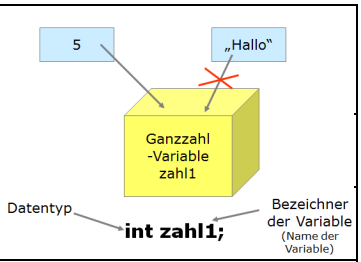
Überblick Java-Befehle

Variablen / Datentypen

Definition Definition und Initialwert Definition von mehreren Variablen	**int iZahl1**; **int iZahl1=10**; long lZahl1=60000000000, lZahl2=-7, x;
Konstante (Wert nicht mehr veränderbar)	**final** double PI=3.14159265;
Wertzuweisungen (linkeSeite ← rechte Seite)	**iZahl1=iZahl +7**;
Datentypen	**int, long, float, double, boolean, char, String**, …



Eingabe/Ausgabe

Ausgabe mit neuer Zeile	**System.out.println**("Hallo Welt!");
Ausgabe ohne neue Zeile	System.out.**print**("Hallo ");
Ausgabe Text und Variable	System.out.println("Wert von (x|y)=("+x+"|"+y+")");
Eingabe definiert (einfach)	String name="Johann";

Eingabe einlesen	//import java.util.Scanner; über der Klasse angeben
	Scanner scan = new Scanner(System.in);
	String name;
	int zahl;
	sout("Bitte Name eingeben:");
	name=**scan.nextLine();**
	sout("Bitte Zahl eingeben:");
	zahl =scan.nextInt();

Operatoren

Arithmetische Operatoren
Beispiele
	+ - * / %
	int a = 1 / 2; // a = 0 da ganzzahlige Division
	int rest = 6 % 4; // rest = 2, da 6 = 1*4+2
	float z = (2.5+3.5)*(0.5); // z = 3.0f

Logische Operatoren Beispiele	**&&** (Und) **||** (Oder) **!** (Nicht) boolean erg1 = (true && true) && !false || true;

Vergleichsoperatoren
Beispiele
	== < <= > >= != .equals()
	if (x<=15 && x>=5) sout("zwischen 5 und 15");
	if (eingabe.equals("Ende")) {sout("Programm beendet");}

Mathematische Ausdrücke	double wurzel = **Math.sqrt**(9); // wurzel = 3
	int zufall = (int)(**Math.random()***6)+1; // Zufallszahl zwischen 1 und 6

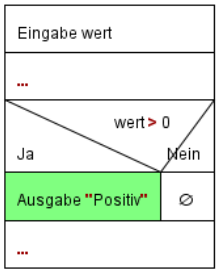
Variablen umwandeln (Typecast)

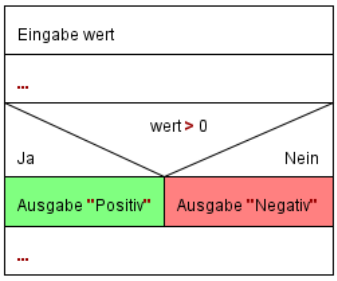
int ↔ float	int iZahl1=5;
	float fZahl1=10.8f;
	
	fZahl1 = **(float)** iZahl1;
	iZahl1 = **(int)** fZahl1; // danach hat iZahl1 den Wert 10

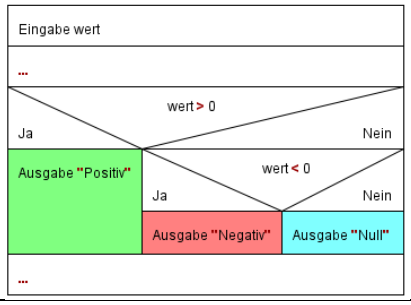
String ↔ int, String ↔ float	String sZahl1="56";
	iZahl1 = **Integer.valueOf**(sZahl1);
	sZahl1 = **String.valueOf**(iZahl1);
	fZahl1 = **Float.valueOf**(sZahl1);
	sZahl1 = String.valueOf(fZahl1);

Überblick Java-Befehle

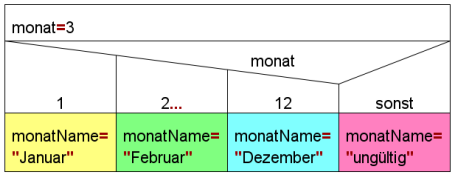
Verzweigungen (if-Bedingung)

