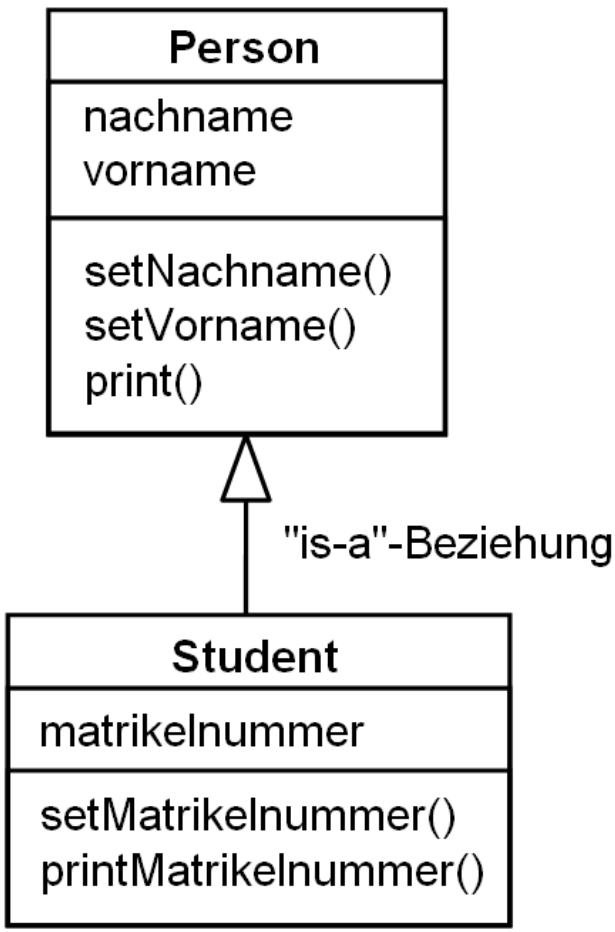
Übersicht

Inhalte

- ✓ Konzept der Vererbung
- Erweitern und Überschreiben
- Besonderheiten bei der Vererbung
- Liskovsche Substitutionsprinzip

Konzept der Vererbung I

- Bei der Vererbung erbt eine Sohnklasse alle Eigenschaften (Datenfelder, Methoden) ihrer Vaterklasse
- Und fügt ihre eigenen individuellen Eigenschaften hinzu

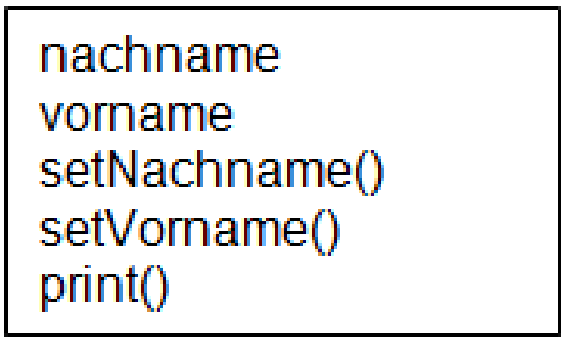


Jedes Objekt der Klasse Student besitzt automatisch alle Instanzvariablen und Methoden, die auch ein Objekt der Klasse Person besitzt

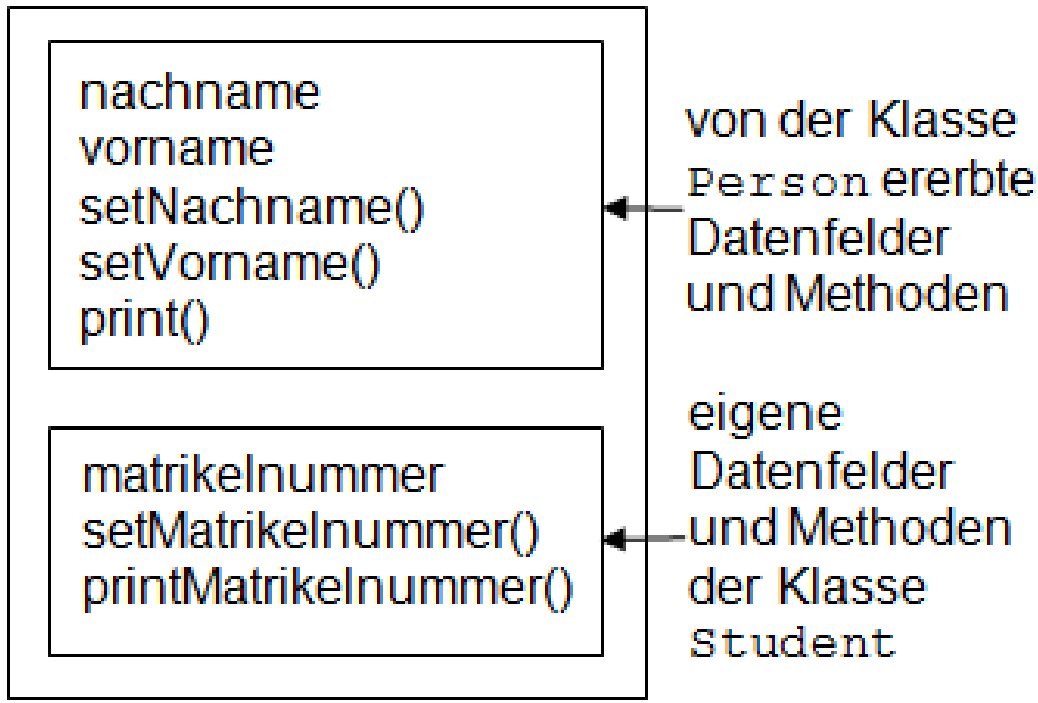
Konzept der Vererbung II

- Konzept der Vererbung

Jedes Objekt der Klasse `Person` hat die Datenfelder und Methoden:

