Überblick

Vererbung

- Vererbung ist die Umsetzung von is-a-Beziehungen zwischen zwei (oder mehreren) Klassen
- Die Vaterklasse A ist eine Generalisierung der abgeleiteten Klasse B, B entsprechend eine Spezialisierung von A
- Das bedeutet, B hat alle Eigenschaften von A und darüberhinaus möglicherweise noch mehr

Vererbung

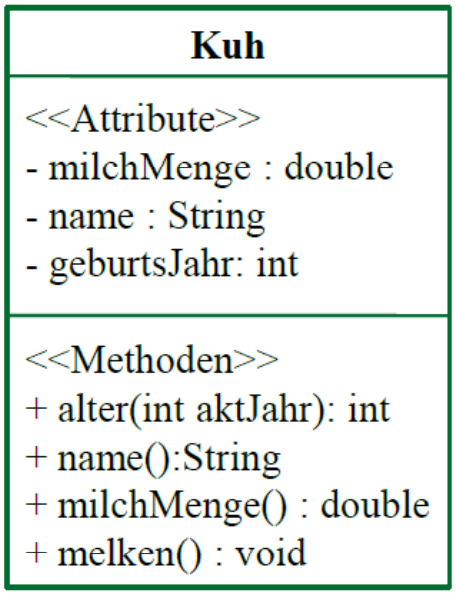
- Alle Eigenschaften/Elemente (Attribute und Methoden) der Vaterklasse A sollen also auf die abgeleitete Klasse B *vererbt* werden, d.h. sie sollen in B nicht mehr extra aufgeführt werden müssen
- Zusätzlich kann die abgeleitete Klasse B neue Elemente (Attribute und Methoden) definieren bzw. ein bereits existierendes Element verfeinern (*überschreiben*)
- Vererbung ist ein wichtiges Mittel zur Wiederverwendung von Programmteilen:
  Funktionalitäten, die in der Vaterklasse A bereits implementiert sind, werden automatisch auf die abgeleitete Klasse B vererbt, müssen also in B nicht noch einmal implementiert werden (außer, die Funktionalität wird redefiniert/überschrieben)

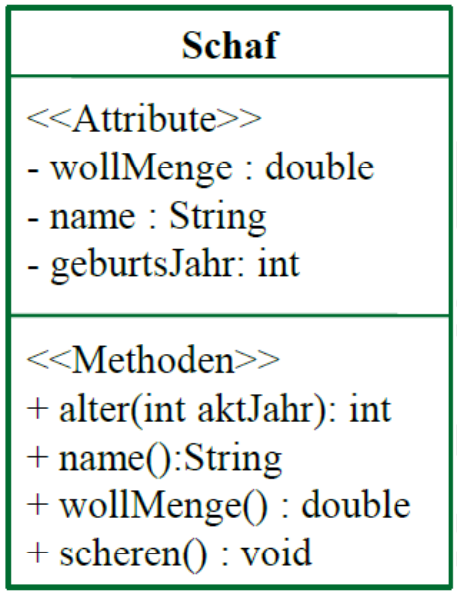
Beispiel

- Wir wollen das Tierreich (bestehend aus Schafen und Kühen) auf einem Bauernhof modellieren.
- Dazu stellen wir fest, dass beide Tierarten gemeinsame Eigenschaften haben, z.B. einen Namen, ein Geburtsjahr, etc.
- Zusätzlich haben beide Tierarten unterschiedliche Eigenschaften, bei den Kühen interessieren wir uns z.B. für die Milchmenge, die sie letztes Jahr abgegeben haben, bei den Schafen interessiert uns die Menge der Wolle, wir scheren konnten, etc.
- Dies kann man mit Vererbung entsprechend modellieren (siehe übernächste Folie).