Einseitige Verzweigung	if(wert>0) {System.out.println("Positiv");} 

Zweiseitige Verzweigung	**if**(wert>0) {
	  System.out.println("Positiv");
	} **else** {
	  System.out.println("Nicht negativ");
	}
	 

Verzweigungsketten	**if**(wert>0) {
	  System.out.println("Positiv");
	} **else if**(wert<0) {
	  System.out.println("Negativ");
	} **else** {
	  System.out.println("Null");
	}
	 

Komplexe Verzweigung	if (**a<20 && a>10**) {sout("Zwischen 10 und 20");}
	if (**a==1 || a==11 || a==111**) {sout("Etwas mit 1");}
	if (**eingabe!=null && !eingabe.equals("")**){sout("Eingabe nicht leer");}

Case / switch	int monat = 3;
	switch (monat) {
	  case 1: monatString = "Januar";
	  break;
	  case 2: monatString = "Februar";
	  break;
	  ...
	  case 12: monatString = "Dezember";
	  break;
	  default: monatString = "Ungültiger Monat";
	  break;
	}
	 

Überblick Java-Befehle

Schleifen

For-Schleife For-Schleife bei 0 beginnend For-Schleife Zweierschritte For-Schleife rückwärts	**for**(int i=**1**;**i<=10**;i++) {sout("Zahl "+i);} **for**(int i=**0**;**i<10**;i=i++) {sout("Zahl "+i);} **for**(int i=1;i<=10;**i=i+2**) {sout("Zahl "+i);} **for**(int i=10;i>=0;i--) {sout("Zahl "+i);} Für i = 1 bis 10 Ausgabe "Zahl" i

While-Schleife	int i=1;
	while(i<=10) {
	sout("Zahl "+i);
	i++;
	}
	i=1
	Solange i<=10
	Ausgabe "Zahl" i
	i=i+1

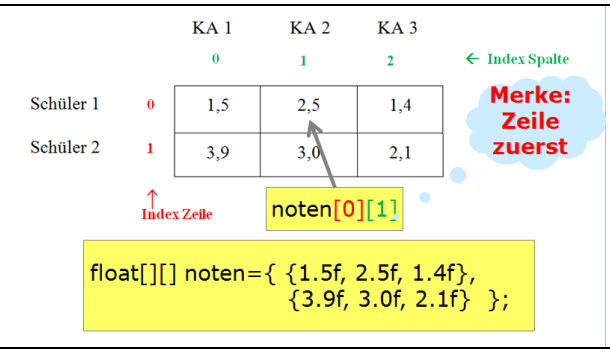
Do-Schleife	int i=1;
	do {
	sout("Zahl "+i);
	i++;
	} **while**(i<=10);
	i=1
	Ausgabe "Zahl" i
	i=i+1
	Solange i<=10

Verschachtelte Schleife	**for**(int **zeile**=1;zeile<=10;zeile++) {
	for(int **spalte**=1;spalte<=10;spalte++) {
	sout("* an ("+zeile+"|"+spalte+")";
	} // of end for spalte
	} // of end for zeile

Arrays

Array initialisieren mit Werte	**int[]** zahlen = **{1,2,3,4,5,6}**;
Array initialisieren, Werte später eintragen	**int[]** zahlen = **new int[1000]**; for(int i=0;i<=zahlen.length;i++) { **zahlen[i]=i**; }
Auf das i-te Element zugreifen	System.out.println(**zahlen[i]**);
Länge des Arrays bestimmen	int laenge = **zahlen.length**;
ForEach-Schleife für Arrays	for(**int zahl : zahlen**) {sout(zahl);}
Array ausgeben	sout(**Arrays.toString**(zahlen)); //import java.util.Arrays;

2Dim-Arrays

Array initialisieren mit Werten	

Array initialisieren, Werte später eintragen	**float[][]** noten = **new float[2][3]**;
	for(int **schueler**=0;schueler<noten.length[];schueler++) {
	for(int **ka**=0;spalte<ka[schueler].length;ka++) {
	zahlen[schueler][ka]=((int)Math.random()*6)+1;
	}
	}