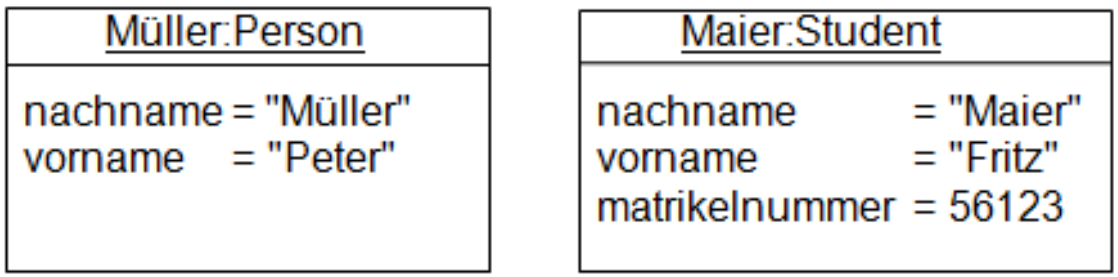


Jedes Objekt der Klasse `Student` hat die Datenfelder und Methoden:



Konzept der Vererbung III

- Objekte der Klasse Person und der Klasse Student



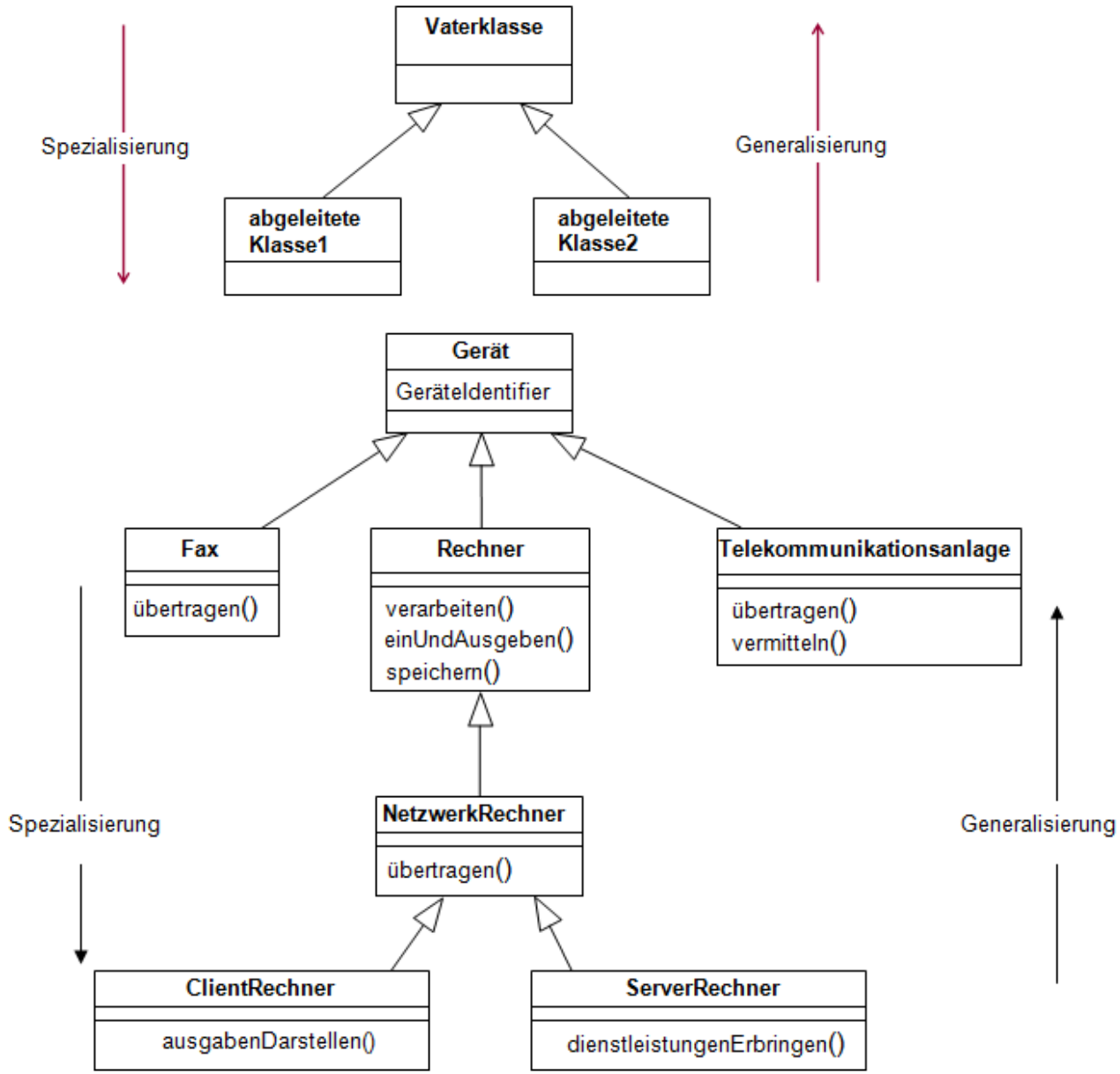
Konzept der Vererbung IV

- Eine erbende Subklasse verwendet das Sprachelement *extends Vaterklasse* hinter dem eigenen Namen

```
public class Student extends Person{  // die Klasse Student wird von
                                      // der Klasse Person abgeleitet
   private int matrikelnummer;

   // Methoden der Klasse Student
   public void setMatrikelnummer (int matrikelnummer){

      this.matrikelnummer = matrikelnummer;
   }

   public void printMatrikelnummer(){

      System.out.println ("Matrikelnummer: " + matrikelnummer);
   }
}
```

