Objektorientierte und Formale Programmierung
-- Java Grundlagen --

7. Vererbung und Polymorphie

7. Vererbung und Polymorphie

7.1. Generalisierung in UML und Java

- Generalisierung: Beziehung zwischen einer allgemeineren Klasse (Basisklasse, Oberklasse) und einer spezialisierteren Klasse (Unterklasse)
  - die spezialisierte Klasse ist konsistent mit der Basisklasse, enthält aber zusätzliche Attribute, Operationen und / oder Assoziationen
  - ein Objekt der Unterklasse kann überall da verwendet werden, wo ein Objekt der Oberklasse erlaubt ist
- Nicht nur: Zusammenfassung gemeinsamer Eigenschaften und Verhaltensweisen,
  sondern immer auch: Generalisierung im Wortsinn
  - jedes Objekt der Unterklasse ist ein Objekt der Oberklasse
- Generalisierung führt zu einer Klassenhierarchie

7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

- Modellierung als unabhängige Klassen:

Angestellter
personalnr name adresse geburtsdatum gehalt bankverbindung
druckeAdresse() überweiseGehalt()

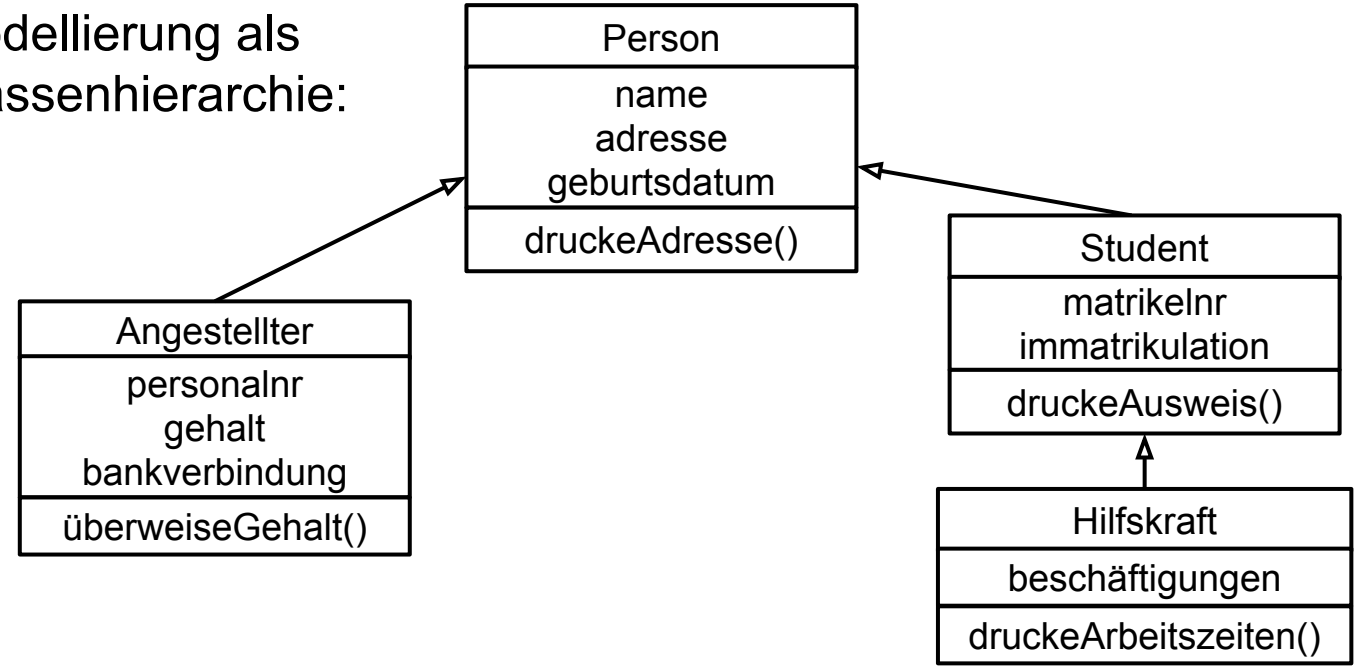
Student
matrikelnr name adresse geburtsdatum immatrikulation
druckeAdresse() druckeAusweis()

Hilfskraft
matrikelnr name adresse geburtsdatum immatrikulation beschäftigungen
druckeAdresse() überweiseGehalt() druckeArbeitszeiten()

7.1. Generalisierung in UML und Java

Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

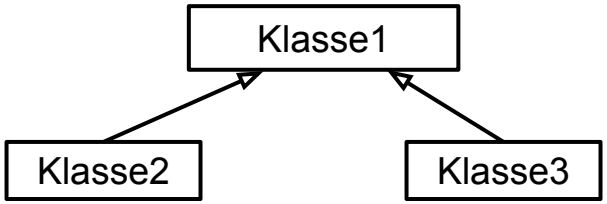
- Modellierung als Klassenhierarchie:

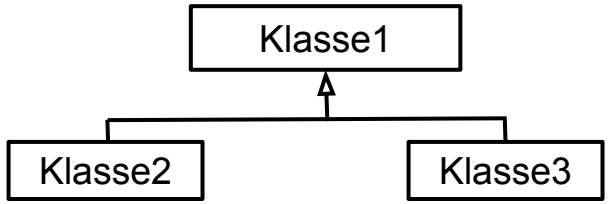


7.1. Generalisierung in UML und Java

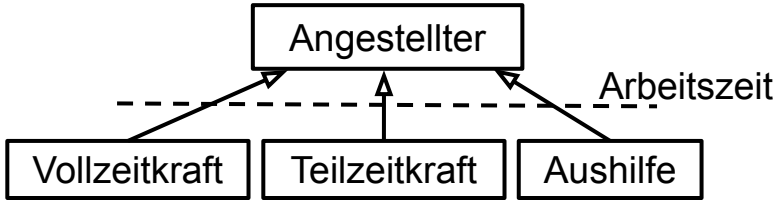
Beispiel Generalisierung: Angestellte, Studenten und studentische Hilfskräfte

