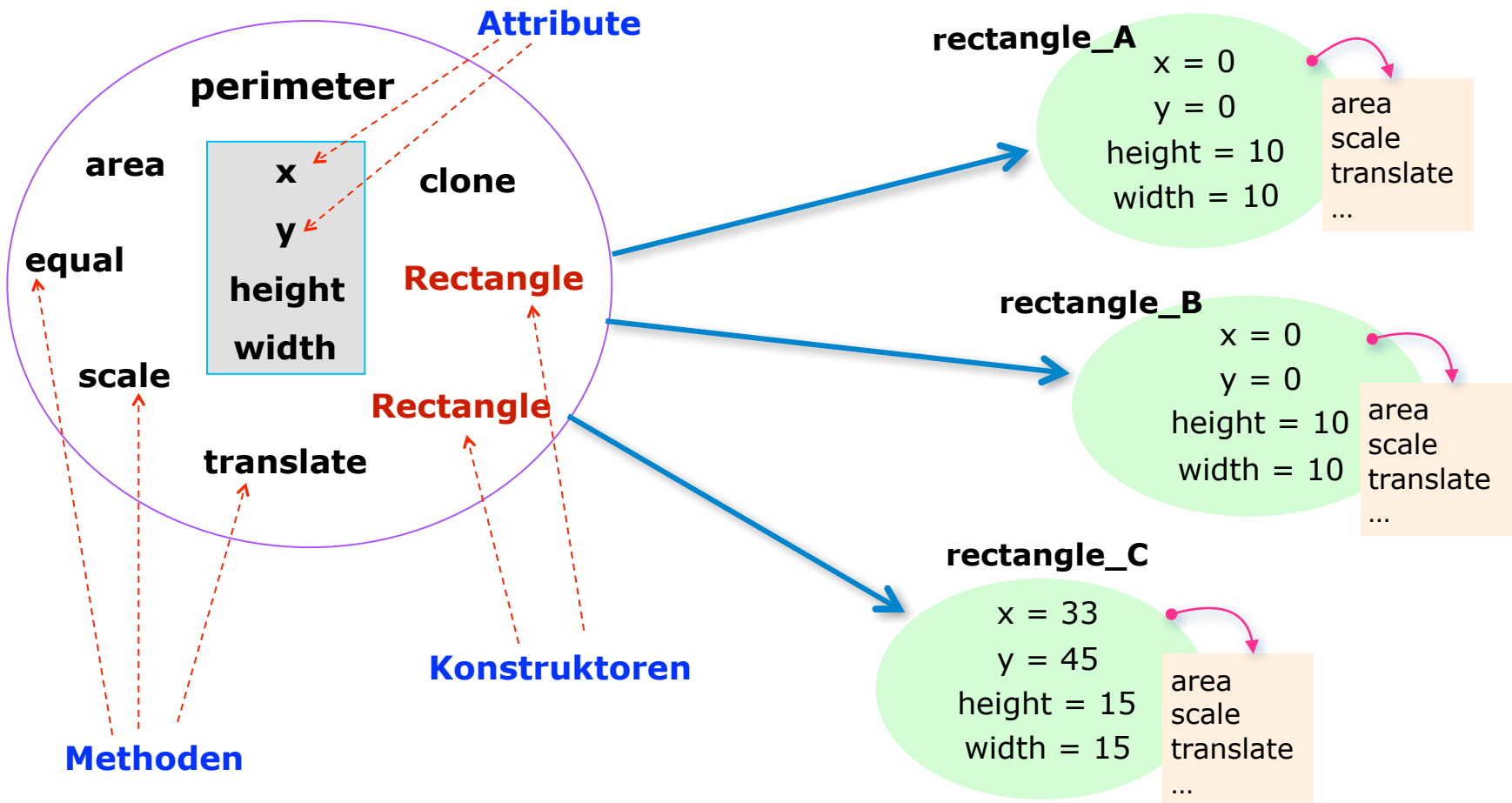
```
public class Rectangle {  
    // Eigenschaften  
    . . .  
    . . .  
    // Konstruktoren  
    . . .  
    . . .  
    // Operationen  
    . . .  
    . . .  
    . . .  
}
```

Dateiname:

Rectangle.java

Rectangle-Klasse

Rectangle-Objekte



Klasse Rectangle

in Java

```
public class TestRectangle{  
  
    // main-Methode  
    public static void main( String[] args ) {  
        Rectangle r1 = new Rectangle();  
        Rectangle r2 = new Rectangle();  
        int u = r1.area();  
        int f = r2.perimeter();  
    }  
} // end of class TestRectangle
```

```
public class Rectangle {  
    // Attribute  
    int x;  
    int y;  
    int width;  
    int height;  
    // Konstruktoren  
    public Rectangle() {  
        x = 0;  
        y = 0;  
        width = 10;  
        height = 10;  
    }  
    // Methoden  
    public int perimeter() {  
        return 2*(width + height);  
    }  
    public int area() {  
        return (width * height);  
    }  
} // end of class Rectangle
```

Klasse Rectangle in Python

Anwendung der Klasse:

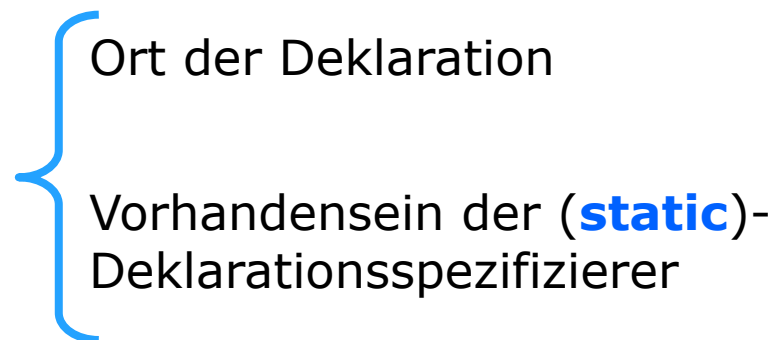
```
r1 = Rectangle()  
print(r1.area())  
print(r1.perimeter())  
  
r1.height = 20  
print(r1.area())
```

```
class Rectangle:  
    # Konstruktor  
    def __init__(self):  
        self.x = 0  
        self.y = 0  
        self.width = 10  
        self.height = 10  
  
    # Methoden  
    def perimeter(self):  
        return 2*(self.width + self.height)  
  
    def area(self):  
        return self.width * self.height
```

Variablen in Java



Diese Klassifikation richtet sich nach folgenden zwei Aspekten:



Variablen in Java

Lokale Variablen

Hilfsvariable für Berechnungen

Sie werden innerhalb von Methoden deklariert.

Lebenszeit: nur solange die Methode ausgeführt wird.

Instanzvariablen

(Feldvariablen,
Attribute)

Variablen, in denen die Eigenschaften
von Objekten gespeichert werden

Lebenszeit: nur solange das Objekt existiert.

Klassenvariablen

Variablen, die zu einer Klasse gehören

Lebenszeit: solange das Programm ausgeführt wird.

Nameskonventionen

Variablennamen beginnen mit Kleinbuchstaben
myName

Klassennamen beginnen mit Großbuchstaben
Rechteck

Konstante
Klassenvariablen nur Großbuchstaben
BLAU

Methoden beginnen mit Kleinbuchstaben
setColor (BLAU)

Instanzvariablen

Die Klasse `Kreis` vereinbart drei *Instanzvariablen* mit jeweils einem *Typ*, einem *Namen* und einem *Wert*.

```
...  
Kreis k = new Kreis();  
k.x = 0;  
k.y = 0;  
...
```

Zugriff nach dem Muster `<Referenz> . <Feldname>`

Instanzvariablen werden beim Erzeugen des Objekts entweder mit dem im Konstruktor angegebenen Wert initialisiert oder mit einem Standardwert:

Klassenvariablen

Variablen haben den Deklarationsspezifizierer **static** oder **final**

static Variablen sind klassenbezogen

..d.h. speichern Eigenschaften, die für eine ganze Klasse gültig sind, und von denen nur ein Exemplar für alle Objekte der Klasse existiert; ihre Lebensdauer erstreckt sich über das ganze Programm.

final-Variablen

der Wert darf nur einmal zugewiesen werden

Beispiel:

```
public class Kreis {
    // Instanzvariablen
    float x;
    float y;
    float radio;

    // Klassenvariable
    public static final float PI = 3.141598f;

    // Methoden
    public float area() {
        // Lokale Variable
        float a;
        a = PI*radio*radio;
        return a;
    } ...
}
```