Auf das ij-te Element zugreifen	System.out.println(**noten[0][1]**);
Länge des Arrays bestimmen	int anzSchueler = **noten.length**; int anzKA = **noten[0].length**;

Methoden

Statische Methodendefinition ohne Rückgabewert	**public static void printHalloWelt() {** sout("Hallo Welt"); **}**
Statische Methodendefinition mit Eingabeparameter, ohne Rückgabewert	**public static void printBegruessung(String text, int wieOft) {** for(int i=0;i<wieOft;i++) {sout("Hallo "+name);} **}**
Statische Methodendefinition mit Eingabeparameter, mit Rückgabewert	**public static double max(double a, double b) {** if(a<b) **return** a; else **return** b; **}**

Aufruf statischer Methoden	public static void main(String[] a) {
	  printHalloWelt();
	  printBegruessung("Carmen",10);
	  double maxWert=max(10.5,-4.5); sout(max);
	}

Methode eines Objekts	public class Wert {
	  int x;
	  public int getX(){ return x; }
	  public void setX(int wert){ x=wert; }
	}

Aufruf einer Methode eines Objekts	public static void main(String[] a) {
	  Wert w = new Wert();
	  w.setX(5);
	  sout("Wert="+**w.getX()**);
	}

Rekursive Methoden	public static void **down(int n)** {
	  // Abbruchkriterium
	  if (n==0) return;
	  // Verarbeitung
	  sout(n);
	  // Rekursionsschritt
	  down(n-1);
	}
	
	public static void main(String[] a) {
	  down(500);
	}

Überblick Java-Befehle

Die Klasse String

String anlegen	String s1 = **"Hallo";** String s2 = **new String**("Welt");
Strings verknüpfen	String s3 = s1 **+** "-" **+** s2; // "Hallo-Welt"
Länge des Strings	int len = s1.**length()**; // len=5
Strings vergleichen	if (s1.**equals**(s2)) {sout("ja");}
i-te Zeichen	char second = s1.**charAt**(1); // second='a'
Teilstring	String teil = s2.**substring**(1,3); // teil="el" (1 inklusiv, 3 exklusiv)
Beginnt mit	if (s1.**startsWith**("Hallo")) {sout("Begrüßung");}
Stelle des ersten Vorkommens	int aPos = s1.**indexOf**("ll"); // aPos=2 (-1 falls nicht gefunden)
Teilstring ersetzen	s3 = s1.**replace**("ll","LihaL"); // s3 = "HaLihaLo"
Regulären Ausdruck ersetzen	s3 = s3.**replaceAll**("(.*)-(.*)","$2-$1"); // s3 = "Welt-Hallo"
Leerzeichen Anfang/Ende löschen	s2= " Hallo ".trim() // s2= "Hallo"
Zahl in String umwandeln	String plz = **String.valueOf**(79100); // plz="79100"
String aufteilen	String[] res = s3.**split**("-"); // res = {"Hallo ","Welt"}

Die Klasse ArrayList

ArrayList anlegen	**ArrayList**<String> myList = new ArrayList<String>();
Elemente hinzufügen	myList.**add**("Hallo");
Länge der Liste	int len = myList.**size()**; // len=1
Vorkommen Element in Liste	boolean res=myList.**contains**("Welt") // res=false
Index Element in Liste	int aPos = myList.**indexOf**("Hallo"); // aPos=0 (-1 falls nicht gefunden)
i-te Element lesen	String el1=myList.**get**(0); // el1="Hallo"
i-te Element überschreiben	myList.**set**(0, "Hallo Welt");
i-te Element löschen	myList.**remove**(0);
Mehrere Elemente hinzufügen	**Arrays.asList(**"a","b","c"**);**
ArrayList kopieren	ArrayList<String> myListCopy = **new ArrayList<>(myList);** ArrayList<String> myListCopy = (ArrayList<String>) **myList.clone()**;
Array in Liste umwandeln	String[] array = {"a","b","c"}; ArrayList<String> myList1 = new ArrayList(**Arrays.asList(array)**);
Liste in Array umwandeln	ArrayList<String> myList2 = new ArrayList<>(); myList2.add("d"); myList2.add("e"); myList2.add("f"); String[] array2 = myList2.**toArray(new String[]{})**;