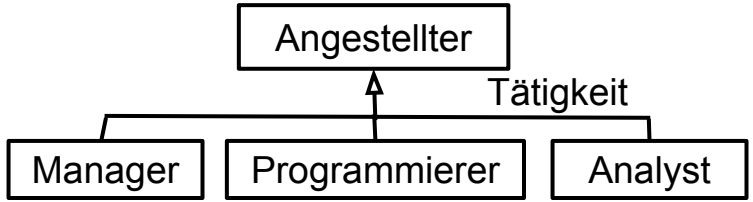
Darstellung:





- Ein zusätzlicher Diskriminator (Generalisierungsmenge) kann das Kriterium angeben, nach dem klassifiziert wird:





7.1. Generalisierung in UML und Java

Abstrakte Klassen und Operationen

- Im vorherigen Beispiel wurden im Modell nur Personen betrachtet, die entweder Angestellter, Student oder Hilfskraft sind
- Die neue Basisklasse ”Person“ wird daher als *abstrakte* Klasse modelliert
- von einer abstrakten Klasse können keine Instanzen (Objekte) erzeugt werden
- Darstellung:
  - Klassenname in Kursivschrift:

    Person

  - Oder für handschriftliche Diagramme:

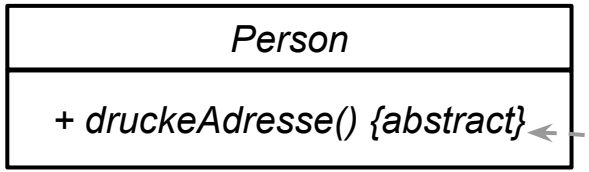
    Person
    {abstract}

```
abstract class Person {
  String name;
  String adresse;
  Date geburtsdatum;
  public void druckeAdresse() {
    //...
  };
}
```

7.1. Generalisierung in UML und Java

Abstrakte Klassen und Operationen

- Eine abstrakte Operation einer Klasse wird von der Klasse nur deklariert, nicht aber implementiert
  - die Klasse legt nur Signatur und Ergebnistyp fest
  - die Implementierung muss in einer Unterklasse durch überschreiben der ererbten Operation erfolgen
- Abstrakte Operationen dürfen nur in abstrakten Klassen auftreten
- Darstellung:



Kursivschrift oder Zusatz {abstract}

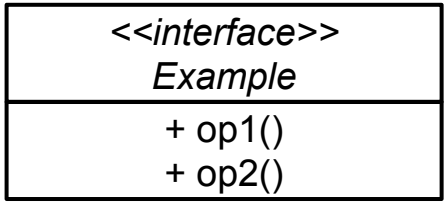
Abstrakte Operationen besitzen keinen Rumpf (d.h. keine Implementierung)

```
abstract class Person {
  String name;
  String adresse;
  Date geburtsdatum;
  public abstract void druckeAdresse();
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Eine (abstrakte) Klasse mit abstrakten Operationen
- Beschreibt eine Menge von Signaturen von Operationen
- Java erlaubt in Schnittstellen nur *öffentliche*, *unveränderliche* und *initialisierte* Klassenattribute: `public static final int M_PI = 3;`,
  `public`, **`static`** und **`final`** können dann auch entfallen: `int M_PI = 3;`
- Die Operationen einer Schnittstelle sind immer abstrakt und öffentlich
  `public` kann in Java auch entfallen
- Darstellung:




```
interface Example {
  public void op1();
  public void op2();
}
```

```
interface Example {
  void op1();
  void op2();
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Schnittstellen definieren ”Dienstleistungen“ für aufrufende Klassen, sagen aber nichts über deren Implementierung aus
- funktionale Abstraktionen, die festlegen was implementiert werden soll, aber nicht wie
- Schnittstellen realisieren damit das Geheimnisprinzip in der stärksten Form:
  - Java-Klassen verhindern über Sichtbarkeiten zwar den Zugriff auf Interna der Klasse, ein Programmierer kann diese aber trotzdem im Java-Code der Klasse lesen
  - der Java-Code einer Schnittstelle enthält nur die öffentlich sichtbaren Definitionen, nicht die Implementierung

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Schnittstellen sind von ihrer Struktur und Verwendung her praktisch identisch mit abstrakten Klassen:
  - sie können wie abstrakte Klassen nicht instanziiert werden
  - Referenzen auf Schnittstellen und auch abstrakte Klassen sind aber möglich, sie können auf Objekte zeigen, die die Schnittstelle implementieren
- Klassen können von Schnittstellen ”erben“
  - ”vererbt“ werden nur die Signaturen der Operationen
  - die Klassen müssen diese Operationen selbst implementieren
  - man spricht in diesem Fall von einer Implementierungs-Beziehung statt von Generalisierung

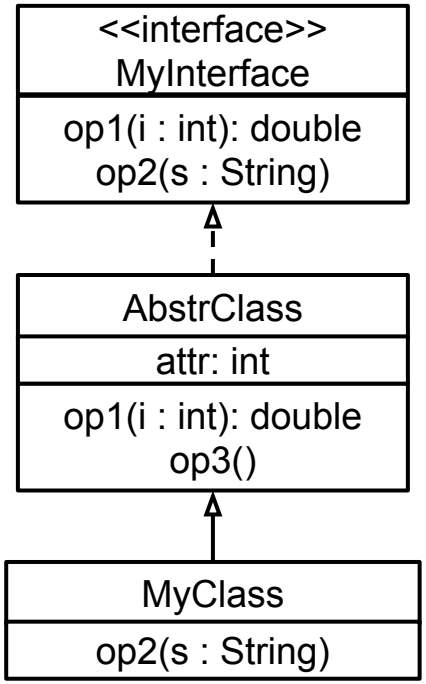
7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen)

- Die Implementierungs-Beziehung **`implements`** besagt, dass eine Klasse die Operationen einer Schnittstelle implementiert
  - die Klasse erbt die abstrakten Operationen, d.h. deren Signaturen incl. Ergebnistyp: Diese müssen dann geeignet überschrieben werden
  - werden nicht alle Operationen überschrieben (d.h. implementiert), so bleibt die erbende Klasse abstrakt
  - die überschreibenden Operationen müssen öffentlich sein
  - die Klasse kann zusätzlich weitere Operationen und Attribute definieren
- Eine Klasse kann mehrere Schnittstellen implementieren (eine Art Mehrfachvererbung, die auch in Java erlaubt is)