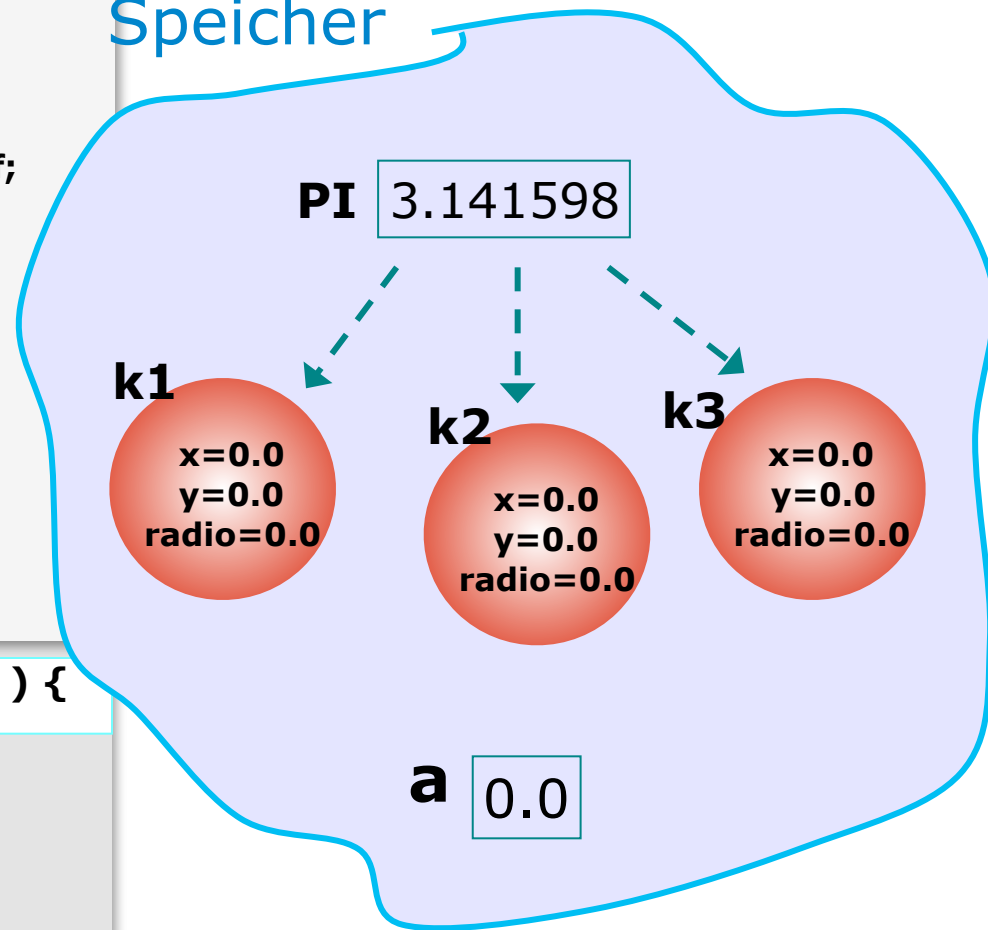
```
public class Kreis {
    // Instanzvariablen
    float x;
    float y;
    float radio;

    // Klassenvariable
    public static final float PI = 3.141598f;

    // Methoden
    public float area() {
        // Lokale Variable
        float a;
        a = PI*radio*radio;
        return a;
    }
}
```

```
public static void main ( String[] args ) {
    Kreis k1 = new Kreis();
    Kreis k2 = new Kreis();
    Kreis k3 = new Kreis();
    float flaeche = k1.area();
}
```

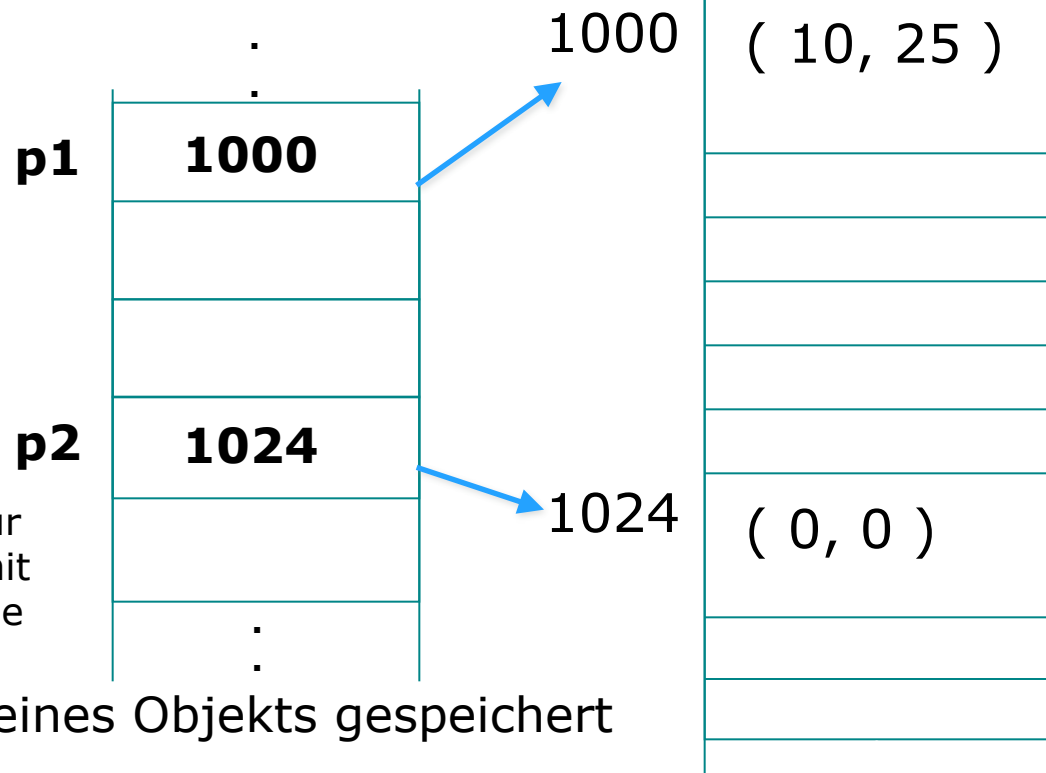
Speicher



Was ist eine Referenz?

```
Point p1;  
Point p2;  
p1 = new Point ( 10, 25 );  
p2 = new Point ( 0, 0 );
```

