Kapitel 7

Vererbung und Polymorphismus

Skript zur Vorlesung

Einführung in die Programmierung

im Wintersemester 2012/13

Ludwig-Maximilians-Universität München

(c) Peer Kröger, Arthur Zimek 2009, 2012

Übersicht

Übersicht

7.1 Das Konzept der Vererbung

- Vererbung ist die Umsetzung von "is-a"-Beziehungen.
- Alle Elemente (Attribute und Methoden) der (generelleren) *Vaterklasse* (*Oberklasse*) sollen auf die (speziellere) *abgeleitete* (*Unter-*)*Klasse* vererbt werden.
- Zusätzlich kann die abgeleitete Klasse neue Elemente (Attribute und Methoden) definieren (da sie ja eine Spezialisierung ist).
- Vererbung ist ein wichtiges Mittel zur Wiederverwendung von Programmteilen: Funktionalitäten, die in der Vaterklasse bereits implementiert sind, werden vererbt, müssen also in den abgeleiteten Klassen *nicht* noch einmal implementiert werden (außer, die Funktionalität wird „*redefiniert*", also *überschrieben*)

7.1 Das Konzept der Vererbung

- Beispiel
  - Wir wollen die Tiere (bestehend aus Schafen und Kühen) auf einem Bauernhof verwalten und dafür modellieren.
  - Dazu stellen wir fest, dass wir für alle Kühe und Schafe jeweils einen Namen und ein Geburtsjahr verwalten wollen, das Alter soll abrufbar sein, ebenfalls für ein beliebiges Jahr das entsprechende Alter der Tiere
  - Bei den Kühen interessieren wir uns für die Milchmenge, die sie aktuell abgegeben können; Kühe können natürlich gemolken werden, was die aktuelle Milchmenge verringert (die Milchmenge wird sich dann wieder irgendwie im Laufe der Zeit erhöhen …)
  - Bei den Schafen interessiert uns die Menge der Wolle, die wir aktuell scheren können; Schafe können geschoren werden, was die aktuelle Wollmenge verringert …

7.1 Das Konzept der Vererbung

- Modellierung (UML)

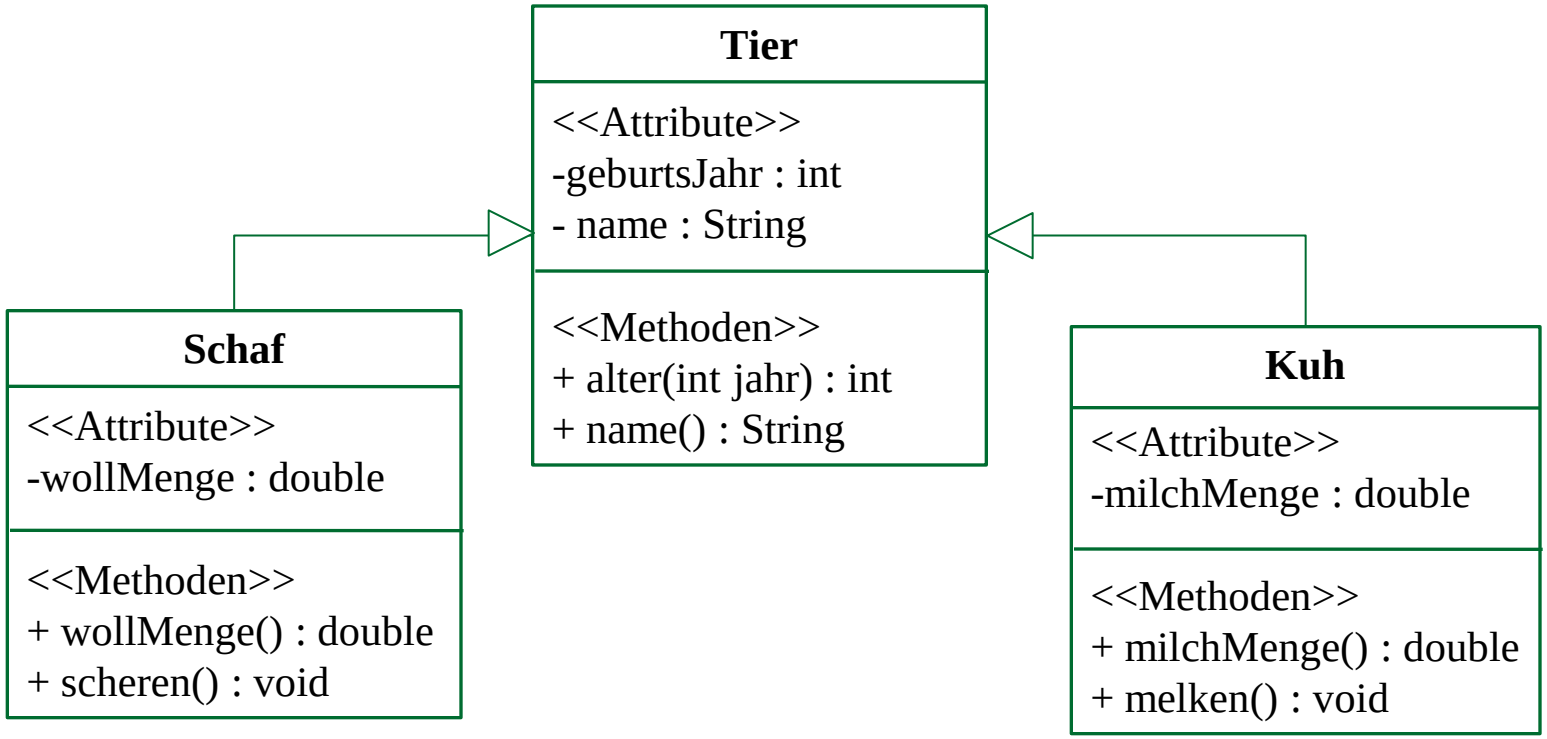
Kuh
<<Attribute>> - milchMenge : double - name : String - geburtsJahr: int
<<Methoden>> + alter(int aktJahr): int + name():String + milchMenge() : double + melken() : void