7.1. Generalisierung in UML und Java

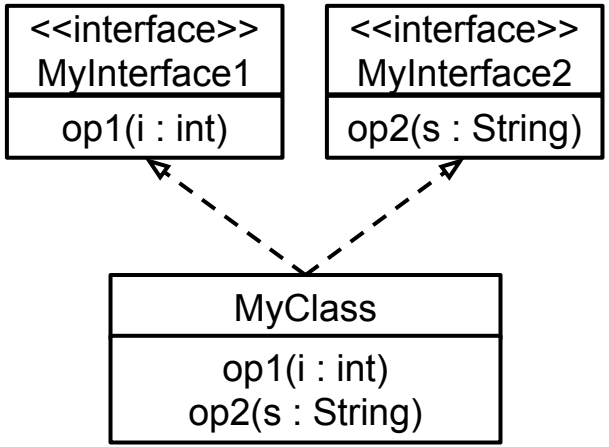
Interfaces (Schnittstellen): Die Implementierungs-Beziehung **implements**




```
interface MyInterface {
  public double op1(int i);
  public void op2(String s);
}

abstract class AbstrClass implements MyInterface {
  protected int attr;
  public double op1(int i) { /* ... */ }
  public void op3() { /* ... */ }
}

class MyClass extends AbstrClass {
  public void op2(String s) { /* ... */ }
}
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Implementierung mehrerer Schnittstellen




```
interface MyInterface1 {
  public void op1(int i);
}

interface MyInterface2 {
  public void op2(String s);
}

class MyClass implements MyInterface1, MyInterface2 {
  public void op1(int i) { /* ... */ }
  public void op2(String s) { /* ... */ }
}
```

7.1. Generalisierung in UML und Java

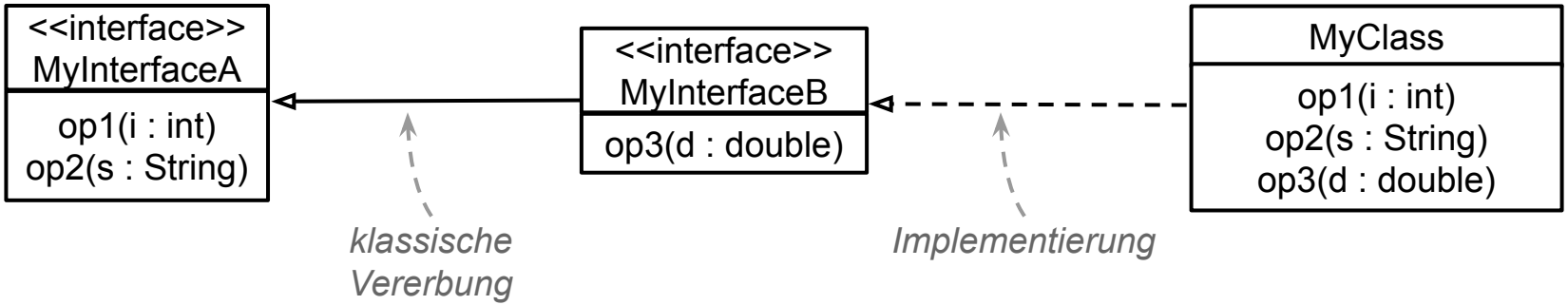
Beispiel Schnittstelle: Interface.java (🌶️🌶️)