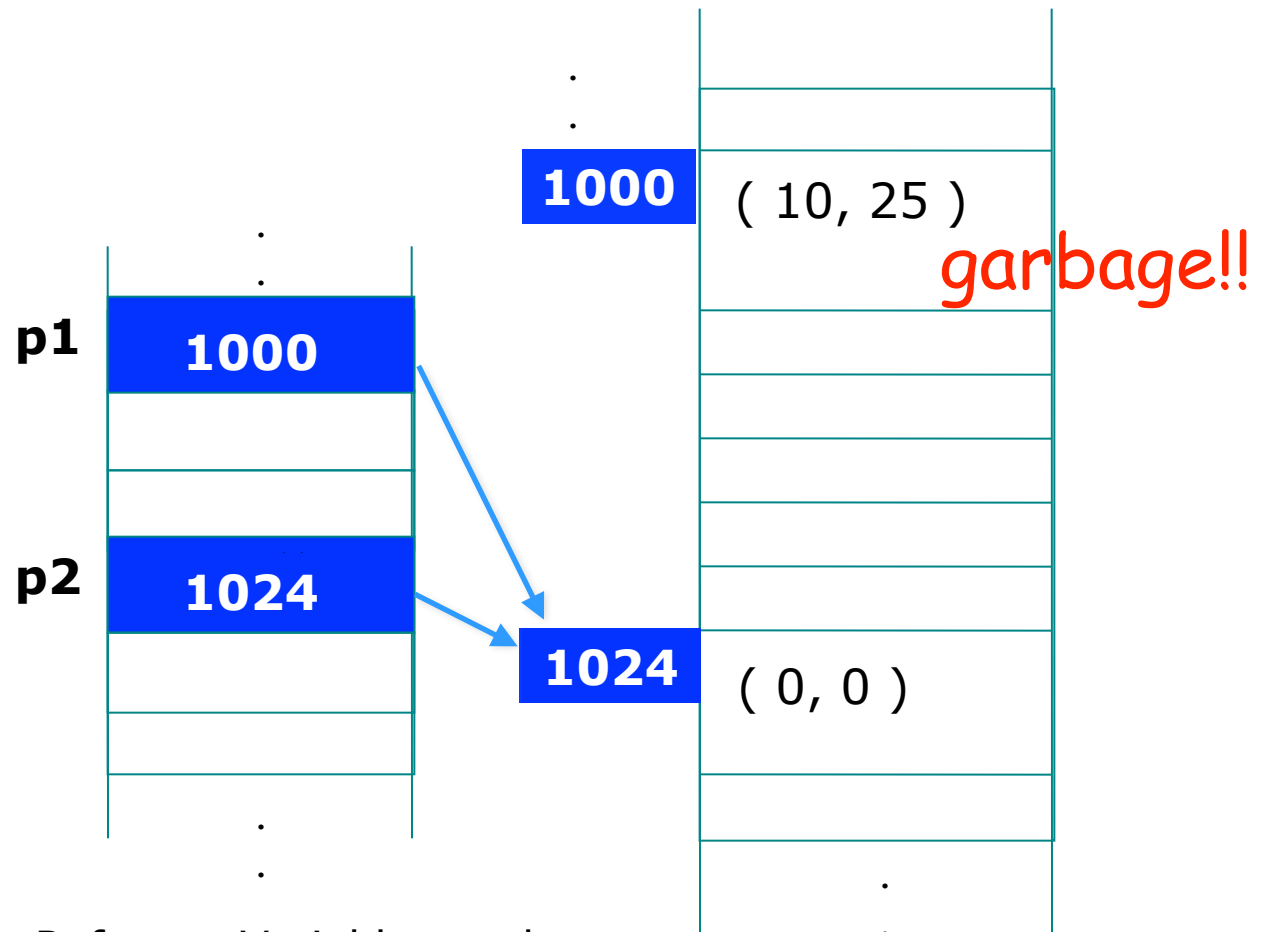
Speicher für
Variablen mit
fester Größe



sind Variablen, wo die Adresse eines Objekts gespeichert wird

Referenz-Variablen

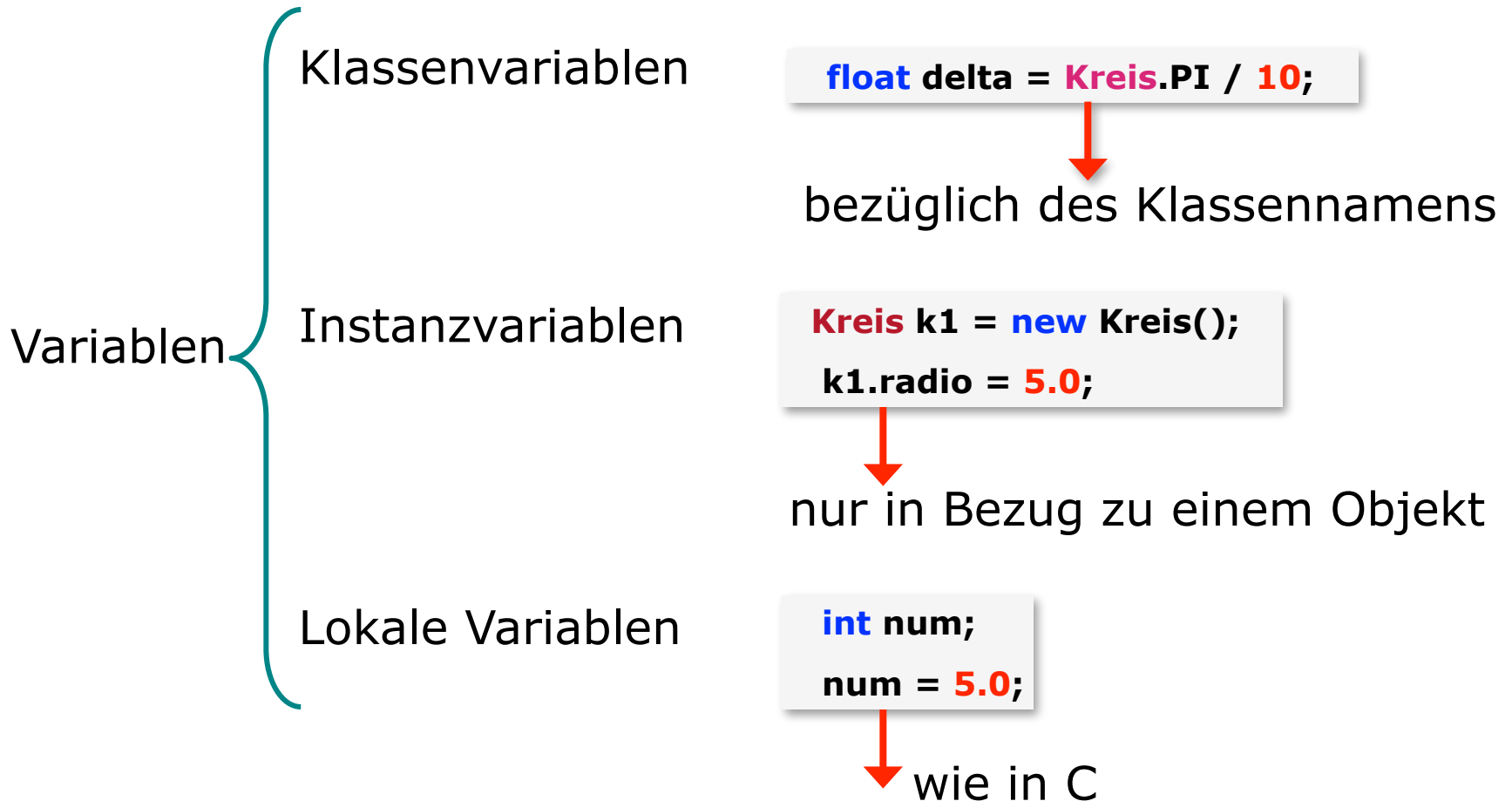
`p1 = p2 ;`



In Java ist Arithmetik mit Referenz-Variablen verboten.
Nur die Operatoren `=`, `==`, `!=` sind erlaubt.

Variablen in Java

Zugriff



Sichtbarkeit von Java-Variablen

Zugriffsangabe
oder
Sichtbarkeit

	Klasse	Unterklassen	Paket	Welt
privat				
package				
protected				
public				

Kein Modifikator ist äquivalent zur **package**

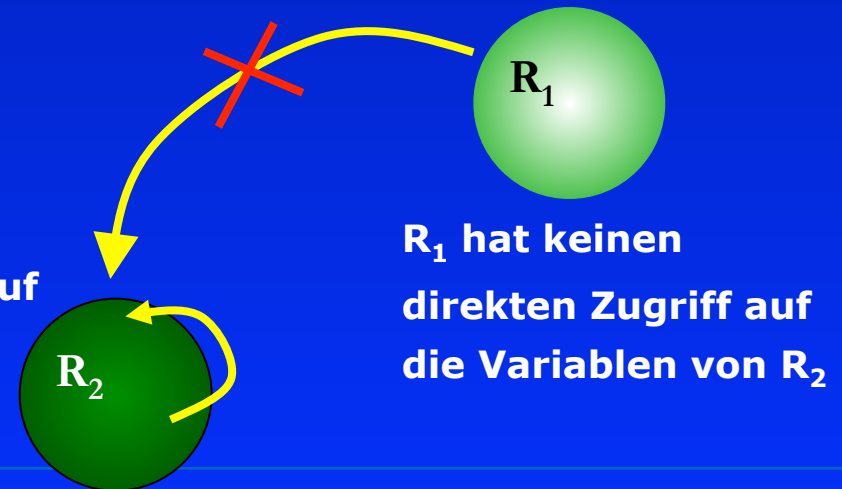
Sichtbarkeit von **private**-Variablen

Rechteck-Klasse

```
public class Rectangle {  
  
    private int x;  
    private int y;  
    private int width;  
    private int height;  
  
    /* Methoden */  
  
    public int area() {  
        return width*height;  
    }  
}
```

Zugriff nur innerhalb
der Rechteck-Klasse

R₂ hat Zugriff auf
seine eigenen
Variablen.




Zugriffsangabe oder Sichtbarkeit von Variablen

Klasse

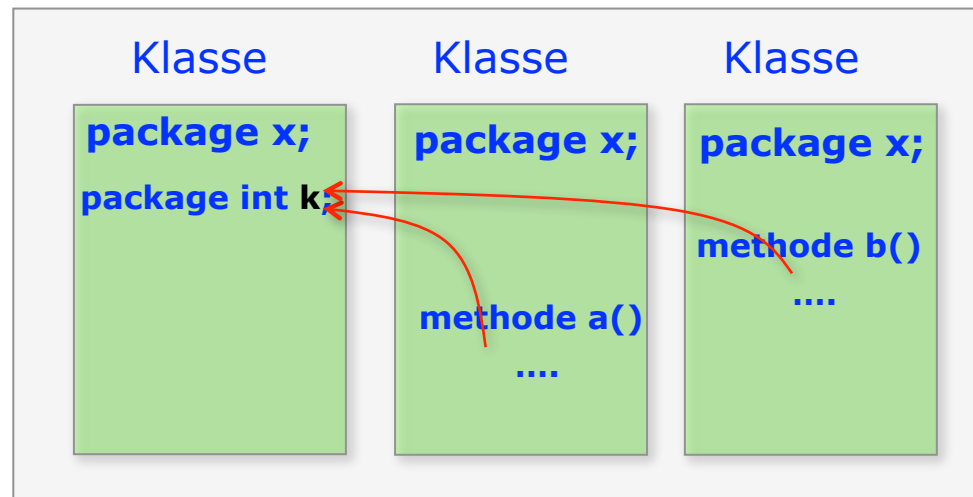
```
privat int k;
...

methode a(){
    k = 10;
}
...
```



Nur das Objekt selbst kann den Inhalt einer privaten Variablen mittels seiner Methoden modifizieren.

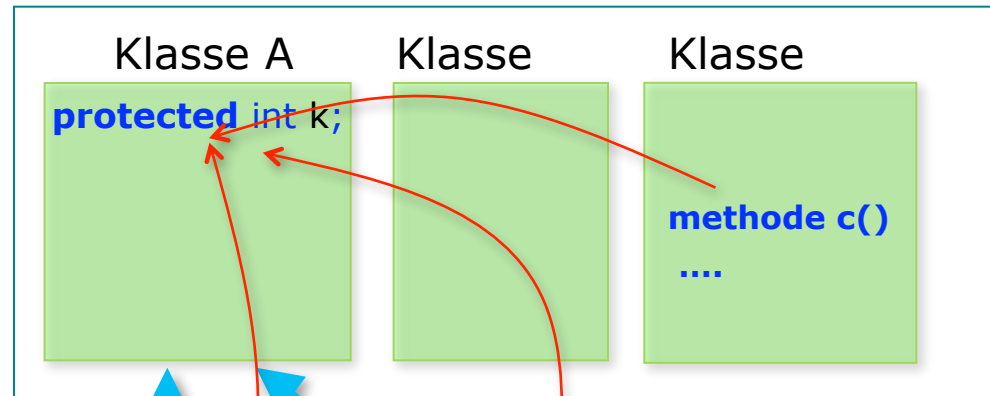
Paket- oder Verzeichnis-x



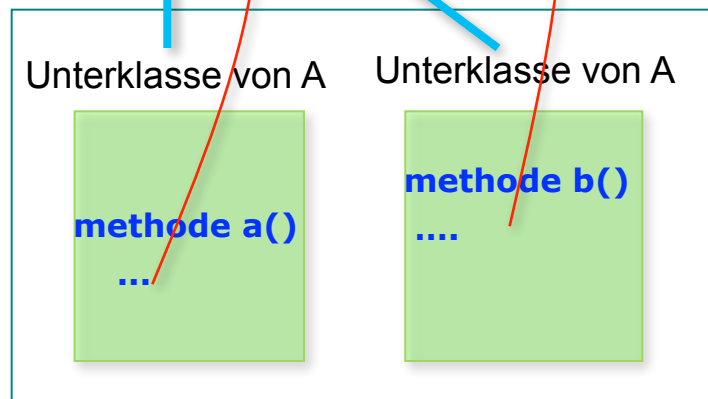
Alle Objekte innerhalb eines Verzeichnisses haben direkten Zugriff auf eine **package**-Variable.