Beispiel

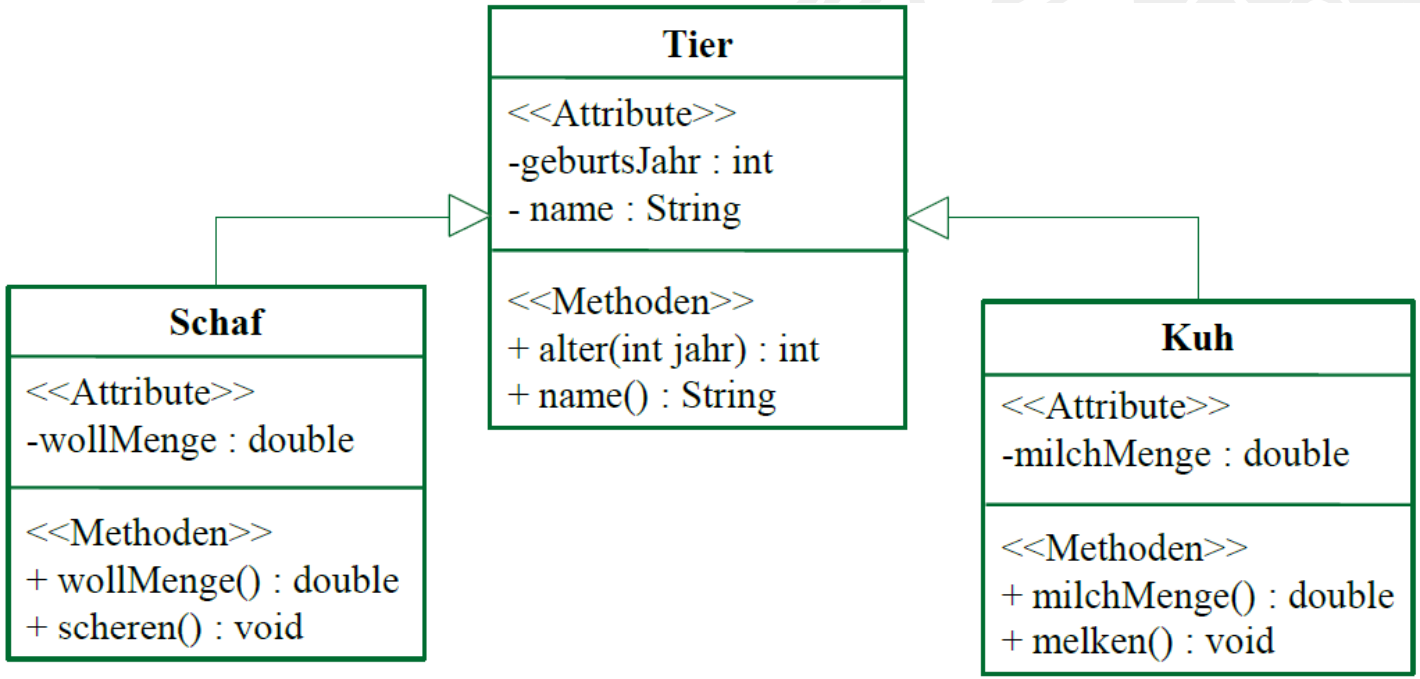
Zunächst ohne Vererbung:





Beispiel

Jetzt mit Vererbung:
Die gemeinsame Eigenschaften (Attribute) und Funktionalitäten (Methoden) sind in eine gemeinsamen Oberklasse ausgelagert



Vorteile

- Die Verebungsbeziehung hat die gewünschte Semantik: alle Elemente der Vaterklasse (hier *Tier*) wird auf die von ihr abgeleiteten Klassen (hier *Kuh* und *Schaf*) *automatisch* vererbt, d.h. diese stehen in den abgeleiteten Klassen zur Verfügung, ohne dass sie explizit noch einmal angegeben werden müssen
- Da die Elemente der Vaterklasse auf die abgeleiteten Klassen vererbt werden, müssen sie nur einmal (in der Vaterklasse) implementiert werden (Wiederverwendung von Code, dadurch u.a. Vermeidung von Fehlern)
- Im Beispiel: Da die Klasse *Tier* das Attribut *name* besitzt, hat auch die Klasse *Kuh* dieses Attribut; entsprechendes gilt für Objekte der Klasse *Kuh*

Ableiten einer Klasse in Java

- In Java wird nur die einfache Vererbung (eine Klasse wird von genau einer Vaterklasse abgeleitet) direkt unterstützt
- Mehrfachvererbung (eine Klasse wird von mehr als einer Vaterklasse abgeleitet) muss in Java mit Hilfe von *Interfaces* umgesetzt werden (dazu später mehr)
- In Java zeigt des Schlüsselwort **`extends`** Vererbung an

Ableiten einer Klasse in Java

- Beispiel: Vaterklasse

```
public class Tier {
   private int geburtsJahr;
   private String name;

   public int alter(int jahr) { ... }

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

Ableiten einer Klasse in Java

- Beispiel: Abgeleitete Klasse