Die Klasse HashMap

HashMap anlegen	**HashMap**<String,String> dict = new HashMap<String,String>();
Elemente hinzufügen	dict.**put**("katze","cat"); dict.put("maus","mouse");
Wert zum Schlüssel finden	String res = dict.**get**("katze"); // res="cat"
Länge der HashMap	int len = dict.**size()**; // len=2
Vorkommen als Schlüssel	boolean res=dict.**containsKey**("katze"); // res=true
Vorkommen als Wert	boolean res=dict.**containsValue**("mouse"); // res=true

Über alle Schlüssel iterieren	**for (String key : dict.keySet())** {
	  sout(key + " = " + dict.get(key));
	} //maus = mouse katze = cat

Über alle Werte iterieren	for (String key : **dict.values()**) {
	  sout(key);
	} //mouse cat

Die Klasse Stack

Stack anlegen	Stack<String> stack = **new Stack<String>**();
Ist der Stack leer?	boolean leer = stack.**empty();**
Element auf den Stack legen	stack.**push**("Aufgabe1");
Element vom Stack nehmen	String el = stack.**pop()**;
Oberstes Element ansehen	String el = stack.**peek()**;

Die Klasse Deque (= Stack und Queue)

Deque anlegen	Deque<String> deque = **new ArrayDeque<String>**();
Ist die Deque leer?	boolean leer = deque.**isEmpty();**
Element hinzufügen	deque.**add**("a"); deque.add("b"); deque.add("c"); //{"a","b","c"}
Element nehmen (als Queue)	String el = stack.**poll**() (removeFirst); // el="a"
Element ansehen (als Queue)	String el = stack.**peek**(); (getFirst);// el="a"
Element nehmen (als Stack)	String el = stack.**pollLast**(); (removeLast); // el="c"
Element ansehen (als Stack)	String el = stack.**peekLast();** (getLast) // el="c"

Dateien einlesen / schreiben

String in Textdatei schreiben	//import java.io.*; über der Klasse angeben
	String s = "Zeile 1\nZeile2 \nEnde!";
	try {
	FileWriter fw = new **FileWriter**("datei.txt");
	fw.**write**(s);
	fw.**close**();
	} catch (IOException ex) {
	System.out.println("Fehler bei Schreiben!");
	}

Textdatei zeilenweise einlesen	//import java.io.*; über der Klasse angeben
	String filename="test.txt";
	try {
	FileReader file = new **FileReader**(filename);
	BufferedReader buff = new **BufferedReader**(file);
	String line;
	while((line=buff.readLine())!=null) {
	System.out.println(line);
	}
	buff.close();
	} catch (IOException e) {System.out.println("Error:"+e);}

Textdatei zeilenweise mit Scanner einlesen	//import java.io.*; import java.util.Scanner; über der Klasse angeben
	String filename="test.txt";
	try {
	Scanner s = **new Scanner(new File(filename))**;
	while (**s.hasNext()**) {
	System.out.println(**s.nextLine()**);
	}
	} catch (IOException e) {System.out.println("Error:"+e);}

Textdatei zeilenweise schreiben
 	
Alternativ ohne PrintWriter
	//import java.io.*; über der Klasse angeben
	String filename="test.txt";
	boolean append=true;
	try {
	FileWriter file = new **FileWriter**(filename,append);
	BufferedWriter buff = new **BufferedWriter**(file);
	PrintWriter out = new **PrintWriter**(buff);
	out.println("Neue Zeile");
	out.flush();
	out.close();
	} catch (IOException e) {System.out.println("Error:"+e);}
	
	…
	buff.append(text);
	buff.newLine();
	…

Überblick Java-Befehle

Objektorientierte Programmierung (OOP): Klassen und Objekte

Idee Klassen und Objekte

Klasse

Abstrakte Beschreibung von Eigenschaften (Attributen) und Aktionen (Methoden).