Zugriffsangabe oder Sichtbarkeit von Variablen

Paket oder Verzeichnis



Paket oder Verzeichnis



Zugriff innerhalb der
Klassen-Hierarchie

Zugriffsangabe oder Sichtbarkeit von Variablen

Paket oder Verzeichnis

Klasse

public

Klasse

Klasse

Paket oder Verzeichnis

Klasse

Klasse

Paket oder Verzeichnis

Klasse

Klasse

Paket oder Verzeichnis

Klasse

Klasse

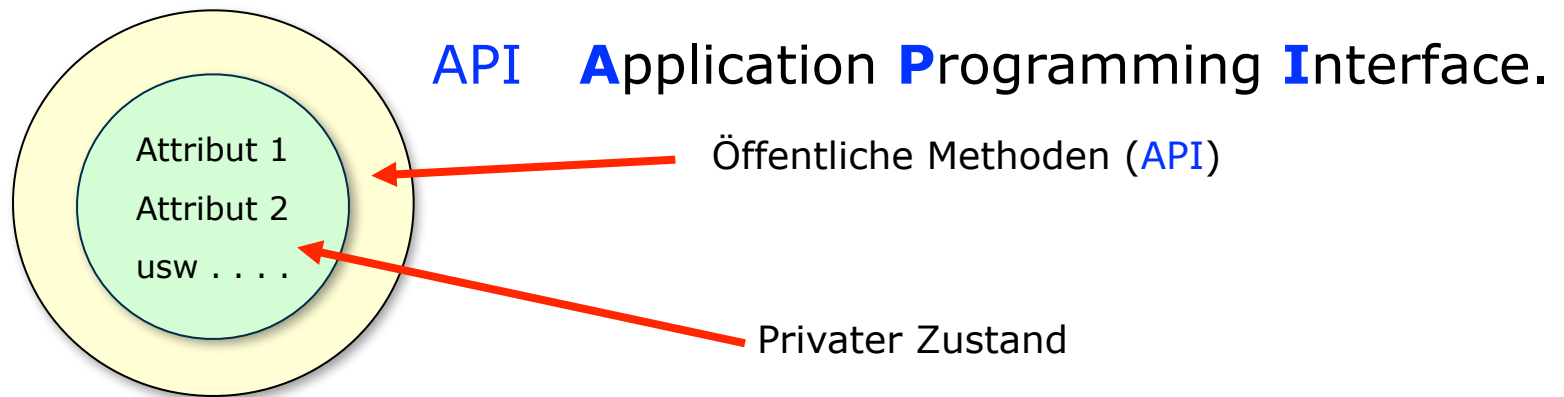
Klasse

Die Welt aller Java-Klassen

Was ist Kapselung ?

Kapselung ist die Einschränkung des Zugriffs auf die Instanzvariablen eines Objektes durch Objekte anderer Klassen.

Man spricht von einer **Kapselung** des Objektzustands.



Kapselung schützt so den Objektzustand vor “unsachgemäßer” Änderung und unterstützt **Datenabstraktion**.

Kapselung erfolgt durch die **Zugriffsmodifizierer** (**public**, **private**, **protected** und **package**)

Beispiel:

Definition der setTime-Methode

```
. . .  
public void setTime( int h, int m, int s ) {  
    if ( (s>59) || (s<0) || (m>59) || (m<0) || (h>23) || (h<0) ){  
        System.out.println("Falsche Zeitangabe:"+h+":"+m+":"+s);  
    } else {  
        hours = h;    minutes = m;    seconds = s;  
    }  
}  
. . .
```

this

Das Schlüsselwort **this** bezeichnet immer eine Referenz auf das aktuelle Objekt selbst.

this kann in Methoden und in Konstruktoren verwendet werden, um durch Argumentnamen "verschattete" Variablennamen zu erreichen:

Definition der setTime-Methode

```
...  
public void setTime( int hours, int minutes, int seconds ) {  
    if ( (seconds>59) || (seconds<0) || (minutes>59)  
        || (minutes<0) || (hours>23) || (hours<0) ){  
        System.out.println("Falsche Zeit:" +hours+":"+minutes+":"+seconds);  
    } else {  
        this.hours = hours;  
        this.minutes = minutes;  
        this.seconds = seconds;  
    }  
}  
...
```

this

Referenz auf
das aktuelle
Objekt selbst

```
public class Kreis {

    public final double PI = 3.141598;
    public double x;
    public double y;
    private double radio;

    public Kreis( double x, double y ){
        ▶ this.x = x;
        ▶ this.y = y;
    }

    public double getRadio() {
        return this.radio;
    }

    public void setRadio( double radio ) {
        if ( r>0 )
            ▶ this.radio = radio;
        else
            System.err.println( "Fehler:...." );
    }

    ...
}
```

Test-Beispiel für die Kreis-Klasse

```
public class TestKreis {

    public static void main(String[] args) {
        Kreis k = new Kreis();
        k.x = 1.0;
        k.y = 5.0;
        k.setRadio( 30 );
        k.setRadio( -6 );
        System.out.println( k.flaeche() );
        System.out.println( k.umfang() );
        double radio = k.getRadio();
        System.out.println(radio);
        System.out.println( k.getRadio() );
    }
}
```

Fehler:....

2827.4382

188.49588

30.0

30.0

null

Das Schlüsselwort **null** bezeichnet immer ungültige, d.h. nicht initialisierte Referenzen.

null kann überall da verwendet werden, wo eine Referenz erwartet wird. Zugriff auf eine Referenz, die gleich null ist, erzeugt einen Laufzeitfehler.

(**NullPointerException**)

```
Rechteck r1;  
Rechteck r2 = null;  
  
...  
r1.gleich( r2 );
```

→ **Verursacht einen Laufzeitfehler!**

Vordefinierte Operationen, die mit Referenz-Variablen erlaubt sind

(Typ)_Operator

```
Object objekt = null;
```

```
Button button = (Button) objekt;
```

Der (.) Operator

Mit dem (.)-Operator hat man Zugriff auf die Eigenschaften und Methoden eines Objekts.

```
Punkt p1 = new Punkt ( 10, 35 );
```

```
int x_koord = p1.x ;
```

Klassendefinition

```
public class Kreis {  
    Klassenvariable → public static final double PI = 3.141598;  
    Instanzvariablen → {  
        public double x;  
        public double y;  
        private double radio;  
    }  
    Konstruktor → {  
        public Kreis() {  
            x = 0.0;  
            y = 0.0;  
            radio = 1.0;  
        }  
    }  
    get-Methode → {  
        public double getRadio() {  
            return radio;  
        }  
    }  
    set-Methode → {  
        public void setRadio( double r ) {  
            if ( r>0 ) radio = r;  
            else      System.err.println( "Fehler:...." );  
        }  
    }  
    Methode → {  
        public double flaeche(){  
            return PI*radio*radio;  
        }  
    }  
    Methode → {  
        public double umfang() {  
            return PI*2*radio;  
        }  
    }  
} // Ende der Kreis-Klasse
```

Objekterzeugung

Objekte werden durch den Aufruf von Konstruktoren erzeugt.
Ein **Konstruktor** wird mit Hilfe der **new**-Operatoren aufgerufen.

```
Kreis k1 = new Kreis();
```

Eine Klassendefinition kann mehrere Konstruktoren haben mit verschiedenen Initialisierungen der Objekteigenschaften.

Wenn in einer Klasse keine Konstruktoren definiert worden sind, werden die Eigenschaften von Objekten mit Defaultwerten initialisiert.

Objekterzeugung

...

```
Kreis first_circle = new Kreis ( 0.0, 0.0, 1.0 );
```

```
Kreis second_circle = new Kreis ( first_circle );
```

```
Kreis four_circle = new Kreis ( 5.0 );
```

```
Kreis third_circle = new Kreis ( );
```

...

Konstruktoren

Ein guter OOP-Stil bedeutet, geeignete *Konstruktoren* zu definieren, die Objekte initialisieren und evtl. initiale Berechnungen durchführen.

```
public class Beverage {
    String name;
    int price,
    int stock;

    // Konstruktor
    Beverage( String name, int price, int stock ) {
        this.name    = name;
        this.price    = price;
        this.stock    = stock;
    }
    . . .
}
```

Ist kein Konstruktor definiert, wird ein **impliziter** Konstruktor ohne Argumente angenommen.

```
public class Kreis {  
    double x, y, radio;  
    ...  
}
```

=

```
public class Kreis {  
    double x, y, radio;  
    public Kreis(){  
    }  
    ...  
}
```

Sobald ein expliziter Konstruktor definiert ist, fällt der implizite Konstruktor weg!

Konstruktoren

"gute Regel" bei mehreren Konstruktoren:

Schreibe *genau einen* Konstruktor, der alle Initialisierungen vornimmt und rufe ihn aus den anderen mit geeigneten Parametern auf. Dies vermindert die Zahl potentieller Fehler.

```
public class Kreis {
    double x, y, radio;
    public Kreis ( double x, double y, double radio ) {
        this.x = x;  this.y = y;  this.radio = radio;
    }
    public Kreis ( double r ) { this ( 0.0, 0.0, r ); }
    public Kreis ( Kreis c ) { this ( c.x, c.y, c.radio ); }
    public Kreis ()      { this ( 1.0 ); }
}
```

Instanzmethoden

Instanzmethoden definieren das Verhalten von Objekten. Sie werden innerhalb einer Klassendefinition angelegt und haben Zugriff auf alle Variablen des Objekts.

Sie haben immer den impliziten Parameter **this**

```
public class Person {
    private String name = "";
    ...
    String getName() {
        return this.name;
    }
    void setName( String name ) {
        this.name = name;
    }
}
```


Zugriffskontrolle auf Methoden

Der Zugriff auf Methoden kann genau so wie im Variablen durch **Modifizierer** gesteuert werden:

- **public**: überall zugänglich.
- **private**: nur innerhalb der eigenen Klasse zugänglich.
- **protected**: in anderen Klassen des selben Packages und in Unterklassen zugänglich.
- **kein Modifizierer**: sind nur für Code im selben **Paket** (package) zugänglich.