```
/** Schreiben Sie ein Java-Programm, um eine Schnittstelle Figur
    mit der Methode getFlaeche() zu erstellen. Erstellen Sie drei
    Klassen, Rechteck, Kreis und Dreieck, die die Figur-
    Schnittstelle implementieren. Implementieren Sie dann die
    Methode getFlaeche() für jede der drei Klassen.
    */
```

7.1. Generalisierung in UML und Java

Interfaces (Schnittstellen): Vererbung

- Schnittstellen können von anderen Schnittstellen erben (analog zu Klassen)
- Darstellung in UML und Java wie bei normaler Vererbung
- Die implementierende Klasse muss die Operationen ”ihrer“ Schnittstelle und aller Ober-Schnittstellen implementieren:



7.1. Generalisierung in UML und Java

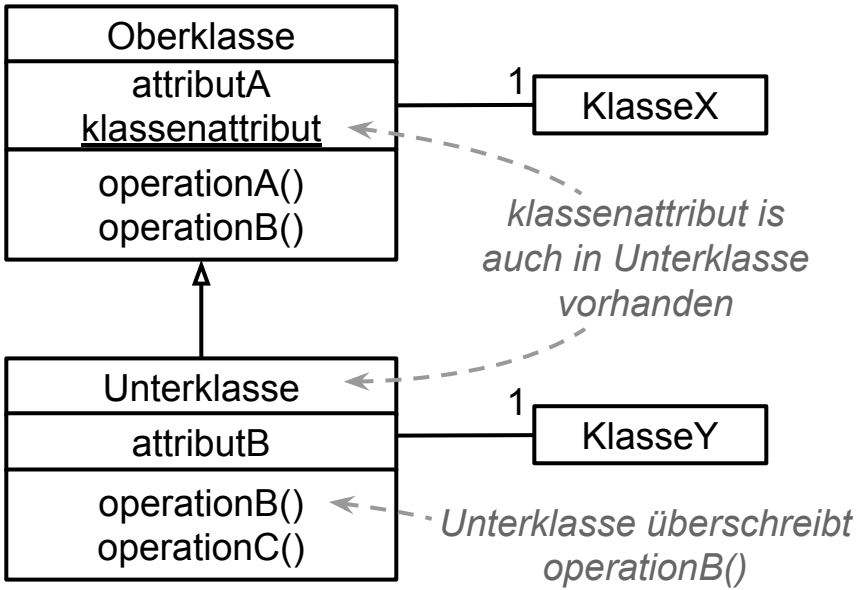
Vererbung

- Eine Unterklasse übernimmt (erbt) von ihren Oberklassen
  - alle *Attribute* und *Klassenattribute*, auch deren Anfangswert wenn definiert
  - alle *Operationen* und *Klassenoperationen*, d.h. alle Operationen einer Oberklasse können auch auf ein Objekt der Unterklasse angewendet werden
  - alle Assoziationen
- Die Unterklasse kann zusätzliche Attribute, Operationen und Assoziationen hinzufügen, aber ererbte *nicht löschen*
- Die Unterklasse kann das Verhalten *neu definieren*, indem sie Operationen der Oberklasse *überschreibt (*d.h. eine Operation gleichen Namens neu definiert)

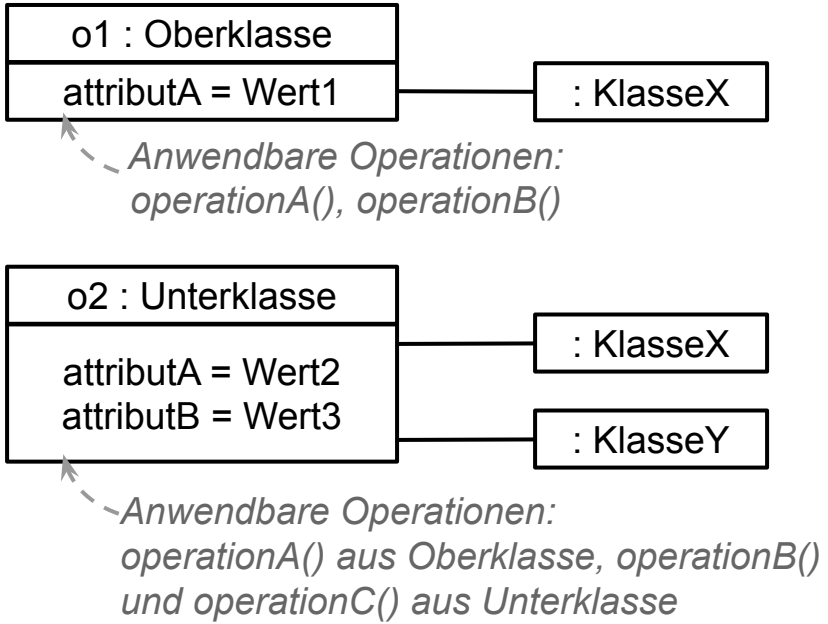
7.1. Generalisierung in UML und Java

Vererbung: Beispiel

Klassendiagramm:



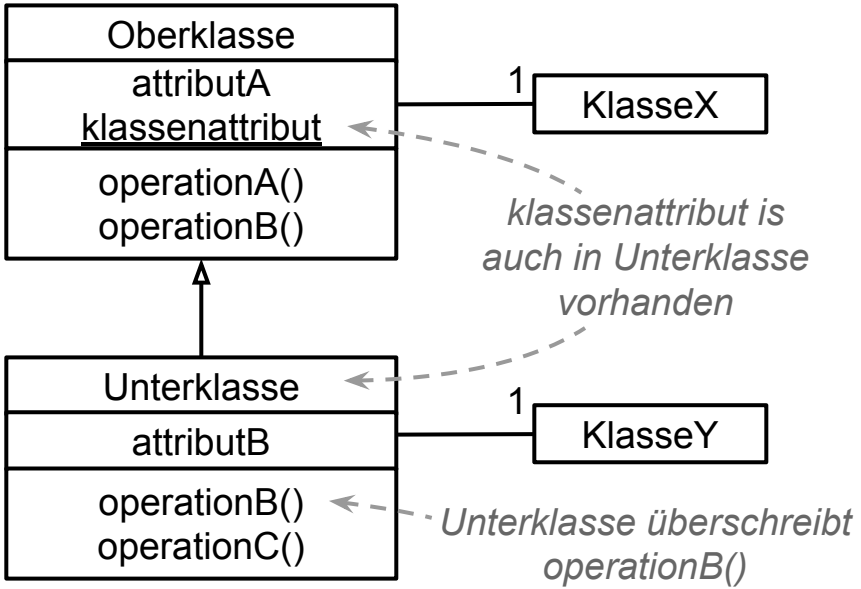
Objektdiagramm:



7.1. Generalisierung in UML und Java

Vererbung: Beispiel

Klassendiagramm:



in Java:

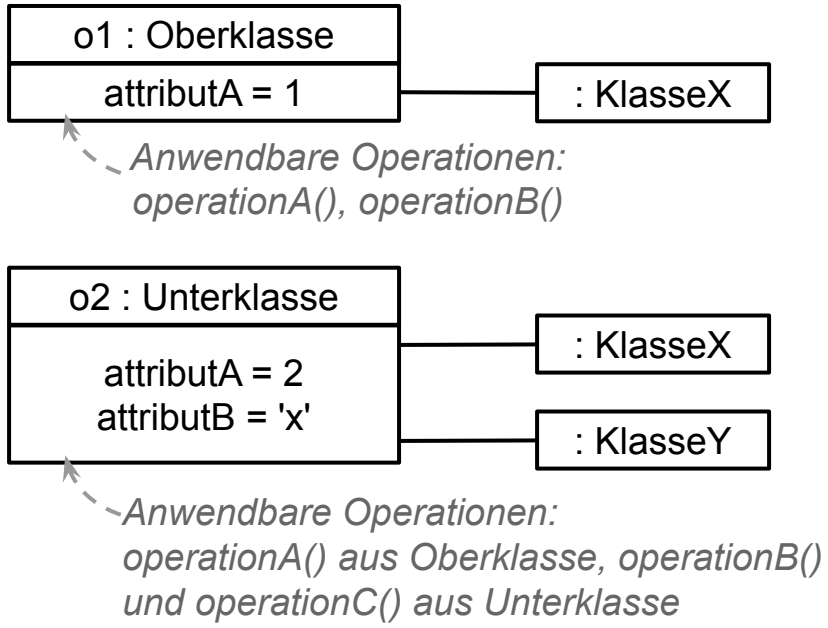
```
class Oberklasse {
     int attributA;
     static double klassenattribut;
     KlasseX kx;
     void operationA() {// ...}
     int operationB() { return 4; }
}

class Unterklasse extends Oberklasse {
     char attributB;
     KlasseY ky;
     int operationB() { return 5; }
     void operationC() {// ...}
}
```

7.1. Generalisierung in UML und Java

Vererbung: Beispiel in Java:

```
KlasseX x = new KlasseX();
KlasseY y = new KlasseY();

Oberklasse.klassenattribut = 1.234;

Oberklasse o1 = new Oberklasse();
o1.attributA = 1; o1.kx = x;
o1.operationA();
o1.operationB();  // returns 4