= Schablone

Eigenschaften
Name:
Haarfarbe:
Alter:

Aktionen
laufen
sitzen
schlafen
essen

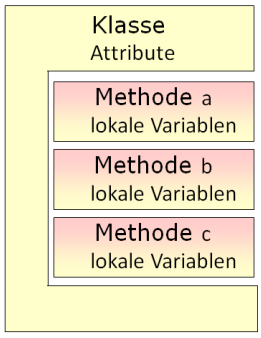
Objekt

Konkrete Instanz einer Klasse.

Eigenschaften
Name: Sascha Müll
Haarfarbe: blond
Alter: 18

Aktionen
laufen
sitzen
schlafen
essen

Umsetzung in Java



Eigenschaften

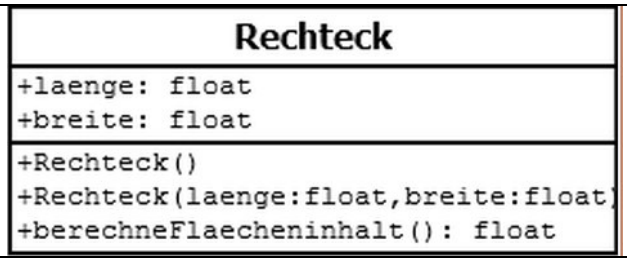
Aktionen

Implementierung in Java

```
public class Mensch {
 String name;
 String haarfarbe;
 int alter;

 public void schlafen(){S.o.p(name+"schläft");}
 public void sitzen(){S.o.p(name+"schläft");}
}
```

```
public class TesteMensch {
  /** Startmethode der Klasse */
  public static void main(String[] args) {
    Mensch patrick = new Mensch();
    patrick.name="Patrick";
    Mensch tim = new Mensch();
    tim.name="Tim";
    patrick.schlafen();
    tim.sitzen();
  }
```

UML



Name

Attribute

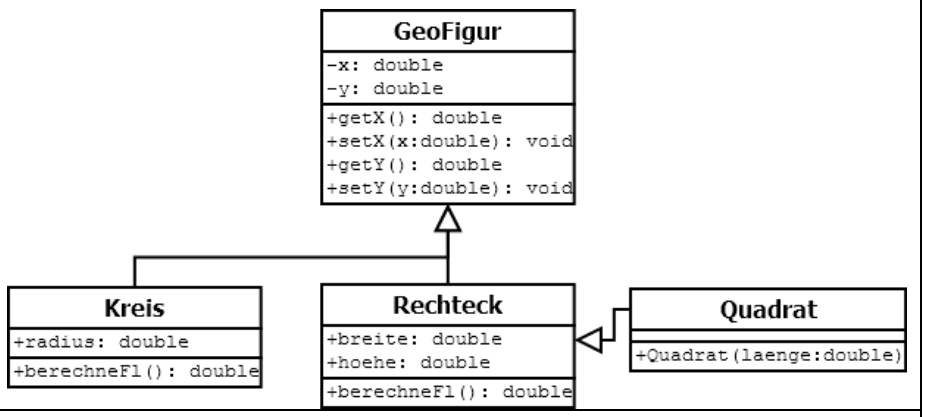
Methoden

OOP Konstruktoren

Konstruktoren heißen wie die Klasse und haben **keinen Rückgabewert**.
 	
Konstruktor mit Angabe von **Höhe und Breite** beim Erstellen des Objekts.
 	
Konstruktor mit Angabe von **nur der Breite** beim Erstellen des Objekts.
 	
Default-Konstuktor, ist immer implizit definiert, falls keine weiteren Konstruktoren angegeben sind.
	public class Rechteck {
	double laenge;
	double breite;
	
	public Rechteck(double l1, double b1){
	hoehe=l1;
	breite=b1;
	}
	
	public Rechteck(double breite) {
	this.breite=breite;
	}
	
	public Rechteck() {}
	}

Objekte erstellen	public class TesteGeoFigur {
	public static void main(String[] a) {
	Rechteck r1 = **new Rechteck(100,200);**
	Rechteck r2 = **new Rechteck(50);** r2.breite=80;
	Rechteck r3 = **new Rechteck();** r3.breite=300; r3.hoehe=400;
	}
	}