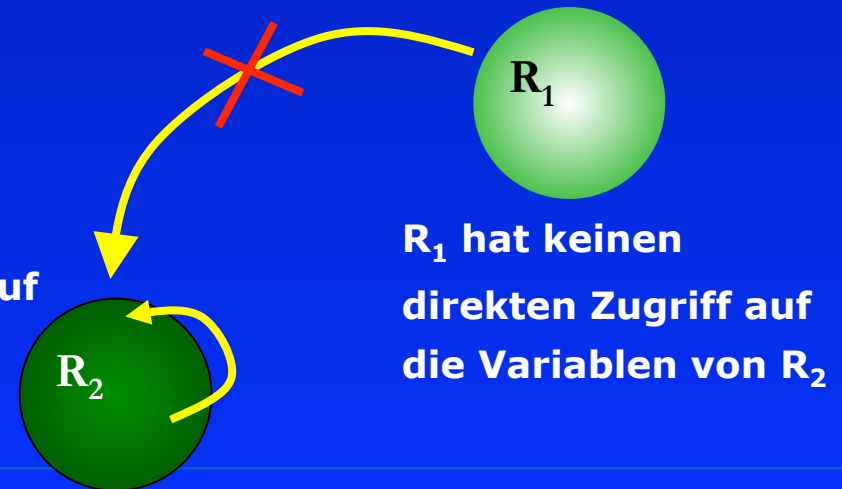
Sichtbarkeit von **private**-Variablen

Rechteck-Klasse

```
public class Rectangle {  
  
    private int x;  
    private int y;  
    private int width;  
    private int height;  
  
    /* Methoden */  
  
    public int area() {  
        return width*height;  
    }  
}
```

Zugriff nur innerhalb
der Rechteck-Klasse

R₂ hat Zugriff auf
seine eigenen
Variablen.



Parameterübergabe in Java

Alle Parameter werden in Java per Wert (**by-value**) übergeben.

Veränderungen der Parameter innerhalb der Methode bleiben **lokal**.

Die formalen Parameter einer Methode-Definition sind **Platzhalter**.

Beim Aufruf der Methode werden die **formalen Parameter** durch **reale Variablen** ersetzt, die den gleichen Typ der formalen Parameter haben müssen.

```
...  
int a = 5 ;  
Point p1 = new Point ( 1, 2 );  
g.methodeAufruf ( a, p1 );  
...
```

Nach Beendigung des Methoden-Aufrufs haben sich der Wert von **a** sowie der Referenz-Wert **p1** nicht geändert.

```
public class Parameteruebergabe {

    public static int fakultaet ( int n ) {
        int fac = 1;
        if (n>1)
            while (n>1) {
                fac = fac*n;
                n--;
            }
        return fac;
    }

    public static void main ( String[] args ) {
        int n = 20;
        System.out.println ( "n=" + n );
        fakultaet ( n );
        System.out.println( "n=" + n );
    }
}
```

Parameterübergabe mit primitiven Datentypen

Die Variable **n** wird solange verändert, bis sie gleich **0** wird.

Nur eine Kopie des Parameterwertes wird übergeben.

Nach Beendigung des Methoden-Aufrufs hat sich der Wert von **n** nicht geändert.

n=20

n=20

Parameterübergabe mit Referenz-Variablen (Objekte)

```
public class Parameteruebergabe {

    public static void scale( Rechteck r, int s ) {
        r.height = r.height*s;
        r.width = r.width*s;
    }

    public static Rechteck clone( Rechteck r ){
        return new Rechteck(r.x, r.y, r.width, r.height );
    }

    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(10,20,30,30);
        scale( r1 , 2 );
        Rechteck r2 = clone( r1 );
        System.out.println( r2.flaeche() );
    }
}
```

[illegible]

Parameterübergabe mit Referenz-Variablen (Objekte)

```
public class Parameteruebergabe {

    public static void scale( Rechteck r, int s ) {
        r.height = r.height*s;
        r.width = r.width*s;
    }

    public static Rechteck clone( Rechteck r ){
        return new Rechteck(r.x, r.y, r.width, r.height );
    }

    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(10,20,30,30);
        scale( r1 , 2 );
        Rechteck r2 = clone( r1 );
        System.out.println( r2.flaeche() );
    }
}
```

152

	10
	20
	30
	30

Parameterübergabe mit Referenz-Variablen (Objekte)

```
public class Parameteruebergabe {

    public static void scale( Rechteck r, int s ) {
        r.height = r.height*s;
        r.width = r.width*s;
    }

    public static Rechteck clone( Rechteck r ){
        return new Rechteck(r.x, r.y, r.width, r.height );
    }

    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(10,20,30,30);
        scale( r1 , 2 );
        Rechteck r2 = clone( r1 );
        System.out.println( r2.flaeche() );
    }
}
```

r1: **152**

152

10

20

30

30

Parameterübergabe mit Referenz-Variablen (Objekte)

```
public class Parameteruebergabe {

    public static void scale( Rechteck r, int s ) {
        r.height = r.height*s;
        r.width = r.width*s;
    }

    public static Rechteck clone( Rechteck r ){
        return new Rechteck(r.x, r.y, r.width, r.height );
    }

    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(10,20,30,30);
        scale( r1 , 2 );
        Rechteck r2 = clone( r1 );
        System.out.println( r2.flaeche() );
    }
}
```

r1: **152**

152

Parameterübergabe mit Referenz-Variablen (Objekte)

```
public class Parameteruebergabe {

    public static void scale( Rechteck r, int s ) {
        r.height = r.height*s;
        r.width = r.width*s;
    }

    public static Rechteck clone( Rechteck r ){
        return new Rechteck(r.x, r.y, r.width, r.height );
    }

    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(10,20,30,30);
        scale( r1 , 2 );
        Rechteck r2 = clone( r1 );
        System.out.println( r2.flaeche() );
    }
}
```

r1:	152
r2:	136

152	10
	20
	60
	60
136	10
	20
	60
	60

Parameterübergabe mit Referenz-Variablen (Objekte)

```
public class Parameteruebergabe {

    public static void scale( Rechteck r, int s ) {
        r.height = r.height*s;
        r.width = r.width*s;
    }

    public static Rechteck clone( Rechteck r ){
        return new Rechteck(r.x, r.y, r.width, r.height );
    }

    public static void main(String[] args) {
        Rechteck r1 = new Rechteck(10,20,30,30);
        scale( r1 , 2 );
        Rechteck r2 = clone( r1 );
        System.out.println( r2.flaeche() );
    }
}
```

Nach Beendigung des Methoden-Aufrufs hat sich der Referenz-Wert **r1** nicht verändert.

r1: **152**

r2: **136**

152

136

Namenskonventionen in Java

Klassen und Schnittstellen	Klassennamen sollen Namenswörter sein und der erste Buchstabe soll groß geschrieben werden.	<code>class</code> Circle <code>Interface</code> Shape
Methoden	Methoden sind Aktionen oder Operationen und sollen Verben sein. Der erste Buchstabe soll klein geschrieben werden.	<code>paint()</code> <code>draw()</code> <code>run()</code> <code>getColor()</code>
Instanzvariablen	Der erste Buchstabe soll klein geschrieben werden. Der Name soll aussagekräftig über ihren Inhalt sein. Wenn mehrere Worte benutzt werden, sollen Großbuchstaben dazwischen geschrieben werden.	<code>maxDistance</code> <code>width</code> <code>breite</code>
Konstanten	Alle Buchstaben sollen groß geschrieben werden. Wenn der Name aus mehreren Worten besteht, sollen '_' dazwischen stehen.	<code>MAX_DISTANCE</code> <code>WEST</code>

Aufzählung-Datentyp

Aufzählungsdantentypen erlauben es uns, Variablen zu definieren, denen nur eine bestimmte Anzahl von konstanten Namen zugewiesen werden kann.

Der Aufzählungsdantentyp in Java ist viel mächtiger als in allen anderen Programmiersprachen.

Vorteile: Statische Typüberprüfung ist möglich (Compiler)

Die Programme sind viel lesbarer

Typ-spezifische Operationen sind definierbar

Der Implementierungsaufwand ist sehr gering

Kleiner Datentyp, der innerhalb einer Klasse definiert wird.

Dadurch Vermeidung zu vieler kleiner Klassendefinitionen.

Aufzählungs-Datentypen

```
public class Enum_Beispiel {
    public enum Season {WINTER, SPRING, SUMMER, FALL};
    Season s = Season.FALL;

    public Season jahreszeit() {
        return s;
    }

    public static void main( String[] main ){
        Season s1 = Season.SUMMER;
        Season s2 = Season.FALL;
        System.out.println( s1 );
        System.out.println( s1.equals(s2) );
        Enum_Beispiel e = new Enum_Beispiel();
        System.out.println( e.jahreszeit() );
        // Season s3 = 1; // Typfehler!!!
    }
}
```

Ein neuer Datentyp mit dem Namen Season wurde vereinbart, der nur die konstanten Namen **Season.WINTER**, **Season.SPRING**, **Season.SUMMER** und **Season.FALL** annehmen kann.

Ausgabe:

```
SUMMER
false
FALL
```

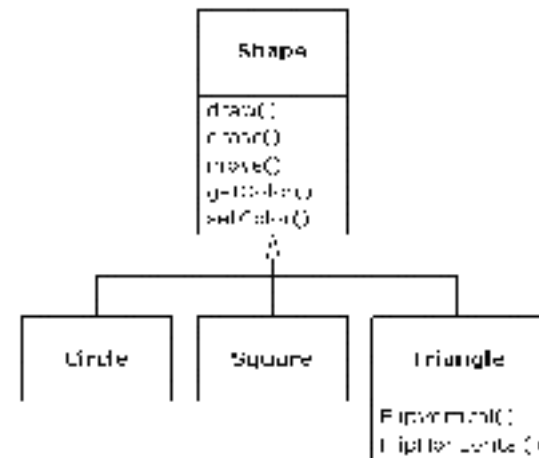
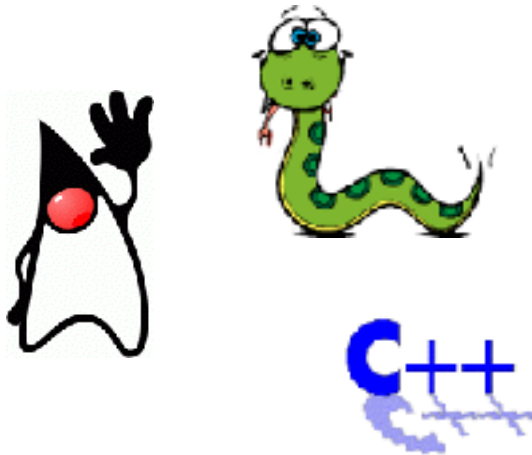
Beispiele:

```
public class Enum_Beispiele {

    public enum Colors{ RED, YELLOW, GREEN};
    public enum Direction { SOUTH, NORTH, EAST, WEST };
    public enum Geldschein {
        FUENF ( 5 ),
        ZEHN ( 10 ),
        ZWANZIG ( 20 ),
        FUENFZIG ( 50 ),
        HUNDERT ( 100 );

        private Geldschein(int w){
            wert = w;
        }
        final int getWert(){
            return wert;
        }
        private final int wert;
    };
}
```

Vererbung



Vererbung

Ein wesentliches Merkmal objektorientierter Sprachen ist die Möglichkeit, Eigenschaften vorhandener Klassen auf neue Klassen zu übertragen. (**Wiederverwendbarkeit**)

Durch Hinzufügen neuer Elemente oder Überschreiben der vorhandenen kann die Funktionalität der abgeleiteten Klasse erweitert werden.