Unterklasse o2 = new Unterklasse();
o2.attributA = 2; o2.attributB = 'x';
o2.kx = x; o2.ky = y;
o2.operationA(); o2.operationC();
o2.operationB();  // returns 5
```

Objektdiagramm:



7.1. Generalisierung in UML und Java

Vererbung: Überschreiben von Methoden

Beim Überschreiben einer vererbten Methode müssen Signatur und Ergebnistyp *exakt* übereinstimmen:

```
// Richtig:

class Ober {
   void op(int p) { /* ... */ }
}

class Unter extends Ober {
   void op(int p) { /* ... */ }

}
```

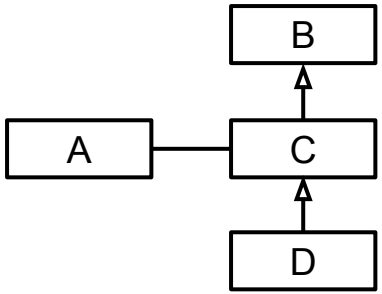
```
// Falsch:

class Unter1 extends Ober {
   // neue Operation (kein Überschreiben):
   void op(double p) { /* ... */ }
}

class Unter1 extends Ober {
   // Fehler:
   int op(int p) { /* ... */ }
}
```

7.1. Generalisierung in UML und Java

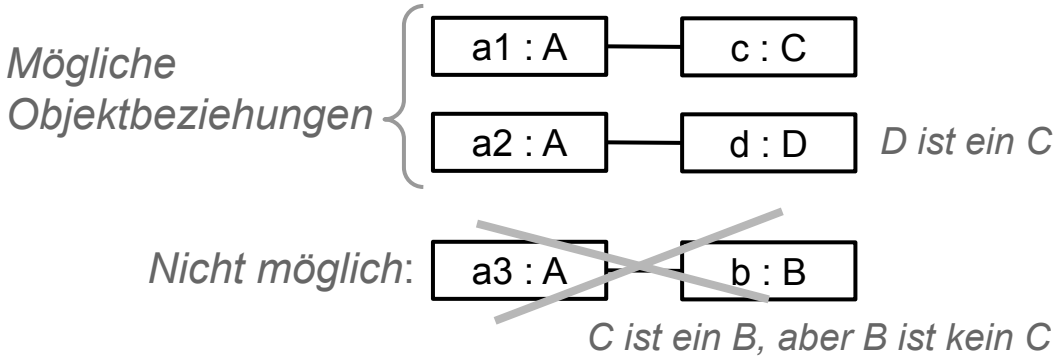
Assoziationen und Generalisierung

- Assoziationsbeziehungen werden ebenfalls vererbt: wenn es eine Assoziation zwischen A und B gibt, kann ein Objekt von A auch mit einem Objekt einer Unterklasse von B in Beziehung stehen
- z.B.:

Klassendiagramm:



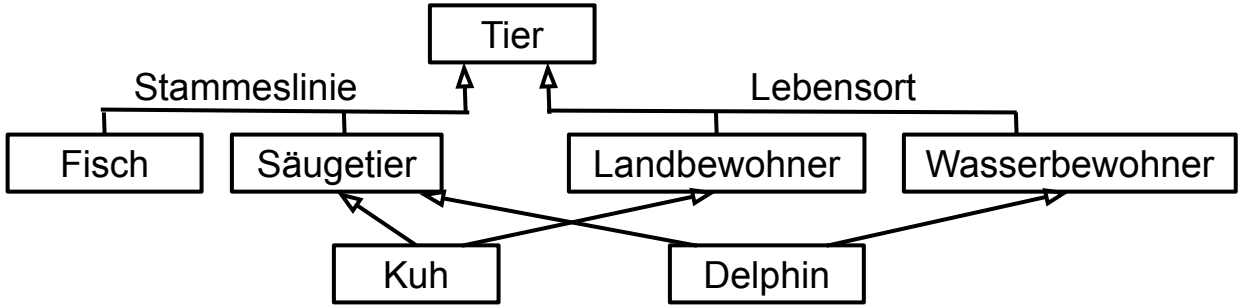
Objektdiagramm:



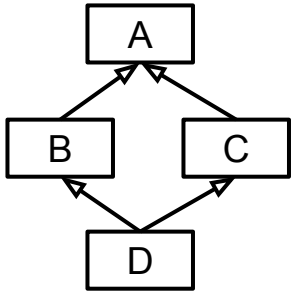
7.1. Generalisierung in UML und Java

Mehrfachvererbung

- Eine Klasse kann in UML auch von mehreren direkten Basisklassen erben:



- Konzept wird nicht von Java unterstützt
  - Probleme, wenn z.B. Oberklassen verschiedene, aber gleichnamige Attribute / Operationen besitzen
- Diamond-Problem entsteht durch Mehrfachvererbung in der objektorientierten Programmierung



7.1. Generalisierung in UML und Java

Diskussion: Vorteile und Probleme

- Bessere Strukturierung des Modell-Universums
- Aufbauend auf vorhandenen Klassen können ähnliche Klassen mit wenig Aufwand erstellt werden
- Einfache Änderbarkeit: Änderung von Attributen / Operationen der Basisklasse wirkt sich automatisch auf die Unterklassen aus
  - dies kann aber auch unerwünscht sein
- Klassen sind schwieriger zu verstehen / zu verwenden
  - auch alle Oberklassen müssen verstanden werden
- Gefahr, überflüssige Klassenhierarchien zu bilden
- Fazit: Generalisierung mit Bedacht verwenden

Beispiel: SnailGame v2 (moodle)

```
class Game extends JPanel implements KeyListener
  private Snail player;
  private Snail[] enemy;
  private boolean bang = false;

  public int xSize = 800;
  public int ySize = 400;

  public Game() { /* ... */ }
  public void keyPressed(KeyEvent e) {...}
  public void keyReleased(KeyEvent e) {}
  public void keyTyped(KeyEvent e) {}
  public void paintComponent( Graphics g ) {...}
  public static void main(String[] a) {...}
}
```

```
interface KeyListener {
  public void keyPressed(KeyEvent e);
  public void keyReleased(KeyEvent e);
  public void keyTyped(KeyEvent e);
}
```

```
public class JPanel extends JComponent
implements Accessible {
  public void paintComponent( Graphics g )
  {...}
}
```

Beispiel: SnailGame v2 (moodle)