Konzept der Vererbung V

- Wegen dem Konzept der Vererbung können Wiederholungen im Entwurf vermieden werden.
- Gemeinsame Eigenschaften mehrerer Klassen werden in gemeinsame Oberklassen ausgelagert, das führt zu mehr Übersicht und zu weniger Wiederholung



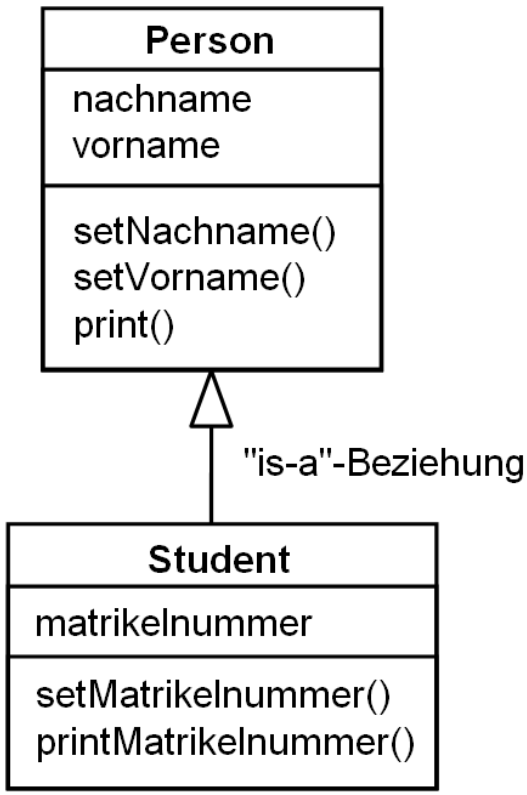
Übersicht

Inhalte

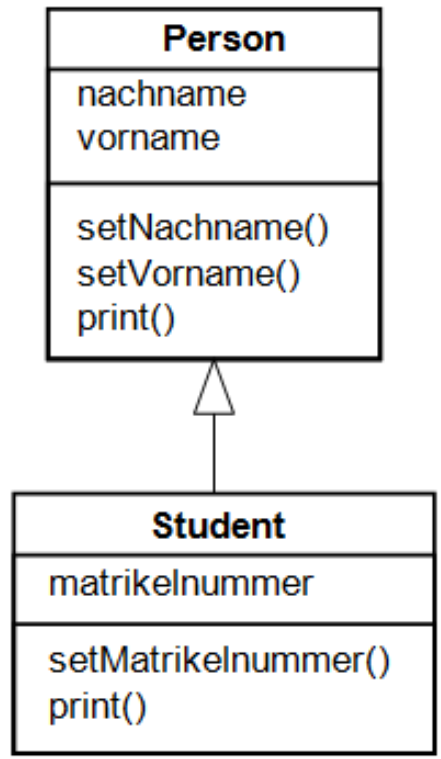
- ✓ Konzept der Vererbung
- ✓ Erweitern und Überschreiben
- Besonderheiten bei der Vererbung
- Liskovsche Substitutionsprinzip

Erweitern und Überschreiben I

- Erweitern und Überschreiben
  - Vererbte Methoden können in der Sohnklasse überschrieben und somit verfeinert und optimiert werden, um eine Spezialisierung zu erreichen



vorher



nachher: mit überschriebener print()-Methode

Erweitern und Überschreiben II

- Erweitern und Überschreiben überschriebene print() Methode

```
public class Student2 extends Person2{
   private int matrikelnummer;

   public Student2 (String vorname, String nachname,int matrikelnummer){

      // Aufruf des Konstruktors der Vaterklasse (später mehr)
      super (vorname, nachname);
      this.matrikelnummer = matrikelnummer;
   }

   // diese Methode gibt es auch in der Vaterklasse,
   // wird hier aber überschrieben
   public void print(){

      System.out.println ("Nachname: " + nachname);
      System.out.println ("Vorname: " + vorname);

      System.out.println ("Matr. Nr: " + matrikelnummer);
   }
}
```

steht so auch in der Vaterklasse

Erweitern und Überschreiben III

- Erweitern und Überschreiben überschriebene print() Methode

```
public class Student2 extends Person2{
   private int matrikelnummer;

   public Student2 (String vorname, String nachname,int matrikelnummer){

      // Aufruf des Konstruktors der Vaterklasse (später mehr)
      super (vorname, nachname);
      this.matrikelnummer = matrikelnummer;
   }

   // diese Methode gibt es auch in der Vaterklasse,
   // wird hier aber überschrieben
   public void print(){

      super.print();
      System.out.println ("Matr. Nr: " + matrikelnummer);
   }
}
```

Erweitern und Überschreiben Aufgaben

Aufgabe 08.01 Vererbungshierarchie für Fahrzeuge

Die Klassen Pkw und Motorrad sollen von der Klasse Fahrzeug abgeleitet werden. In der Klasse FahrzeugTest sollen die Klassen Pkw und Motorrad getestet werden. Das folgende Java-Programm enthält die Klassen Fahrzeug, Pkw, Motorrad und FahrzeugTest. Die fehlenden und zu ergänzenden Teile des Programms sind durch . . . . . gekennzeichnet.