Was ist **Vererbung**?

Übernahme aller Bestandteile einer Klasse in eine Unterklasse, die als **Erweiterung** oder **Spezialisierung** der Klasse definiert wird.

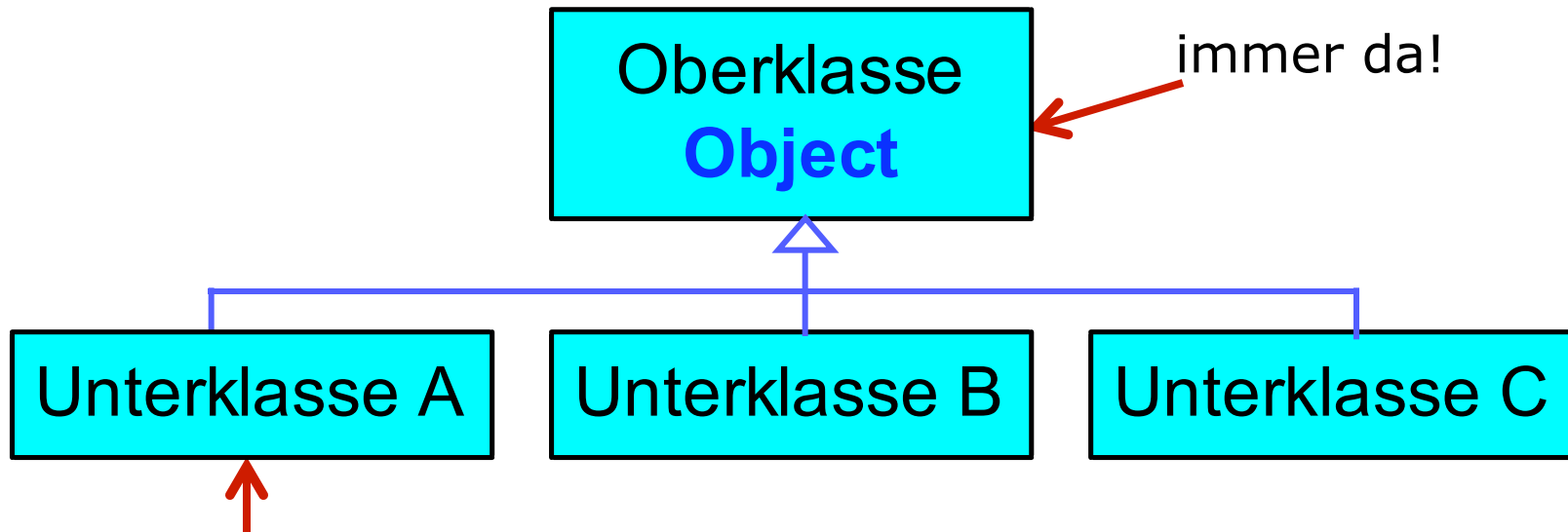
Objekte einer **Unterklasse** besitzen alle Eigenschaften und Methoden ihrer Oberklasse

+

die Erweiterungen, die in der Unterklasse selber definiert worden sind.

Vorsicht!

Klassenhierarchie

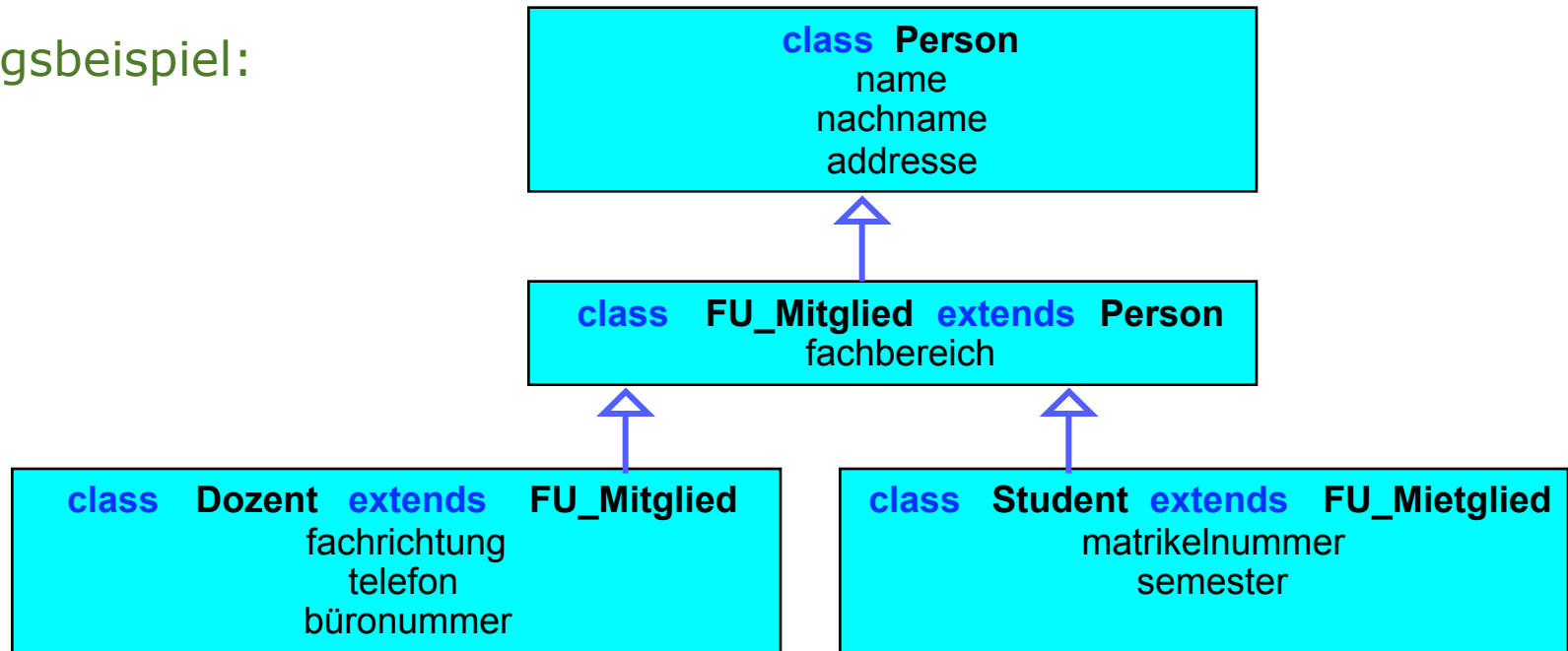


Die **Unterklasse A** besitzt alle Eigenschaften und Methoden ihrer Oberklasse

+

die Erweiterungen, die in der Unterklasse A definiert worden sind.

Vererbungsbeispiel:

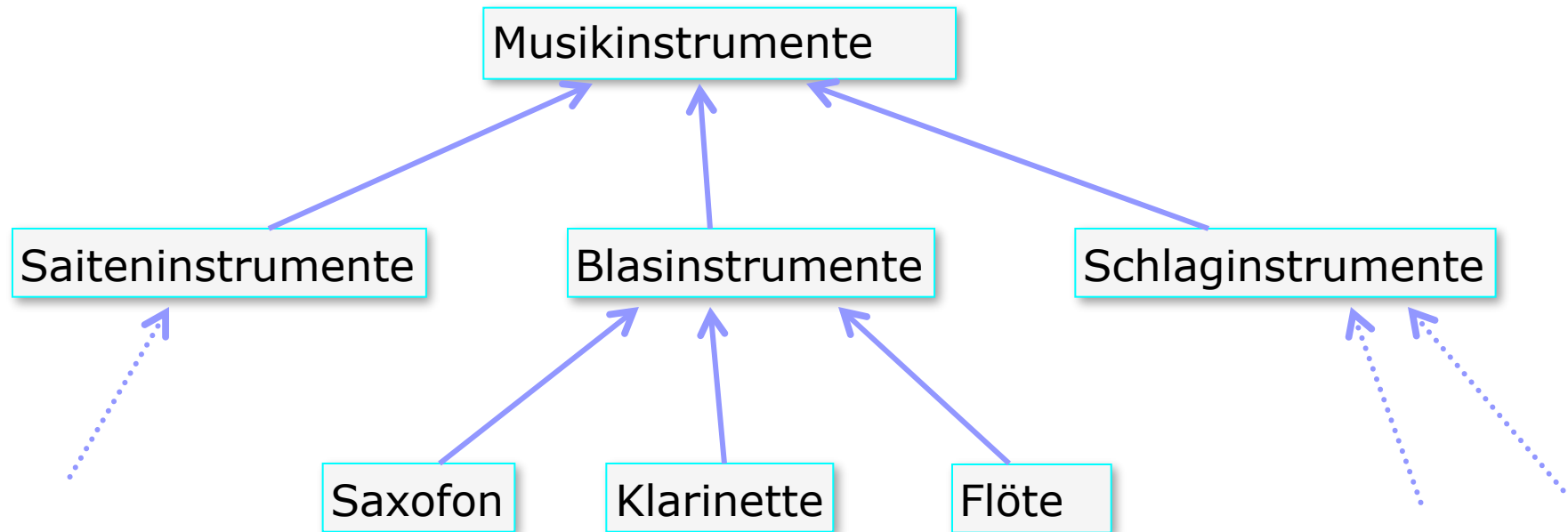


Dozent name
nachname
adresse
fachbereich
fachrichtung
telefon
büronummer

Student name
nachname
adresse
fachbereich
matrikelnummer
semester

Die Methoden
werden auch
vererbt.