```
class Game extends JPanel implements KeyListener {
  private Snail player;
  private Snail[] enemy;
  private boolean bang = false;

  public int xSize = 800;
  public int ySize = 400;

  public Game() { /* ... */ }
  public void keyPressed(KeyEvent e) {...}
  public void keyReleased(KeyEvent e) {}
  public void keyTyped(KeyEvent e) {}
  public void paintComponent( Graphics g )
  public static void main(String[] a) {...}
}
```

```
class Snail {
  private short len = 7;
  private int[][] pos = null;
  public static short segSize = 20;

  public Snail(short len) {/* ... */}
  public Snail(short len, int x, int y) {...}
  public boolean isOverlapped(Snail snail) {...}
  public void moveBody() {/* ... */}
  public void moveHead(int x, int y) {/* ... */}
  public int getLength() {/* ... */}
  public int getPosX(int i) {/* ... */}
  public int getPosY(int i) {/* ... */}
  private int getPosXY(int i) {/* ... */}
}
```

7.2. Konstruktoren und Vererbung

- Die Konstruktoren einer Klasse werden nicht an die Unterklassen vererbt
- In einem Konstruktor kann mittels **`super( [<Parameterliste>] )`** ein Konstruktor der Oberklasse aufgerufen werden:

```
class Shape {
    Shape(int color) { /*...*/ }
    // ...
}
class Circle extends Shape {
    Circle(double[] center, double radius, int color) {
        super(color);   // ruft Konstruktor Shape(int color),
        // ...          // muss erste Anweisung sein!
    }
    // ...
}
```

7.2. Konstruktoren und Vererbung

- Vor der Ausführung eines Unterklassen-Konstruktors wird immer ein Konstruktor der Oberklasse ausgeführt
- falls kein expliziter Aufruf mit **`super`** erfolgt, wird der Default-Konstruktor der Oberklasse ausgeführt:

```
class A {
   A() {
     // Rumpf A ...
   }
}
```

```
class B extends A {
   B(int i) {
     // Rumpf B ...
   }
}
```

```
class C extends B {
   C() { super(9);
     // Rumpf C ...
   }
}
```

- Reihenfolge der *Konstruktoraufrufe* bei new C(): **C() → B(9) → A()**
- Reihenfolge der *Abarbeitung der Konstruktor-Rümpfe*: **A → B → C**

7.2. Konstruktoren und Vererbung

Die ”Referenzvariable“ `super`

- In Instanzmethoden abgeleiteter Klassen gibt es eine spezielle ”Referenzvariable“ `super`
- Sie erlaubt u.a. den Zugriff auf überschriebene Methoden der Basisklasse:

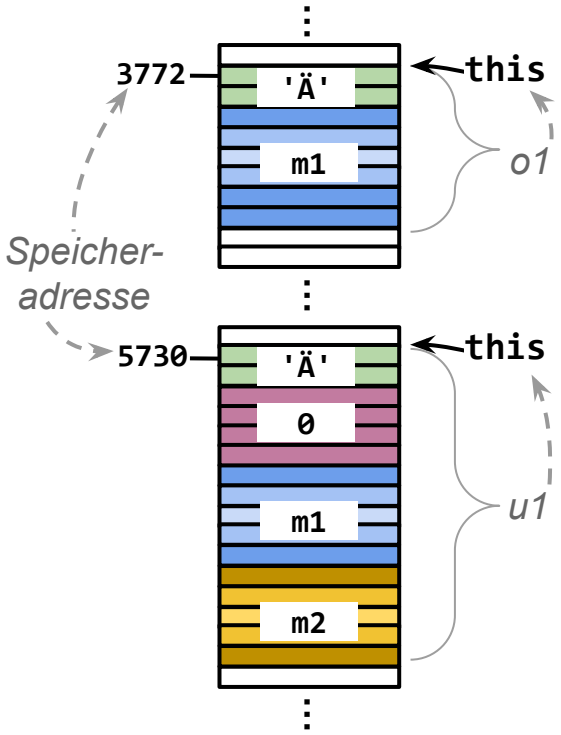
```
class Ober {
    int op(int i) { /* ... */ }
}
class Unter extends Ober {
    int op(int i) { return super.op(i)/2; /* Ruft op(i) in Klasse Ober */}
}
```

- Wie `this` muß auch `super` nicht deklariert werden
- Im Gegensatz zu `this` ist `super` aber keine Referenz auf ein reales Objekt

7.2. Konstruktoren und Vererbung

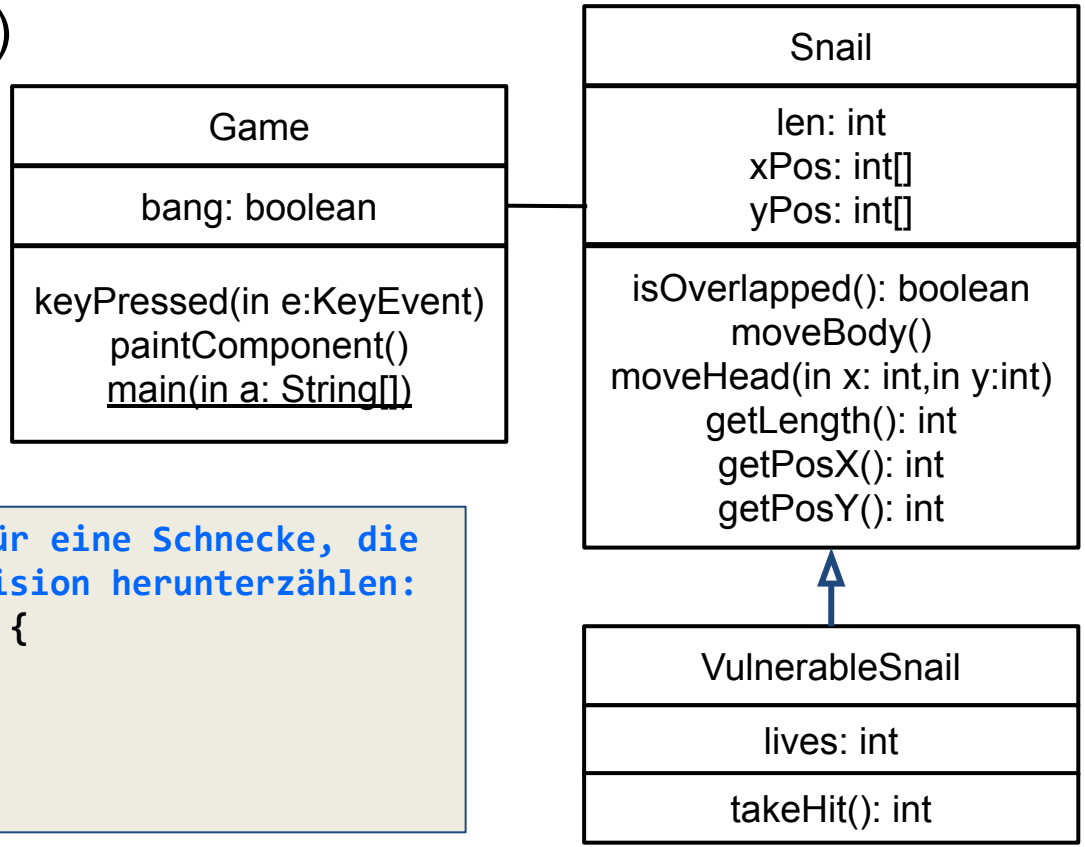
Beispiel **this** und **super**