Importieren Sie die Quellcode-Vorlagen Fahrzeug.java, Pkw.java, Motorrad.java, FahrzeugTest.java und Tools.java aus dem OnlineCampus nach Eclipse.

a) Schreiben Sie die Methode getPreis() der Klasse Fahrzeug.

b) Vervollständigen Sie den Konstruktor der Klasse Pkw.

c) Überschreiben Sie in der Klasse Pkw die Methode print() der Klasse Fahrzeug. Die Methode print() der Klasse Pkw soll alle Datenfelder eines Objektes der Klasse Pkw unter Zuhilfenahme der Methode print() der Basisklasse ausgeben. Ergänzen Sie die Methode print() der Klasse Pkw. Ergänzen Sie in analoger Weise die Methode print() der Klasse Motorrad.

d) Ergänzen Sie die fehlenden Teile der Klasse FahrzeugTest.

Erweitern und Überschreiben Aufgaben

Aufgabe 08.02 Vererbungshierarchie für Fertigungsgüter

Ein produzierender Betrieb verwaltet seine hergestellten Produkte zurzeit mit folgenden drei Klassen:

```
public class Membranpumpe
{
private String name;
private int tiefe;
private float maximalerBetriebsdruck;
private int hoehe;
private String membranmaterial;
private int gewicht;
private int maximaleFoerdermenge;
private int breite;
}
```

```
public class Kreiselpumpe
{
private int breite;
private int hoehe;
private int gewicht;
private int anzahlSchaufeln;
private int maximaleFoerdermenge;
private int maximaleDrehzahl;
private String name;
private int tiefe;
private float maximalerBetriebsdruck;
}
```

```
public class Auffangbecken
{
private int tiefe;
private int volumen;
private int breite;
private int gewicht;
private String name;
private int hoehe;
}
```

Entwickeln Sie eine passende Vererbungshierarchie, welche die gemeinsamen Attribute in Basisklassen zusammenfasst.

Übersicht

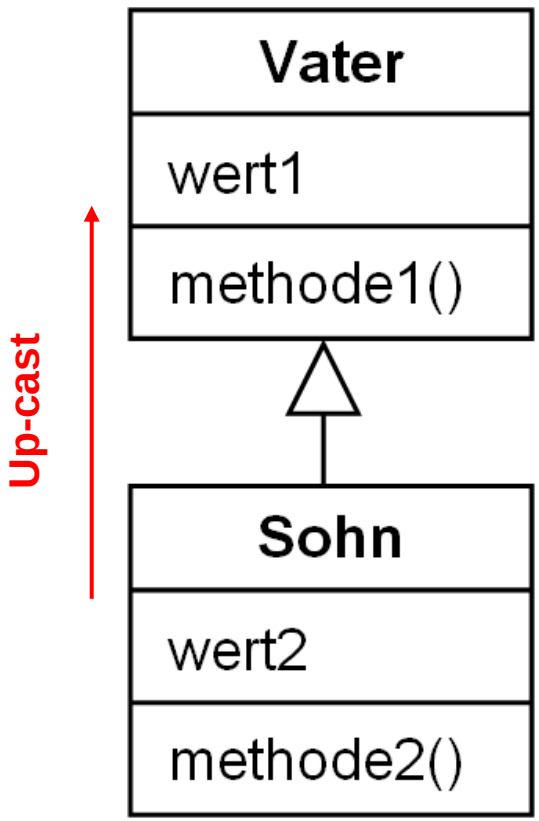
Inhalte

- ✓ Konzept der Vererbung
- ✓ Erweitern und Überschreiben
- ✓ Besonderheiten bei der Vererbung

Liskovsche Substitutionsprinzip

Besonderheiten bei der Vererbung I

- Implizite Typkonvertierung
  - Eine Referenzvariable vom Typ einer Sohnklasse wird einer Referenzvariablen vom Typ einer Vaterklasse zugewiesen

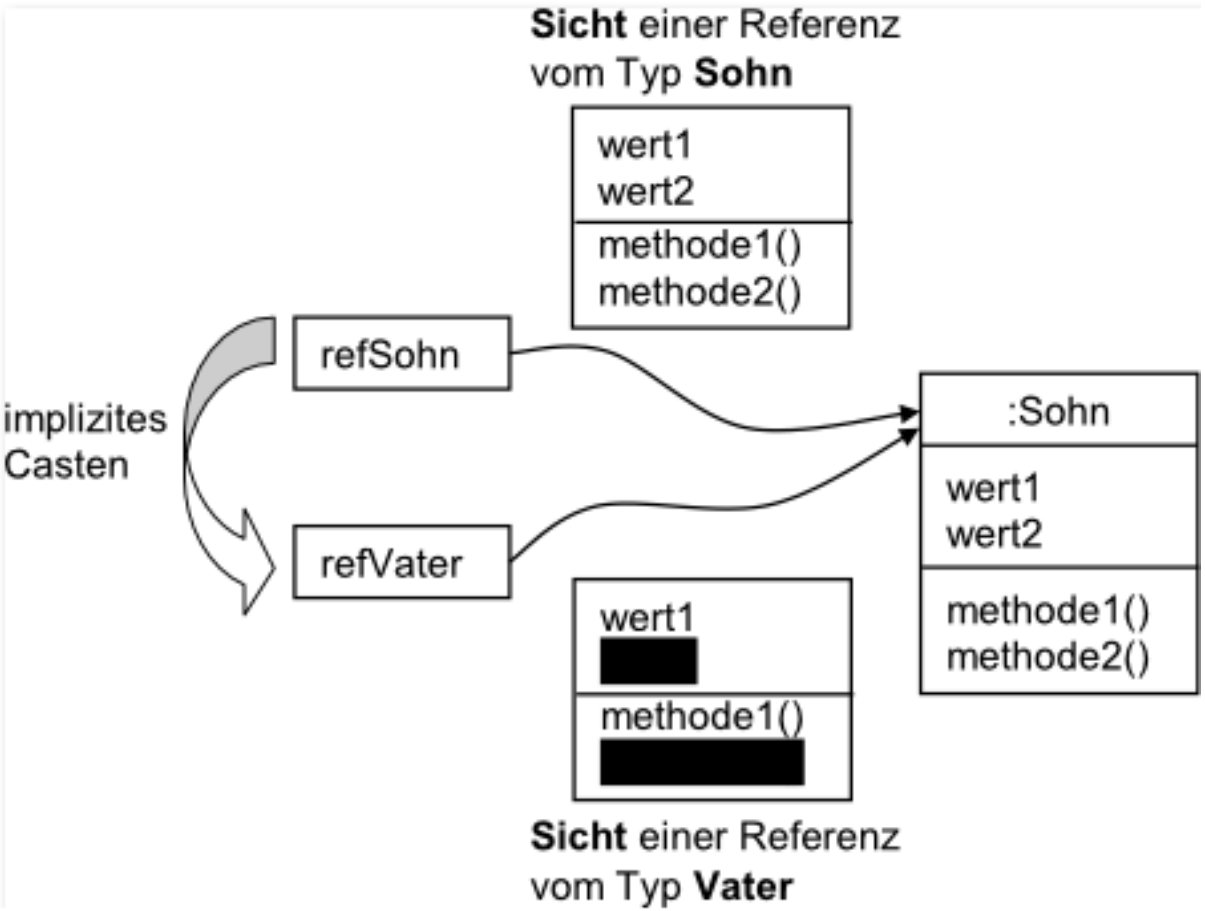



```
Sohn refSohn = new Sohn();

// implizite Typkonvertierung (Up-cast)
Vater refVater = refSohn;
```