Klassifizieren von Objekten



Bei guten Modellierungen klingt es logisch, wenn gesagt wird, dass ein Element der Unterklasse auch ein Element aller ihrer Oberklassen ist.

Eine Flöte ist ein Blasinstrument

Eine Flöte ist ein Musikinstrument

Vererbung

FU_Mitglied ist eine Unterklasse

"Spezialisierung "

"Erweiterung"

"**is-a-Relation**"

"Ableitung"

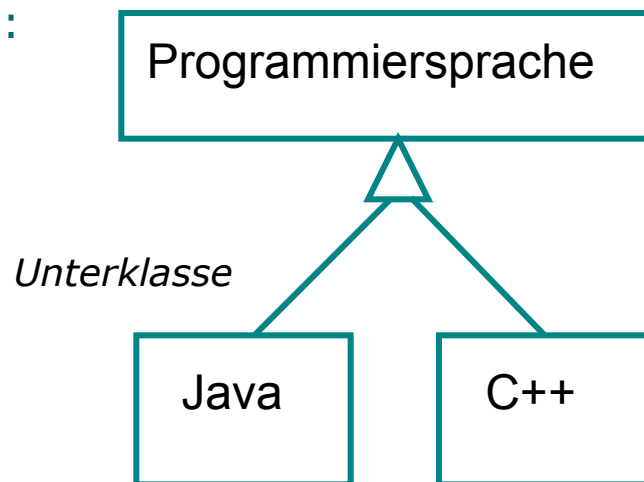
von **Person**

- Die Kunst ist es, eine möglichst gute Klassenhierarchie für die Modellierung von Softwaresystemen zu finden.
- OOP ist keine Religion.
- Nicht alle Teile eines Problems können gut mit rein objektorientierten Techniken gelöst werden.
- Nicht immer gelingt es, eine saubere Klassenhierarchie zu finden!

Verfeinerung und Verallgemeinerung

Verfeinerungen sind Beziehungen zwischen gleichartigen Elementen unterschiedlichen Detaillierungsgrades.

Beispiel:



Die Spezialisierung wird als Generalisierungs-Pfeil dargestellt. Er zeigt in Richtung der allgemeineren Klasse.

Vererbungsbeispiel:

Haustier

```
public class Haustier {

    public static enum Zustand {
        TUT_NICHTS,
        SPIELT,
        SCHLAEFT,
        ISST,
        SPRICHT
    };

    public String name;
    public String lieblingsessen;
    public Person besitzer;
    public String laute;
    private Zustand zustand;

    public Haustier(String name){
        this.name = name;
        this.zustand = Zustand.TUT_NICHTS;
        this.laute = "";
    }
    ...
}
```

Haustier

```
public class Haustier {
    . . .

    public void sprich(){
        zustand = Zustand.SPRICHT;
    }
    public void iss(){
        zustand = Zustand.ISST;
    }
    public void schlaf(){
        zustand = Zustand.SCHLAEFT;
    }
    public void ausruhen(){
        zustand = Zustand.TUT_NICHTS;
    }
}
```


Katze

```
public class Katze extends Haustier {  
  
    public Katze( String name ){  
        super( name );  
        this.lieblingsessen = "Mäuse";  
        this.laute = "Miau! Miau!";  
    }  
  
    public void sprich(){  
        super.sprich();  
        System.out.println(laute);  
    }  
    ...  
}
```

Hund

```
public class Hund extends Haustier {

    public Hund( String name ){
        super( name );
        this.lieblingessen = "Fleisch";
        this.laute = "Guau! Guau";
    }

    public void sprich(){
        super.sprich();
        System.out.println(laute);
    }
    . . .
}
```

Überschreiben von Feldern

- Es gibt bei Instanzvariablen keine dynamische Bindung, weil Instanzvariablen im Gegensatz zu Methoden einen anderen Typ haben können als in der Superklasse.
- Die Instanzvariable **name** in der **Katze-Klasse** verdeckt die Instanzvariable **name** in **Haustier**.
- Zugriff auf Instanzvariablen mit dem gleichen Namen in der Oberklasse erfolgt nur mit.

super.zustand

Konstruktoren in Unterklassen

- **Konstruktoren werden nicht vererbt**, d.h. Unterklassen müssen jeweils eigene Konstruktoren angeben und die Oberklassenkonstruktoren explizit aufrufen.
- Wenn man keinen Konstruktor der Oberklasse verwendet, wird *nur* der implizite default-Konstruktor **super()** aufgerufen.
- Wenn man einen Konstruktor der Oberklasse verwenden will, muss der **Aufruf am Anfang** der jeweiligen Konstruktoren stehen.

```
public class Person {
```

```
    public static int anzahl = 0;
```

```
    public String vorname;
```

```
    public String nachname;
```

```
    private Date geburtsdatum;
```

```
    public Person (String vorname, String nachname, Date geburtsdatum) {
```

```
        this.vorname = vorname;
```

```
        this.nachname = nachname;
```

```
        this.geburtsdatum = geburtsdatum;
```

```
        anzahl++;
```

```
    }
```

```
    public Person () {
```

```
        this ( "", "", new Date() );
```

```
    }
```

```
    ...
```

```
...  
public Date getGeburtsdatum() {  
    return geburtsdatum;  
}  
  
public void setGeburtsdatum( Date geburtsdatum ) {  
    Date heute = new Date();  
    if ( geburtsdatum.after(heute) )  
        System.err.println( "Falsches Geburtsdatum" );  
    else  
        this.geburtsdatum = geburtsdatum;  
}  
  
public int alter(){  
    Date heute = new Date();  
    long time = heute.getTime() - geburtsdatum.getTime();  
    time = time/1000;  
    return (int) (time/(365*24*3600));  
}  
} // end of class Person
```

Vererbung (Beispiel):

```
public class Student extends Person {
    // Klassenvariablen
    public static long semesterDauer = 3600*24*183;
    // Instanzvariablen
    public String fachbereich;
    public int matrikelnr;
    private Date anfangsdatum;
    // Konstruktor
    public Student( Date anfangsdatum, String fachbereich, int matrikelnr {
        super();
        this.anfangsdatum = anfangsdatum;
        this.fachbereich = fachbereich;
        this.matrikelnr = matrikelnr;
    }
    . . .
}
```

Der Konstruktors der Oberklasse wird hier aufgerufen.

Vererbung (Beispiel)

```
public class Student extends Person {
    . . .
    // Instanzmethoden
    public Date getAnfangsdatum() {
        return anfangsdatum;
    }
    public void setAnfangsdatum(Date anfangsdatum) {
        this.anfangsdatum = anfangsdatum;
    }
    public int semester(){
        Date heute = new Date();
        long time = heute.getTime() - anfangsdatum.getTime();
        time = time/1000;
        return (int)(1 + time/semesterDauer);
    }
}
```


Vererbung

```
public class TestPersonStudent {

    public static void main(String[] args) {
        Date geburt1 = Datum.toDate( "23.02.1985" );
        Date geburt2 = Datum.toDate( "11.05.1983" );
        Person p1 = new Person( "Peter", "Meyer", geburt1 );
        System.out.println( p1.getGeburtsdatum().toString() );
        Student s1 = new Student( "Sandra", "Smith", geburt2 );
        s1.setAnfangsdatum( Datum.toDate( "01.10.2004" ) );
        System.out.println( s1.getGeburtsdatum().toString() );
        System.out.println( s1.semester() );
        System.out.println( p1.alter() );
        System.out.println( "Anzahl der Personen = " + Person.anzahl );
    }
}
```

super

```
public class Circle {  
    double x, y, r;  
    public Circle(){  
    }  
    ...  
}
```

Der Konstruktor der Oberklasse wird implizit aufgerufen.

```
public class Circle {  
    double x, y, r;  
    public Circle(){  
        super();  
    }  
    ...  
}
```

Der Konstruktor der Oberklasse wird explizit aufgerufen.

Wenn man keinen Konstruktor der Oberklasse verwendet, wird *nur* der implizite default-Konstruktor **super()** aufgerufen.

super

```
public class Person {
    String vorname;
    String nachname;

    public Person ( String vorname, String nachname ) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
}
```

```
public class Student extends Person {
    String fachbereich;

    public Student ( String vorname, String nachname, String fachbereich ) {
        super ( vorname, nachname );
        this.fachbereich = fachbereich;
    }
}
```

Ein Konstruktor der Oberklasse muss hier explizit aufgerufen werden.

super

Das Schlüsselwort **super** dient nicht nur dazu, um Konstruktoren der Oberklasse aufzurufen sondern wird verwendet, um den Zugriff auf verdeckte Instanzvariablen und Methoden der Oberklasse zu ermöglichen.