```
class Ober {
  protected char a1 = 'Ä';
  public boolean m1(char a1) { return this.a1 == a1; }
  public Ober() { a1 = '?'; }
}
class Unter extends Ober {
  private int a2 = 0;
  public boolean m1(char a1) { return a2 == (int) a1; }
  public void m2() { this.a2 = (int) super.a1; }
  public Unter() { super(); a2 = (int) '?'; }
}
// ...
Ober  o1 = new Ober();
Unter u1 = new Unter();
u1.m1('A');
```



7. Vererbung und Polymorphie

Beispiel: SnailGame v2 (github)




```
// Implementiere eine neue Klasse für eine Schnecke, die
// 3 Leben hat, die nach jeder Kollision herunterzählen:
class VulnerableSnail extends Snail {
  private int lives = 3;
  // Konstruktor mit super
  public int takeHit() {/* … */}
}
```

7. Vererbung und Polymorphie

7.3. Polymorphie

- griech.”Vielgestaltigkeit“
- Eigenschaft eines Bezeichners (Operation, Funktion, Variable, ...), je nach Umgebung verschiedene Wirkung zu zeigen
- Verschiedene Arten der Polymorphie:
  - Überladen von Bezeichnern (z.B. ’**+**’ für **`int`**, **`double`**, **`String`**)
  - parametrisierbare Datentypen / Klassen (Typ als Parameter)
  - polymorphe *Funktionen* können Ergebnisse unterschiedlichen Typs liefern
  - polymorphe *Variable* können je nach Umgebung verschiedenartige Größen bezeichnen
- Referenzvariablen sind polymorphe Variablen
  - können auf Objekte unterschiedlicher Klassen verweisen

7.3. Polymorphie

```
import java.util.Random;

abstract class Tier {
   protected String gattung;
   public Tier(String gattung) { this.gattung = gattung; }
   public void print() {
      System.out.println("Tier der Gattung " +  gattung + ".");
   }
}

class Hund extends Tier {
   protected String name, rasse;
   public Hund(String aName, String aRasse) {
      super("Hund"); name = aName; rasse = aRasse; }
   public void print() {
      System.out.println("Ich bin " + name + ", der " + rasse + ".");
   }
}
```

7.3. Polymorphie

```
class Katze extends Tier {
   protected String name;
   public Katze(String n) { super("Katze"); name = n; }
   public void print() {
      System.out.print("Ich bin " + name + ". "); super.print(); }
}
public class DemoPolymorphie {
   public static void main(String[] args) {
     Tier[] tiere = new Tier[3];
     tiere[0] = new Hund("Waldi", "Dackel");  // tiere[] enthaelt verweise
     tiere[1] = new Hund("Hasso", "Boxer");   // auf Objekte verschiedener
     tiere[2] = new Katze("Chanty");          // Klassen (Hund, Katze)
     Random rnd = new Random();
     for (int i=0; i<5; i++) {
        Tier t = tiere[rnd.nextInt(3)]; // t ist eine polymorphe Variable
        t.print();                      // print richtet sich nach dem
     }                                  // Objekt, auf das t verweist
   }
}
```

7.3. Polymorphie

Mögliche Ausgabe des Beispiels:

```
Ich bin Chanty.
Ich bin ein Tier der Gattung Katze.
Ich bin Hasso, der Boxer.
Ich bin Waldi, der Dackel.
Ich bin Chanty.
Ich bin ein Tier der Gattung Katze.
Ich bin Waldi, der Dackel.
```

7.3. Polymorphie

Binden von Bezeichnern

Bezeichner können zu unterschiedlichen Zeiten an Objekte, Datentypen oder Datenstrukturen gebunden werden:

- zur Übersetzungszeit:
  statische Bindung, **frühe Bindung**
- zur Laufzeit:
  dynamische Bindung, **späte Bindung**

```
// statische / fruehe Bindung:
Student t = new Student();
System.out.println( t.getMatrNr() );
```

```
// dymanische / spaete Bindung:
// Array tiere[] enthaelt verweise:
Tier[] tiere = new Tier[3];
tiere[0] = new Hund("Waldi", "Dackel");
tiere[1] = new Katze("Chanti");
// t ist eine polymorphe Variable:
Tier t = tiere[0];
t.print();  // print -> Hund
t = tiere[1];
t.print();  // print -> Katze
```

7.3. Polymorphie

Binden von Bezeichnern

- **Methoden** werden in Java **dynamisch** gebunden
  - abhängig vom Objekt, für das die Methode aufgerufen wird (Polymorphie)
  - Merkregel: jedes Objekt kennt seine Methoden
- **Klassenmethoden** und **Attribute** werden **statisch** gebunden
- abhängig vom Typ der Referenzvariable, die für den Zugriff benutzt wird

7.3. Polymorphie -- Beispiel

```
interface I {
   public int op(int i);
}

class Ober implements I {
   public int attr = 1;
   public static int kattr = 10;
   public int op(int i) { return i+1; }
   public static int kop() { return 4; }
}

class Unter extends Ober {
   public int attr = 2;
   public static int kattr = 20;
   public int op(int i) { return i+2; }
   public int op(double d) { return 7; }
   public static int kop() { return 8; }
}
```