Hier wird implizit der cast-Operator (Vater) angewandt.

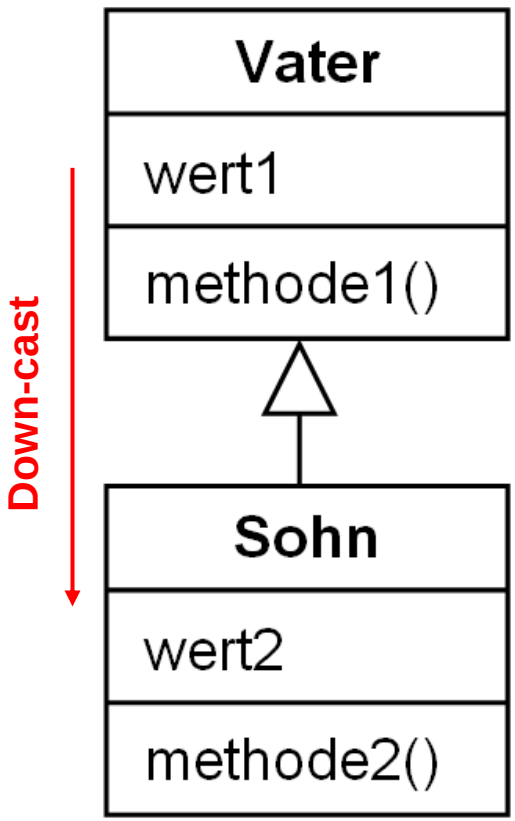
Besonderheiten bei der Vererbung II

- Implizite Typkonvertierung
  - die Referenz vom Typ Sohn sieht das gesamte Objekt
  - die Referenz vom Typ Vater sieht nur die Vateranteile des Sohn-Objektes



Besonderheiten bei der Vererbung III

- Explizite Typkonvertierung
  - Eine Referenzvariable vom Typ einer Vaterklasse wird einer Referenzvariablen vom Typ einer Sohnklasse zugewiesen

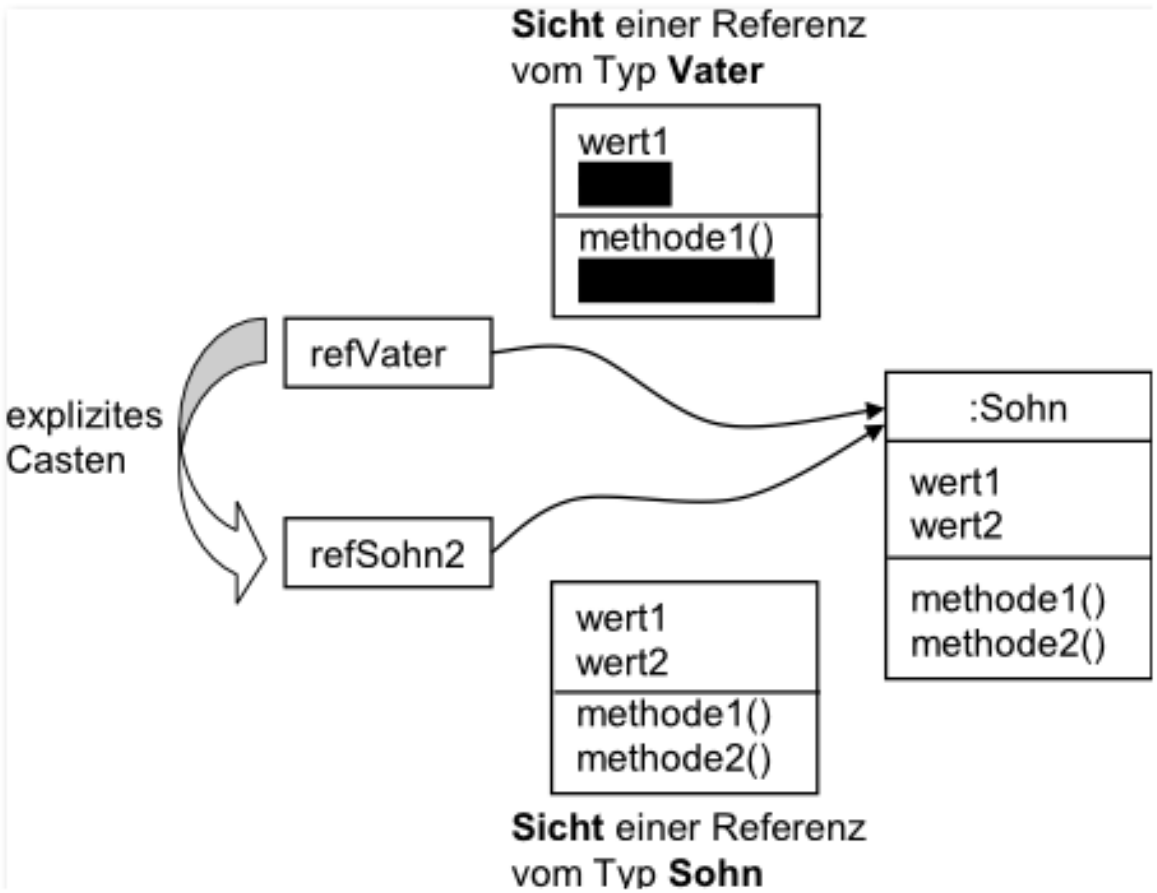



```
Sohn refSohn = new Sohn();
Vater refVater = refSohn;

// explizite Typkonvertierung (Down-cast)
Sohn refSohn2 = (Sohn) refVater;
```