OOP Vererbung

UML: Oberklasse, Unterklasse	

Java: Oberklasse, Unterklasse	public class Kreis **extends** GeoFigur{
	public double radius;
	public double berechneFl() {return radius*radius*3.1415926;}
	}

Objekte nutzen Vererbung	public class TesteGeoFigur {
	public static void main(String[] a) {
	Rechteck r1 = new Rechteck();
	r1.setX(10); r1.setY(20); // verwendet Methode der Oberklasse
	Quadrat q1 = new **Quadrat(50)**; // setzt Attribute der Oberklasse
	sout(**q1.berechneFl()**); // verwendet Methode der Oberklasse
	}
	}

Überblick Java-Befehle

OOP Zugriffsrechte (Sichtbarkeiten)

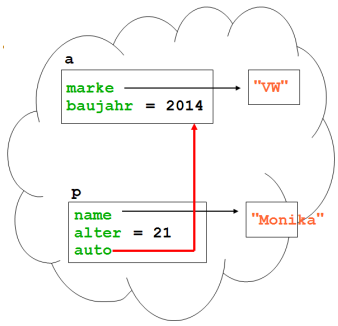
Bedeutung der Schlüsselwörter für die Zugriffsrechte auf Attribute und Methoden (Datenkapselung)	**Java** ➢ **public** ➢ Zugriff für alle Klassen ➢ **protected** ➢ für die eigene Klasse, package und Subklassen ➢ **(keins)** ➢ für Klassen im package ➢ **private** ➢ Zugriff nur für die eigene Klasse **UML** ➢ **+** ➢ **#** ➢ **-**

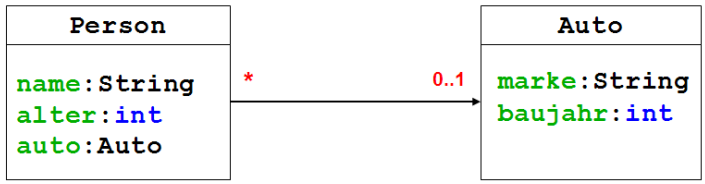
Getter- / Setter-Methoden
 	
Empfehlung: Attribute private und public Getter/Setter-Methoden für den Zugriff von Außen.
	public class GeoFigur {
	private double **x**;
	public getX() {return x;}
	public setX(double x) {this.x=x;}
	}

OOP Assoziationen

Beziehung (Assoziation) "Eine Person hat ein Auto" in UML	

Realisierung der Klassen in Java	`class Person{`
	`String name;`
	`int alter;`
	`Auto auto;`
	`}`
	
	`class Auto{`
	`String marke;`
	`int baujahr;`
	`}`

Realisierung der Objekte in Java (Beziehungen sind Referenzen).	`class TesteAuto{`
	`public static void main(String[]`
	`Auto a = new Auto();`
	`a.marke = "VW";`
	`a.baujahr = 2014;`
	`Person p = new Person();`
	`p.name = "Monika";`
	`p.alter = 21;`
	`p.auto = a;`
	`}`
	`}`
	
	 

Multiplizität (Kardinalität) Eine Person hat kein oder ein Auto. Ein Auto kann beliebig vielen Personen gehören.	

Überblick Java-Befehle

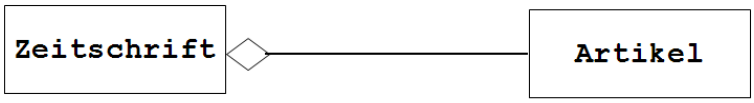
Realisierung von Assoziationen mit beliebig vielen Objekten.


```
class Person{
String name;
int alter;
ArrayList<Auto> autos;
}
```

```
class Auto{
String marke;
int baujahr;
}
```