Konstruktoren werden nicht vererbt, d.h. Unterklassen müssen jeweils eigene Konstruktoren angeben und die Oberklassenkonstruktoren benutzen.

super

Beispiel:

```
public class O {
    int y;
    int x = 10;

    int q() { return x*x; }
}
```



```
public class U extends O
{
    int x = 2;

    int q() { return x*x; }
    int q1() { return super.x*super.x; }
    int q2() { return super.q(); }
}
```

Verdeckt die **x** der Oberklasse

Verdeckt die **q**-Methode der Oberklasse

```
...
U u = new U();
System.out.println( u.q1() );
System.out.println( u.q() );
System.out.println( u.q2() );
...
```

Ausgabe: **100**
4
100

Beispiel:

```
public class Rectangle {  
    public int x, y;  
    private int width, height;  
    ...  
} //end of class Rectangle
```



```
import java.awt.Color;  
import java.awt.Graphics;
```

```
public class DrawableRectangle extends Rectangle {  
    private Color color = Color.BLACK;  
  
    public Color getColor() {  
        return color;  
    }  
  
    public void setColor(Color color) {  
        this.color = color;  
    }  
  
    public void draw( Graphics g ){  
        g.setColor(this.color);  
        g.drawRect(x,y,getWidth(),getHeight());  
    }  
} // end of class ColorRectangle
```

Beispiel:

```
import java.awt.*;

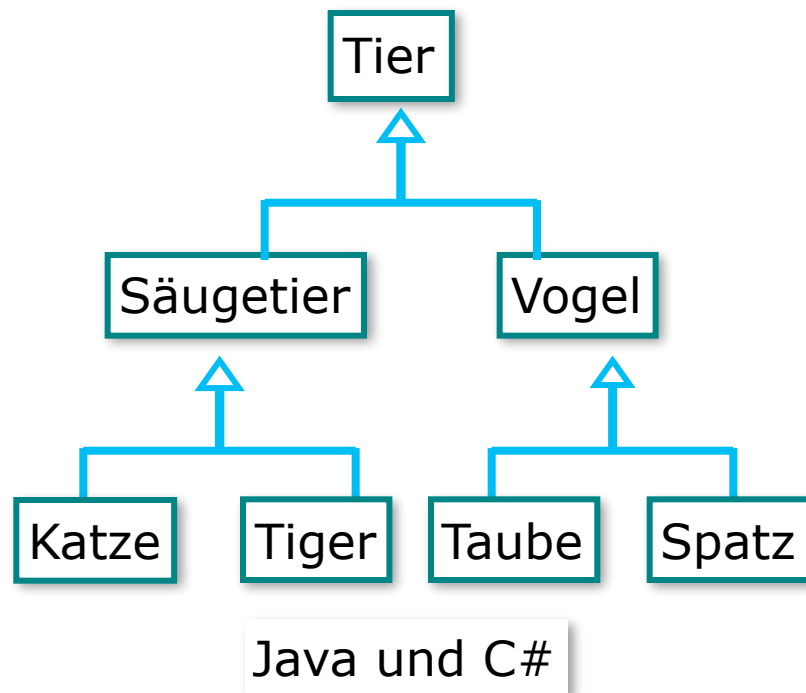
public class Fenster extends JFrame {
    // Konstruktor
    public Fenster() {
        setTitle( "Fensterchen" );
        setLocation( 500,300 );
        setSize( 200, 200 );
        setVisible( true );
    }
} // end of class MyFenster
```

Die Klasse Fenster vererbt alle Eigenschaften und Methoden, die die Oberklasse JFrame bereits besitzt.

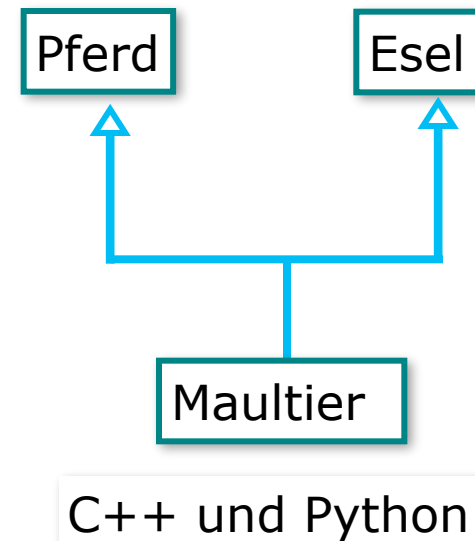
Einfache und mehrfache Vererbung

Beispiele:

Einfache Vererbung

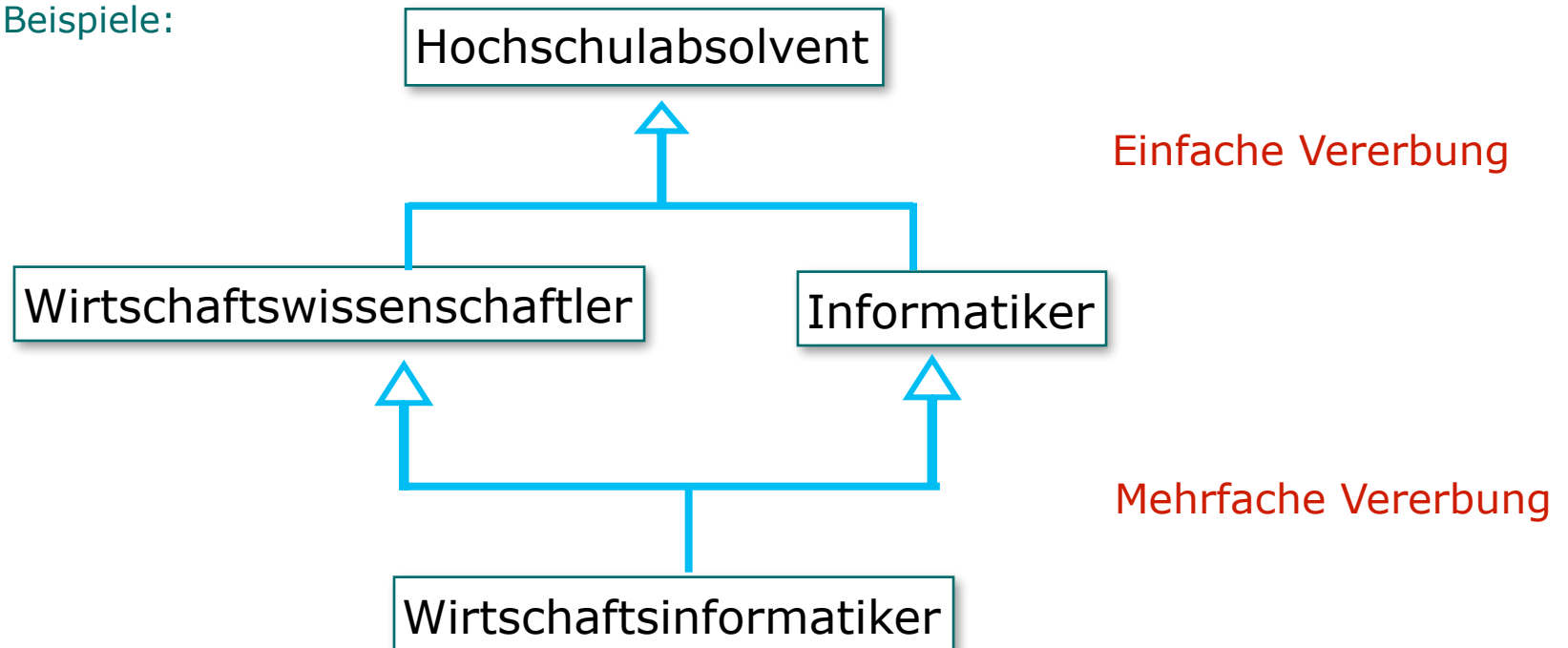


Mehrfache Vererbung



Einfache und mehrfache Vererbung

Beispiele:



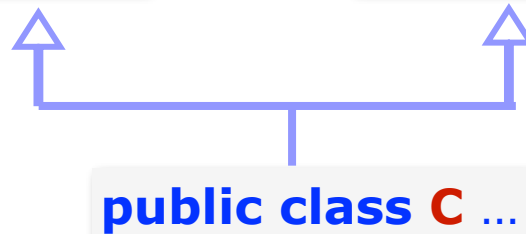
Einfache und mehrfache Vererbung

Probleme:

Bei mehrfacher Vererbung besteht die Gefahr der **Namenskollisionen**, weil Methodennamen oder Attribute mit gleichem Namen aus verschiedenen Oberklassen vererbt werden können.

```
public class A
int x;
int getX() {...}
```

```
public class B
int x;
int getX() {...}
```




Welche der beiden `getX`-Methoden sollen in der Klasse C verwendet werden?

Überschreiben von Methoden

```
public class Person {  
    ...  
    public String name() {  
        return name;  
    }  
}
```

Der aktuelle Typ des aufgerufenen Objekts bestimmt, welche Methode tatsächlich benutzt wird.



```
public class Mann extends Person {  
    ...  
    public String name() {  
        return "Herr " + super.name();  
    }  
}
```

Überschreiben von Methoden

Der aktuelle Typ des aufgerufenen Objekts bestimmt, welche Methode tatsächlich benutzt wird

```
...  
Shape figure_1 = new Rectangle(0, 0, 10, 10);  
Shape figure_2 = new Circle(0.0, 0.0, 1.0);  
...  
figure_1.paint();  
figure_2.paint();  
...
```

Überladen von Methoden

Die Methoden werden vom Übersetzer durch Anzahl und Typ der Parameter unterschieden.

```
public void draw ( String s )
```

```
public void draw ( int i )
```

```
public void draw ( double d )
```

```
public void draw ( double d, int x, int y )
```

Die Parametertypen bestimmen die *Signatur* einer Methode.

Überschreiben von Methoden

- Eine als **final** markierte Methode kann nicht in Unterklassen überschrieben werden

public final String nachname()

- Von einer **final** Klasse können keine Unterklassen gebildet werden

public final class String { ... }

Überladen von Konstruktoren

Die Konstruktoren werden vom Übersetzer durch Anzahl und Typ der Parameter unterschieden.

```
public Person ( String vorname, String nachname, Date geburtsdatum )
```

```
public Person ( String vorname, String nachname )
```

```
public Person ( String vorname )
```

```
public Person ()
```

Signatur des Konstruktors

Die Parametertypen bestimmen die *Signatur* des Konstruktors.

Vererbung

Ein wesentliches Merkmal objektorientierter Sprachen ist die Möglichkeit, Eigenschaften vorhandener Klassen auf neue Klassen zu übertragen. (**Wiederverwendbarkeit**)

Durch Hinzufügen neuer Elemente oder Überschreiben der vorhandenen kann die Funktionalität der abgeleiteten Klasse erweitert werden.