Hier muss explizit der cast-Operator (`Sohn`) angewendet werden.

Besonderheiten bei der Vererbung IV

- Explizite Typkonvertierung
  - verdeckte Eigenschaften werden durch das Down-casten der Referenz wieder sichtbar
  - Down-cast erfordert immer die explizite Angabe des cast-Operators



Besonderheiten bei der Vererbung V

- Konstruktoren bei abgeleiteten Klassen Beispiel:

```
public class Person4
{
    private String name;
    private String vorname;

    public Person4 (String name, String vorname)
    {
       System.out.println ("Konstruktoraufruf von Person4");
       this.name = name;
       this.vorname = vorname;
    }
}

public class Student4 extends Person4
{
    private int matrikelnummer;

    public Student4 (String name, String vorname, int m)
    {
       super (name, vorname); // Aufruf des Konstruktors der
                              // Superklasse
       System.out.println ("Konstruktoraufruf von Student4");
       matrikelnummer = m;
    }
}
```

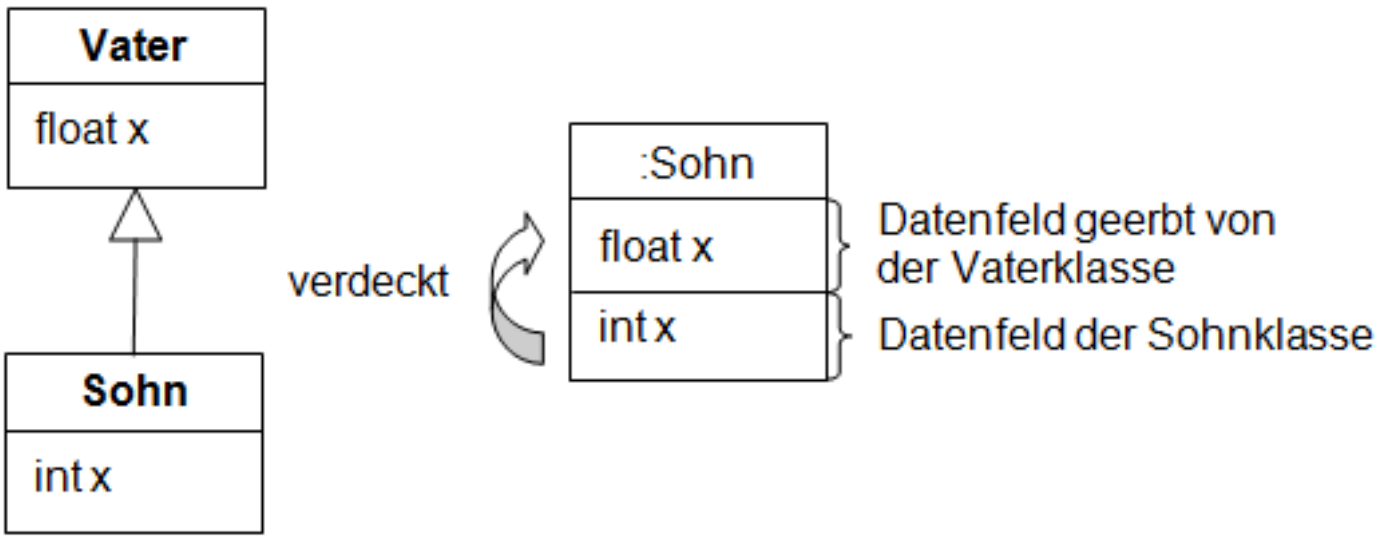
```
public class Test4
{
    public static void main (String[] args)
    {
       Person4 p = new Person4 ("Müller", "Peter");
       Student4 s = new Student4 ("Brang", "Rainer", 666666);
    }
}
```