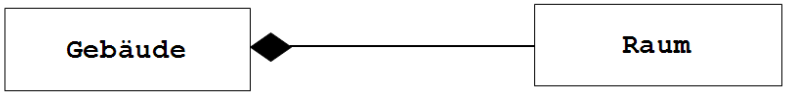
```
public class Hauptprogramm {

    public static void main(String[] args) {


        //Drei Auto-Objekte erstellen
        Auto mercedes = new Auto("Mercedes", 2012);
        Auto audi = new Auto("Audi", 2013);
        Auto bmw = new Auto("BMW", 2011);

        //Ein ArrayList-Objekt erstellen
        ArrayList<Auto> listeVonAutos = new ArrayList<Auto>();

        //Obige Auto-Objekte in die ArrayList hinzufügen
        listeVonAutos.add(mercedes);
        listeVonAutos.add(audi);
        listeVonAutos.add(bmw);

        Person hans = new Person("Hans", 23, listeVonAutos);

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

Aggregation
Ist-Teil-von-Beziehung



Jedes Objekt von Artikel ist Teil von einer Zeitschrift.
Verschwindet die Zeitschrift, so bleibt der Artikel jedoch noch bestehen.
Gleiche Implementierung wie bei der Assoziation.

Komposition
Ist-Teil-von-Beziehung
(… besteht aus …)



Jedes Objekt von Raum ist Teil von nur einem Gebäude.
Verschwindet das Gebäude, **muss** der Raum auch verschwinden.
Implementierung muss das garantieren (z.B. durch geeignete Konstruktoren).

OOP Methoden überladen / überschreiben

Methoden überladen:
Die gleiche Methode mit unterschiedlichen Eingabeparameter (Rückgabewert muss gleich bleiben),
	public class Rechteck {
	…
	public Rechteck(double laenge, double breite){…}
	public Rechteck(double breite) {…}
	public Rechteck() {…}
	}

Methode überschreiben:
Die Methode der Oberklasse wird in der Unterklasse neu definiert.
 	
Aufruf der überschriebenen Methode.
	public class GeoFigur {
	…
	public double berechneFl() {
	sout("Bitte in der Unterklasse definieren");
	return -1;}
	}
	
	public class Kreis extends GeoFigur {
	…
	public double berechneFl() {
	return pi*radius*radius;}
	}
	
	public class TesteGeoFigur {
	public static void main(String[] a) {
	Kreis k1 = new Kreis(10);
	sout(k1.berechneFl()); // ruft die Methode der Klasse Kreis auf
	sout(k1.**super**.berechneFl()); // ruft die Methode der Klasse GeoFigur auf
	}
	}

OOP Abstrakte Klassen und Methoden

Von abstrakten Klassen können **keine Objekte** erzeugt werden (nur von deren Unterklassen).
 	
Die Unterklasse **muss** die abstrakten Methoden implementieren.
	public **abstract** class GeoFigur {
	…
	public abstract double berechneFl() ;
	}
	
	public class Kreis extends GeoFigur{
	…
	public double berechneFl() {
	return pi*radius*radius;}
	}

OOP Interfaces

Ein Interface definiert einen eigenen Datentyp. In ihm wird festgelegt, welche Methoden die implementierende Klasse besitzen **muss**.	public **interface Paintable** {
	  public void paint(Graphics g);
	}
	
	public class Kreis **implements Paintable** {
	…
	  public void paint(Graphics g)
	  g.fillOval(x,y,radius,radius);
	  }
	}
	
	public class TestePaintable {
	  Paintable[] paintObj= new Paintable[100];
	  public static void main(String[] a) {
	  …
	  paintObj[0] = new Kreis(10);
	  …
	  for(Paintable obj : paintObj) {
	  obj.paint(g);
	  }
	  }
	}

OOP Polymorphie

Ein Rechteck ist ein Rechteck und gleichzeitig eine GeoFigur.
 	
Die Figuren werden als **Unterklasse erzeugt**, aber in einer Variablen der **Oberklasse gespeichert**.
 	
Der Aufruf **berechneFl()** nutzt **automatisch** die **richtige Methode** des jeweiligen Objekts.
	public class TesteGeoFigur {
	  public static void main(String[] a) {
	  GeoFigur f1 = new **Rechteck**();
	  GeoFigur f2 = new **Quadrat**();
	  GeoFigur f3 = new **Kreis**();
	  GeoFigur figuren = {f1,f2,f3};
	  for(int i=0;i<figuren.length;i++) {
	  sout(**figuren[i].berechneFl()**);
	  }
	  }
	}