```
I[] i = { new Ober(), new Unter() };
i[0].op(1);     // = 2
i[1].op(1);     // = 3

int z;
Ober o = new Ober();
Unter u = new Unter();

z = i[0].op(3); // = 4
z = i[1].op(4); // = 6
z = o.op(5.1);  // FEHLER
z = u.op(6.2);  // = 7
z = o.attr;     // = 1
z = u.attr;     // = 2
z = o.kop();    // = 4
z = u.kop();    // = 8
z = o.kattr;    // = 10
z = u.kattr;    // = 20
```

7.3. Polymorphie

Konversion von Objekttypen

- Eine neue Klasse definiert in Java auch einen neuen Typ
  - z.B.:
    ```
    Tier einTier;
    Hund fido = new Hund("Fido", "Spaniel");
    ```
    `einTier` und `fido` sind Variablen verschiedenen Typs
- Java ist *typsicher*: Zuweisungen und Operationen sind nur mit Variablen bzw. Ausdrücken des korrekten Typs erlaubt
- Wir können aber:
  ```
  einTier = fido;
  ```
  schreiben, weil Java eine implizite Typkonversion von einer Klasse zu einer Oberklasse (bzw. implementierten Schnittstelle) macht

7.3. Polymorphie

Konversion von Objekttypen

- Explizite Typkonversionen (mit `einTier` und `fido` aus der letzte Folie):

```
einTier = (Tier)fido;           // korrekt, unnötig
fido = (Hund)einTier;           // korrekt
Student s = (Student)einTier;   // Compilerfehler
Katze pucki = (Katze)einTier;   // Laufzeitfehler
```

- Typkonversion bewegt sich in der Klassenhierarchie immer nur
  - aufwärts (implizite oder explizite Konversion), oder
  - abwärts (nur explizite Konversion)
- Test des Objekttyps über Operator `instanceof` möglich:

```
if (einTier instanceof Katze)
  Katze pucki = (Katze)einTier;  // OK, wird nicht ausgefuehrt
```

7.3. Polymorphie

Beispiel -- Verteilung

```
import java.util.Random;

class Verteilung {
  protected int yLen = 24;
  protected int xLen = 80;
  protected int runs = 120;
  protected boolean[][] display = null;
  protected Random r;

  public Verteilung(int height, int width, int runs) {
   r = new Random();
   yLen = height; xLen = width; this.runs = runs;
   display = new boolean[xLen][yLen];
   for (int x = 0; x < xLen; x++)
     for (int y = 0; y < yLen; y++)
       display[x][y] = false;
  }
```

7.3. Polymorphie

Beispiel -- Verteilung

```
  public void print() {
   for (int y = 0; y < yLen; y++) {
     System.out.print("|");
     for (int x = 0; x < xLen; x++)
         System.out.print( (display[x][y] == true)?"\u2589":" ");
     System.out.println("|");
   }
  }
  public static void main(String[] args) {
    Verteilung v;
    if ( (args.length > 0) && (args[0].equals("u")) ) {
      v = new Uniform(24, 100, 240);
    } else {
      v = new Gauss(24, 100, 240);
    }
    v.print();
  }
}
```

7.3. Polymorphie

Beispiel -- Verteilung

```
class Uniform extends Verteilung {
  public Uniform(int height, int width, int runs) {
    super(height, width, runs);
    for (int i = 0; i < runs; i++ ) {
      int c = r.nextInt(height);
      int j = 0;
      while (display[j][c]) { j++; }
      display[j][c] = true;
    }
  }
}
```

7.3. Polymorphie

Beispiel -- Verteilung

```
class Gauss extends Verteilung {
  public Gauss(int height, int width, int runs) {
    super(height, width, runs);
    for (int i = 0; i < runs; i++ ) {
      int c = (int)( yLen/2 + r.nextGaussian() );
      int j = 0;
      while (display[j][c]) { j++; }
      display[j][c] = true;
    }
  }
}
```

7.4. Die universelle Klasse Object

- Java definiert eine Klasse **`Object`**, die an der Spitze jeder Klassenhierarchie steht
- Alle Klassen, die nicht explizit von einer anderen Klasse abgeleitet werden, sind (implizit) von **`Object`** abgeleitet: `public class Tier {...}` heißt damit `public class Tier extends Object { ...}`
- Die Klasse **`Object`** definiert 9 Methoden, die alle Klassen erben (direkt oder indirekt):

```
protected Object clone()
Class<?> getClass()
void notifyAll()
```

```
boolean equals(Object obj)
int hashCode()
void wait()
```

```
protected void finalize()
void notify()
String toString()
```

  - diese Methoden können wie alle anderen auch überschrieben werden

7.4. Die universelle Klasse Object

- **`public String toString()`**
  - gibt eine textuelle Darstellung des **`Object`** zurück
  - Implementierung in **`Object`**: “Klassenname@Adresse”
- **`public boolean equals(Object obj)`**
  - stellt fest, ob zwei Objekte gleich sind
  - Implementierung in **`Object`**: `return this == obj;`
- **`protected Object clone()`**
  - erzeugt eine Kopie des Objekts
  - Implementierung in **`Object`** kopiert alle Instanzvariablen
- **`protected void finalize()`**
  - aufgerufen, bevor *Garbage Collector* **`Object`** löscht
  - Implementierung in **`Object`** tut nichts

```
class MyClass {

  protected void finalize() {
    System.out.println("Bye!");
  }

  public static void main(String[] s) {
    MyClass me = new MyClass();
    MyClass you = new MyClass();

    // test toString:
    System.out.println(me.toString());

    // test equals:
    boolean eq = me.equals(you);
    System.out.println(eq);

    // test finalize:
    me = null;
    you = null;
    System.gc();  // garbage collector
  }
}
```

7.4. Die universelle Klasse Object

- **`System.out.println()`** kann beliebige Objekte ausdrucken, z.B.:

```
class Hund extends Tier {
    // ...
    public String toString() {
        return "Ich bin " + name + ", der " + rasse + ".";
    }
}
```

```
    Hund waldi = new Hund ("Waldi", "Dackel");
    System.out.println(waldi);
```

- weil es so implementiert wurde:

```
public void println(Object obj) {
    String str = obj.toString();  // Polymorphie
    println(str);                 // Aufruf der überladenen Methode
}
```