Besonderheiten bei der Vererbung VI

- Konstruktoren bei abgeleiteten Klassen
  - Aufruf mittels *super()* im Konstruktor.
  - Der super() -Aufruf muss immer in der *ersten Zeile des Konstruktors* stehen. Lässt man ihn weg, so fügt Java implizit einen Aufruf des Default-Konstruktors der Superklasse ein.
  - Der voreingestellte Default-Konstruktor der Klasse *Object* hat einen leeren Rumpf, d.h. er tut nichts. Ein voreingestellter Default-Konstruktor einer anderen Klasse ruft automatisch den parameterlosen Konstruktor der Vaterklasse auf.
  - Ein Konstruktor der Basisklasse wird immer vor der Initialisierungsanweisung des Konstruktors der abgeleiteten Klasse ausgeführt.
  - Ein Default-Konstruktor hat keine Parameter. Der Programmierer kann selbst einen Konstruktor ohne Parameter schreiben. Dann wird vom Compiler dieser selbst geschriebene Default-Konstruktor und nicht der vom Compiler zur Verfügung gestellte Default-Konstruktor aufgerufen.
  - Ein formaler Parameter des Konstruktors der abgeleiteten Klasse kann als aktueller Parameter an den Konstruktor der Basisklasse übergeben werden. Beispiel folgt.

Verdecken I

- Verdecken von Datenfeldern
  - Bei Namensgleichheit in Sohn- und Vaterklasse



Verdecken II

- Zugriff auf verdeckte Instanzvariable der Vaterklasse

```
class Vater{
   int x = 2;
}

class Sohn extends Vater{
   int x = 1;

   public Sohn(){
      System.out.println ("x des Sohnes: " + x);
      System.out.println ("x des Sohnes: " + this.x);
      System.out.println ("vom Vater geerbtes x: " + super.x);
      System.out.println ("vom Vater geerbtes x: "+((Vater)this).x);
   }
}

public class VaterSohnTest{
   public static void main (String[] args){
      Sohn s = new Sohn();
   }
}
```

Finale Methoden und finale Klassen I

- Finale Methoden können in einer Subklasse nicht überschrieben werden.
- Finale Klassen sind Klassen, von denen man keine weiteren Klassen ableiten kann. Damit kann man nur die Benutzung von Klassen, aber nicht die Ableitung erlauben.

```
public final class Konstanten{

   private Konstanten(){        // Von der Klasse können keine
                                // Objekte erzeugt werden.
   }

   public static final float PI = 3.141f;
   public static final int MAX = 255;
}
```

Abstrakte Basisklassen I

- nicht immer soll eine Klasse sofort ausprogrammiert werden
  - z.B. dann nicht, wenn die Oberklasse lediglich Methoden für die Unterklassen vorgeben möchte, aber nicht weiß, wie sie diese implementieren soll
- In Java gibt es dafür zwei Konzepte
  - abstrakte Klassen
  - Schnittstellen (Interfaces → siehe Kapitel 11)
- abstrakte Klassen können selbst nicht instantiiert werden
  - sie enthalten jedoch Vorgaben für die abgeleiteten Unterklassen
  - sie können Variablen, Konstanten, implementierte und abstrakte Methoden enthalten
  - abstrakte Methoden enthalten keinen Methodenrumpf (nur die Signatur)
  - abstrakte Methoden müssen dann von einer Unterklasse implementiert werden

Abstrakte Basisklassen II

- Beispiel: 2d Geometrische Figur
  - besitzt bestimmte Eigenschaften: z.B. Koordinaten x und y, Flächeninhalt
  - es macht aber keinen Sinn eine Instanz von „geometrischer Figur“ zu erzeugen, da es so etwas direkt nicht gibt
  - es gibt aber die Unterklassen wie Rechteck, Dreieck, Quadrat etc., deren gemeinsamen Eigenschaften in der abstrakten Oberklasse geometrische Figur zusammengefasst sind

```
public abstract class GeometrischeFigur2d{

   protected int x = 0;
   protected int y = 0;

   public abstract int berechneFlaeche();
}
```

```
public class Quadrat extends
GeometrischeFigur2d{

   …
   public int berechneFlaeche(){
       return laenge*laenge;
   }
}
```

Demo in Eclipse:
GeometrischeFigur2d.java
und Quadrat.java

Aufgabe

Aufgabe 08.04 Flächen- und Umfangsberechnung

In dieser Übung sollen die beiden Klassen Kreis und Rechteck implementiert werden. Hierzu leiten beide Klassen von der abstrakten Basisklasse GeometrischeFigur ab und werden mit Hilfe der Klasse TestBerechnung getestet. Die beiden Klassen haben die Aufgabe, die Fläche und den Umfang eines Kreises bzw. Rechtecks zu berechnen.

- Kreis — Umfang: $2 \cdot \pi \cdot r$ — Fläche: $\pi \cdot r \cdot r$
- Rechteck — Umfang: $2 \cdot a + 2 \cdot b$ — Fläche: $a \cdot b$

- Eine Konstante für die Zahl π ist in der Klasse java.lang.Math definiert.
- Importieren Sie für die Bearbeitung die beiden Klassen GeometrischeFigur.java und TestBerechnung.java aus dem Online-Campus nach Eclipse.