```
public class Kuh extends Tier {
   private double milchMenge;

   public double milchMenge() { ... }

   public void melken() { ... }
}
```

- Objekte der Klasse `Kuh` erben damit alle Attribute und Methoden der Klasse `Tier`

Ableiten einer Klasse in Java

- Was bedeutet das?
- Ein Objekt der Klasse `Kuh` kann z.B. nun auch die Methode `alter` ausführen, da diese Methode aus der Vaterklasse vererbt wird:

```
...
Kuh resi = new Kuh();
...
int aktuellesAlter = resi.alter(2015);
...
```

  Analoges gilt natürlich für die Methode `name`
- Resi (und alle anderen Kühe) hat also auch zusätzlich zum Attribut `milchMenge` die Attribute `geburtsjahr` und `alter`

Die Klasse `Object`

- Enthält eine Klasse keine **`extends`**-Klausel, so besitzt sie die implizite Vaterklasse `Object` (im Paket `java.lang`)
- Also: Jede Klasse, die keine **`extends`**-Klausel enthält, wird direkt von `Object` abgeleitet
- Jede explizit abgeleitete Klasse ist am oberen Ende ihrer Vererbungshierarchie von einer Klasse ohne explizite Vaterklasse abgeleitet und ist damit ebenfalls von `Object` abgeleitet
- Damit ist `Object` *die Vaterklasse aller anderen Klassen*

Die Klasse `Object`

- Die Klasse `Object` definiert einige elementare Methoden, die für alle Arten von Objekten nützlich sind, u.a.:
  - **`boolean`** `equals(Object obj)`

    testet die Gleichheit zweier Objekte, d.h. ob zwei Objekte den gleichen Zustand haben
  - `Object clone()`

    kopiert ein Objekt, d.h. legt eine neues Objekt an, das eine genaue Kopie des ursprünglichen Objekts ist
  - `String toString()`

    erzeugt eine `String`-Repräsentation des Objekts
  - etc.
- Damit diese Methoden in abgeleiteten Klassen sinnvoll funktionieren, müssen sie bei Bedarf überschrieben werden

Vererbung von Attributen und Methoden

- Neben ererbten Attributen und Methoden dürfen neue Attribute und Methoden in der abgeleiteten Klasse definiert werden
- Es dürfen aber auch Attribute und Methoden, die von der Vaterklasse geerbt wurden, neu definiert werden
- Bei Attributen tritt dabei der Effekt der *Versteckens* auf: Das Attribut der Vaterklasse ist in der abgeleiteten Klasse nicht mehr sichtbar
- Bei Methoden tritt zusätzlich der Effekt des *Überschreibens* (auch: *Überlagerns*) auf: Die Methode modelliert in der abgeleiteten Klasse i.d.R. ein anderes Verhalten als in der Vaterklasse

Beispiel: Überlagern

- Der Zugriff auf den Namen könnte in `Tier` wie folgt implementiert sein:

```
public class Tier {
    ...
    public String name() {
        return "Der Name des Tiers ist "+this.name;
    }
}
```

- Und in `Kuh` dagegen:

```
public class Kuh extends Tier {
    ...
    public String name() {
        return "Der Name der Kuh ist "+super.name;
    }
}
```

Vererbung von Attributen und Methoden

- Ist ein Element (Attribut oder Methode) `x` der Vaterklasse in der abgeleiteten Klasse neu definiert, wird also immer, wenn `x` in der abgeleiteten Klasse aufgerufen wird, das neue Element `x` der abgeleiteten Klasse angesprochen
- Will man auf das Element `x` der Vaterklasse zugreifen, kann dies mit dem expliziten Hinweis **`super.`**`x` erreicht werden
- Ein kaskadierender Aufruf von Vaterklassen-Elementen (z.B. **`super.super.`**`x`) ist nicht erlaubt!
- **`super`** ist eine Art Verweis auf die Vaterklasse (Vorsicht: keine Referenz auf ein ensprechendes Objekt so wie z.B. ei **`this`**)
- Existiert in der Vaterklasse `A` ein Konstruktor `A(<Parameterliste>)`, so kann im Rumpf eines Konstruktors der abgeleiteten Klasse dieser Konstruktor mit **`super`**`(<Parameterliste>)` aufgerufen werden

Beispiel: Aufruf von `super`

- Beispiel: Abgeleitete Klasse