Übersicht

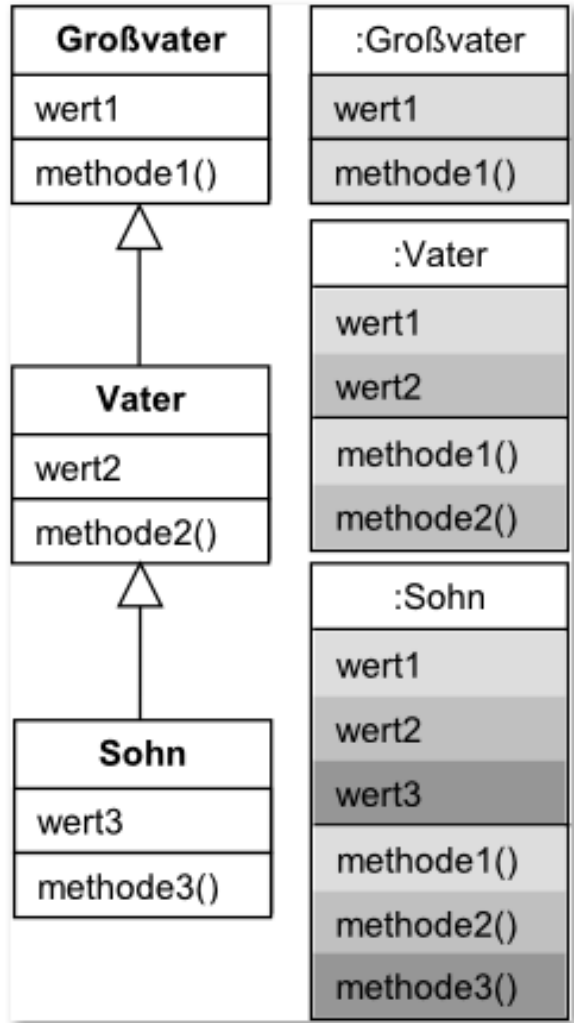
Inhalte

- ✓ Konzept der Vererbung
- ✓ Erweitern und Überschreiben
- ✓ Besonderheiten bei der Vererbung
- ✓ Liskovsche Substitutionsprinzip

Polymorphes Verhalten I

- **Liskovsches Substitutionsprinzip**: ein Objekt einer abgeleiteten Klasse kann an die Stelle eines Objektes einer Basisklasse treten (siehe Up-cast)

- Polymorphie erlaubt es, generalisierten Code für Basisklassen zu schreiben, der dann später von Objekten beliebiger abgeleiteter Klassen benutzt werden kann.



Polymorphes Verhalten II

- Beispiel:

```
public class Ware{
   protected int nummer;
   protected String name;
   protected float preis;
   protected static int aktuelleNummer = 0;

   public Ware (String name, float preis){
      nummer = aktuelleNummer++;
      this.name = name;
      this.preis = preis;
   }

   public int getNummer(){
      return nummer;
   }

   public void print(){
      System.out.print ("ID: " + nummer + "  Bezeichnung: " + name +
                        "  Preis: " + preis);
   }
}
```

Polymorphes Verhalten III

- Beispiel:

```
public class WarenLager{
   protected Ware[] arr;

   public WarenLager (int max){
      arr = new Ware[max];
   }
   public int getNummer(){
      return nummer;
   }
   // es kann jede Art von Ware hinzugefügt werden (Vorteil der Polymorphie)
   public void aufnehmen(Ware neueWare){
      for(int i=0; i<arr.length; i++){

          if(arr[i]==null){  // erstes freies Feld gefunden
            arr[i] = neueWare;
            break;
          }
      }
   }
}
```

Polymorphes Verhalten IV

- Beispiel: jetzt kann man z.B. Milch, Brot etc. in Verbindung mit dem Warenlager nutzen

```
public class Milch extends Ware{
   private float fettGehalt;

   public Milch (float fett){
       super("Milch",0.79);
       this.fettGehalt = fett;
   }
}
```

```
…
WarenLager lager = new WarenLager(4);
Milch milch = new Milch(1.5);

// jede Art von Ware kann nun hinzugefügt
werden
lager.aufnehmen(milch);
…
```

Aufgabe

- **Aufgabe 08.05** Polymorphie - Portorechner

  Implementieren Sie eine abstrakte Basisklasse Versandgut mit den int-Attributen gewicht, laenge, breite und hoehe sowie den zugehörigen get-Methoden.
- Implementieren Sie zwei weitere Klassen Brief und Paket, welche von Versandgut erben. Ein Brief besitzt zusätzlich noch das boolean-Attribut einschreiben und das Paket ein boolean-Attribut zerbrechlich. Diese abgeleiteten Klassen erhalten einen Konstruktor, der sämtliche Attribute per Übergabeparameter initialisiert.
- Vervollständigen Sie nun die Klasse Portorechner aus dem Online-Campus. Legen Sie dazu jeweils ein Objekt Brief und Paket in main an und rufen Sie für beide die zu implementierende Methode berechnePorto(Versandgut v) auf.

Es gibt drei Kategorien von Porto:

1) längste + kürzeste Seite < 30 cm und Gewicht < 100 g, dann Porto = 1.50 Euro

2) längste + kürzeste Seite zwischen 30 und 79 cm und Gewicht < 5.000 g, dann Porto = 4,80 Euro

3) sonst Porto = 16,40 Euro

Polymorphes Verhalten V

- Instanceof – Operator: mit dem instanceof-Operator kann getestet werden, ob eine Referenz auf ein Objekt eines bestimmten Typs zeigt
  - z.B. wenn eine Referenz vom Typ einer Basisklasse ist

```
// Boolescher Ausdruck
a instanceof Klassenname
```

- Beispiel
  - alle Klassen sind von der Klasse Object abgeleitet, deshalb ist immer ein Up-cast in einen Referenztyp der Klasse Object möglich

```
Object refA = new Grossvater();
Object refB = new Vater();
Object refC = new Sohn();

// alle folgenden Ausdrücke geben true zurück
refA instanceof Grossvater
refB instanceof Grossvater
refC instanceof Grossvater
```

Demo in Eclipse:
InstanceOfTest.java