```
public class Kuh extends Tier {
   private double milchMenge;

   public double milchMenge() { ... }

   public void melken() { ... }
}
```

- Objekte der Klasse `Kuh` erben damit alle Attribute und Methoden der Klasse `Tier`

Die Sichtbarkeitsklasse `protected`

- Zur Spezifikation der Sichtbarkeit von Attributen und Methoden einer Klasse hatten wir bisher kennengelernt:
  - **`public`**: das Element ist in allen Klassen sichtbar
  - **`private`**: das Element ist nur in der aktuellen Klasse sichtbar, also auch *nicht* in abgeleiteten Klassen!

    (Achtung: nicht sichtbar, aber natürlich vorhanden!!!)
- Zusätzlich gibt es das Schlüsselwort **`protected`**
- Elemente des Typs **`protected`** sind in der Klasse selbst und in den Methoden aller abgeleiteter Klassen sichtbar

Verstecken von Attributen

- Das Verstecken von Attributen ist eine gefährliche Fehlerquelle und sollte daher grundsätzlich vermieden werden
- Meist geschieht das Verstecken von Attributen aus Unwissenheit über die Attribute der Vaterklasse
- Problematisch ist, wenn Methoden aus der Vaterklasse vererbt werden, die auf ein verstecktes Attribut zugreifen, deren Name aber durch den Verstecken-Effekt irreführend wird, weil es ein gleich benanntes neues Attribut in der abgeleiteten Klasse gibt

Verstecken von Attributen: Beispiel

- Die Klasse `Tier` definiert das Attribut `name` in dem der Name des Tieres gespeichert ist
- Zudem gibt es eine Methode `public String getName()`, die den Wert des Attributs `name` zurrückgibt.
- In der abgeleiteten Klasse `Kuh` gibt es nun ebenfalls ein Attribut `name`, in dem der Name des Besitzers gespeichert werden soll
- Die Methode `getName()` aus der Vaterklasse wird nicht überschrieben (möglicherweise denkt der Programmierer, dass diese Methode bereits genau das richtige tut)
- Wenn nun ein Objekt der Klasse `Kuh` die Methode `getName()` aufruft, wird der Name der Kuh ausgegeben und nicht der Name des Besitzers

Überschreiben von Methoden

- Das Überschreiben von Methoden ist dagegen ein gewünschter Effekt, denn eine abgeleitete Klasse zeichnet sich eben gerade durch ein unterschiedliches Verhalten gegenüber der Vaterklasse aus
- *Late Binding* bei Methodenaufrufen: Erst zur Laufzeit wird entschieden, welcher Methodenrumpf nun ausgeführt wird, d.h. von welcher Klasse das aufrufende Objekt nun ist
- Dieses Verhalten bezeichnet man auch als *dynamisches Binden*
- Beispiel: eine Variable vom Typ `Tier` kann Objekte vom Typ `Tier`, Objekte vom Typ `Kuh` oder Objekte vom Typ `Schaf` enthalten
- Es kann erst zur Laufzeit entschieden werden, welchen Typ die Variable aktuell hat
- Überschreiben von Methoden und dynamisches Binden ist ein wichtiges Merkmal von *Polymorphismus* (siehe unten)

Warum keine Mehrfachvererbung?

- Wie bereits erwähnt, ist Mehrfachvererbung in Java nicht erlaubt
- Der Grund hierfür ist, dass bei Mehrfachvererbung verschiedene Probleme auftauchen können
- Ein solches Problem kann im Zusammenhang mit Methoden entstehen, die aus beiden Vaterklassen (mit der selben Signatur) vererbt werden und nicht in der abgeleiteten Klasse überschrieben werden
- In diesem Fall ist unklar, welche Methode ausgeführt werden soll, wenn eine dieser Methoden in der abgeleiteten Klasse aufgerufen wird

Warum keine Mehrfachvererbung?

- Beispiel: Klasse `AmphibienFahrzeug` wird von den beiden Klassen `LandFahrzeug` und `WasserFahrzeug` abgeleitet
- Die Methode `int getPS()`, die die Leistung des Fahrzeugs zurückgibt, könnte bereits in den beiden Vaterklassen implementiert sein und nicht mehr in der Klasse `AmphibienFahrzeug` überschrieben werden
- Problem:

```
AmphibienFahrzeug a = new AmphibienFahrzeug();
int ps = a.getPS();
```

- Welche Methode `getPS()` wird ausgeführt?
- Achtung: In C++ ist Mehrfachvererbung erlaubt; hier muss in unserem Beispiel dann aber explizit angegeben werden, welche Methode ausgeführt werden soll (was eine grundsätzliche Einschränkung von Polymorphismus darstellt)

Konstruktoren und Vererbung

Nochmal zu Konstruktoren:

- Grundsätzlich gilt: Konstruktoren werden *nicht* vererbt!
- Dies ist auch sinnvoll, schließlich kann ein Konstruktor der Klasse `Tier` keine Objekte der spezielleren Klasse `Kuh` erzeugen, sondern eben nur Objekte der generelleren Klasse `Tier`
- Es müssen also (wenn dies gewünscht ist), in jeder abgeleiteten Klasse eigene explizite Konstruktoren definiert werden
- Wir haben bereits gesehen, wie der Konstruktor der Vaterklasse in der abgeleiteten Klasse aufgerufen werden kann

Konstruktoren und Vererbung

- Wenn ein Objekt durch Aufruf des **`new`**-Operators und eines entsprechenden Konstruktors erzeugt wird, wird grundsätzlich immer auch (explizit oder implizit) der Konstruktor der Vaterklasse aufgerufen
- Explizit kann man dies wie bereits erwähnt durch Aufruf von **`super`**`(<Parameterliste>)` erreichen
- Achtung: dies muss der *erste* Befehl in einem expliziten Konstruktor sein! Wie ebenfalls schon diskutiert kann `<Parameterliste>` leer sein (Default-Konstruktor) oder muss zur Signatur eines expliziten Konstruktors der Vaterklasse passen

Konstruktoren und Vererbung

- Steht in einem expliziten Konstruktor der abgeleiteten Klasse kein **`super`**-Aufruf an erster Stelle, wird implizit der Default-Konstruktor der Vaterklasse **`super`**`()` aufgerufen
- Achtung: Es ist nicht erlaubt, den Default-Konstruktor aufzurufen, obwohl ein expliziter Konstruktor in der Vaterklasse vorhanden ist und der Default-Konstruktor nicht existiert (wir erinnern uns: sofern ein expliziter Konstruktor vorhanden ist, muss der Default-Konstruktor explizit angegeben werden, damit er zur Verfügung steht!)

Konstruktoren und Vererbung: Beispiel

1.Fall:

In `Tier` *ist kein* expliziter Konstruktor definiert

```
public class Kuh extends Tier
{
  public Kuh(double bisherigeMilchMenge) {
    this.milchMenge = bisherigeMilchMenge;  // (*)
  }

}
```

Beim Aufruf des Konstruktors, z.B. `Kuh erni = new Kuh(35.0);` wird, bevor Zeile `(*)` ausgeführt wird, zunächst der Konstruktor `Tier()` implizit aufgerufen.

Konstruktoren und Vererbung: Beispiel

2.Fall:

In `Tier` *ist ein* expliziter Konstruktor

`Tier(String name, int geburtsJahr)`

definiert.

```
public class Kuh extends Tier
{
  public Kuh(String name, int geburtsjahr, double  bisherigeMilchMenge) {
    super(name, geburtsjahr);
    this.milchMenge = bisherigeMilchMenge;
  }
}
```

Die Lösung von Fall 1 geht hier nicht, da der (implizit vor Zeile `(*)` aufgerufene) Default-Konstruktor nicht existiert.

Konstruktoren und Vererbung

- Offenbar dient der Aufruf des Vaterklassen-Konstruktors dazu, die vererbten Attribute in der abgeleiteten Klasse zu initialisieren
- Natürlich kann dies auch in der abgeleiteten Klasse explizit gemacht werden (ist aber meist wenig sinnvoll und widerspricht der Idee der is-a-Beziehung)
- Konstruktoren werden nicht vererbt, müssen aber (implizit oder explizit) in abgeleiteten Klassen verwendet werden (können).

Abstrakte Methoden

- Überschreiben wird besonders bei abstrakten Methoden interessant
- *Abstrakte* Methoden enthalten im Gegensatz zu konkreten Methoden nur die Spezifikation der Signatur aber keinen Methodenrumpf, der die Implementierung der Methode vereinbart
- Abstrakte Methoden werden mit dem Schlüsselwort **`abstract`** versehen und anstelle der Blockklammern (in denen der Methodenrumpf untergebracht ist) mit einem simplen Semikolon beendet
- Beispiel:

```
public abstract <Typ> abstrakteMethode(<Parameterliste>);
```

- Abstrakte Methoden können nicht aufgerufen werden, sondern definieren eine *Schnittstelle*: Erst durch Überschreiben in einer abgeleiteten Klasse und (dortige) Implementierung des Methodenrumpfes wird die Methode konkret und kann aufgerufen werden

Abstrakte Klassen

- Abstrakte Methoden spezifizieren daher eine gemeinsame Funktionalität, die alle abgeleiteten konkreten Klassen zur Verfügung stellen (aber möglicherweise unterschiedlich implementieren)
- Klassen, die mindestens eine abstrakte Methode haben, sind selbst abstrakt und müssen ebenfalls mit dem Schlüsselwort **`abstract`** gekennzeichnet werden
- Eine von einer abstrakten Vaterklasse abgeleiteten Klassen wird konkret, wenn alle abstrakten Methoden der Vaterklasse implementiert sind
- Die Konkretisierung kann auch über mehrere Vererbungsstufen erfolgen
- Es können *keine* Objekte (Instanzen) von abstrakten Klassen erzeugt werden!

Ein Beispiel für Polymorphismus

- Im folgenden sehen wir uns ein Programm an, das die Mitarbeiter der LMU verwaltet, insbesondere deren brutto Monatsgehalt berechnet
- Dazu wird zunächst die abstrakte Klasse `Mitarbeiter` definiert, die alle grundlegenden Eigenschaften eines Mitarbeiters modelliert
- In abgeleiteten Klassen werden dann die einzelnen Mitarbeitertypen `Arbeiter`, `Angestellter` und `Beamter` abgebildet und konkret implementiert
- Die Klasse `Gehaltsberechnung` zur Berechnung der Gehälter verwendet diese Klassen dann *polymorph*

Ein Beispiel für Polymorphismus

```
public abstract class Mitarbeiter {
    private int persNr;
    private String name;
    private int dienstAlter;


    public Mitarbeiter(int persNr, String name)
    {
        this.persNr = persNr;
        this.name = name;
        this.dienstAlter = 0;
    }

    public abstract double monatsBrutto();
}
```

Für einen Mitarbeiter kann also grundsätzlich das Monatsbrutto ermittelt werden. Details sind aber hier noch nicht möglich, daher ist die entspr. Methode abstrakt.

Ein Beispiel für Polymorphismus

```
public class Arbeiter extends Mitarbeiter {
    private double stundenLohn;
    private double anzahlStunden;
    private double ueberstundenZuschlag;
    private double anzahlUeberstunden;

    public Arbeiter(int persNr, String name,
                    double sL, double aS, double uZ, double aU)
    {
        super(persNr, name);
        this.stundenLohn = sL;
        this.anzahlStunden = aS;
        this.ueberstundenZuschlag = uZ;
        this.anzahlUeberstunden = aU;
    }

    /****  Konkretisierung  *****/
    public double monatsBrutto()
    {
        return   stundenLohn * anzahlStunden +
                 (stundenLohn + ueberstundenZuschlag) * anzahlUeberstunden;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Angestellter extends Mitarbeiter {
    private double grundGehalt;
    private double ortsZuschlag;
    private double zulage;

    public Angestellter(int persNr, String name, double gG, double oZ, double z)
    {
        super(persNr, name);
        this.grundGehalt = gG;
        this.ortsZuschlag = oZ;
        this.zulage = z;
    }

    /****  Konkretisierung  *****/
    public double monatsBrutto()
    {
        return grundGehalt + ortsZuschlag + zulage;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Beamter extends Mitarbeiter {
    private double grundGehalt;
    private double familienZuschlag;
    private double stellenZulage;

    public Beamter(int persNr, String name, double gG, double fZ, double sZ)
    {
        super(persNr, name);
        this.grundGehalt = gG;
        this.familienZuschlag = fZ;
        this.stellenZulage = sZ;
    }

    /****  Konkretisierung  *****/
    public double monatsBrutto()
    {
        return grundGehalt + familienZuschlag + stellenZulage;
    }
}
```

Ein Beispiel für Polymorphismus

```
public class Gehaltsberechnung {
    public static void main(String[] args)
    {
        Mitarbeiter[] ma = new Mitarbeiter[3];

        ma[0] = new Beamter(1, "Meier", 3021.37, 91.50, 10.70);
        ma[1] = new Angestellter(2, "Maier", 2303.21, 502.98, 132.65);
        ma[2] = new Arbeiter(3, "Mayr", 20.0, 113.5, 35.0, 11.0);

        double bruttoSumme = 0.0;

        for(int i=0; i<ma.length; i++)
        {
            bruttoSumme += ma[i].monatsBrutto();
            // Nur moeglich, da die Klasse Mitarbeiter diese Methode
            // bereitstellt (als abstrakte Methode)
        }

        System.out.println("Bruttosumme = "+bruttoSumme);
    }
}
```