Schaf
<<Attribute>> - wollMenge : double - name : String - geburtsJahr: int
<<Methoden>> + alter(int aktJahr): int + name():String + wollMenge() : double + scheren() : void

Probleme???

7.1 Das Konzept der Vererbung

- Offensichtlich haben beide Tierarten gemeinsame Eigenschaften (Namen, Geburtsjahr) und Funktionalitäten (Methoden)
- Es liegt nahe, diese Gemeinsamkeiten zu generalisieren und in eine gemeinsamen Oberklasse auszulagern



7.1 Das Konzept der Vererbung

- Vorteil?

  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)

- Es ist also ein Zeichen für einen guten oo-Entwurf, gemeinsame Eigenschaften von Klassen zu identifizieren und in eine gemeinsame Oberklasse auszulagern
- Vorsicht, dies ist nicht immer sinnvoll; die Vererbungshierarchie sollte einigermaßen intuitiv sein

Übersicht

7.2 Vererbung 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 (siehe später).
- In Java zeigt des Schlüsselwort **`extends`** Vererbung an.
  Beispiel:

```
public class Kuh extends Tier
{
        ...
}
```

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

7.2 Vererbung in Java

- Enthält eine Klasse keine **`extends`**-Klausel, so besitzt sie die *implizite Vaterklasse* `Object` (im Paket `java.lang`, das automatisch geladen wird, also nicht explizit mit **`import`**-Anweisung importiert werden muss).
- Also: Jede Klasse, die keine **`extends`**-Klausel enthält, wird direkt von `Object` (eigentlich `java.lang.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.

7.2 Vererbung in Java

- Die Klasse `Object` definiert einige elementare Methoden, die für alle Arten von Objekten nützlich sind, u.a.:
  - `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.

7.2 Vererbung in Java

- 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 des *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.
- Also:
  - Wenn ein Element (Attribut oder Methode) der Vaterklasse in der abgeleiteten Klassen definiert wird, wird dieses Element *zusätzlich* zu dem in der Vaterklasse angelegt!
  - Wenn nichts angegeben ist, werden die Elemente der Vaterklasse automatisch an die abgeleitet Klasse weitergegeben!

7.2 Vererbung in Java

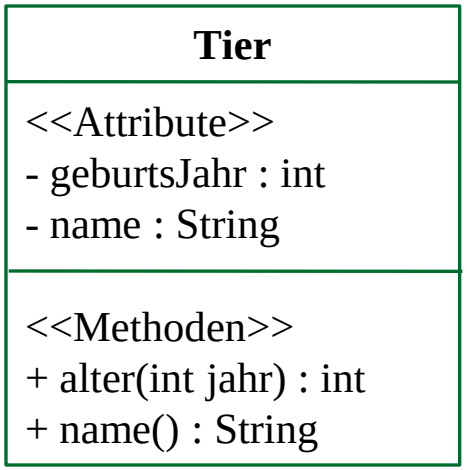
- Ist ein Element (Attribut oder Methode) x der Vaterklasse in der abgeleiteten Klasse neu definiert, wird also immer, wenn x in der abgeleiteten Klasse aufgerufen wird, das neue Element x der abgeleiteten Klasse angesprochen.
- Will man auf das Element x der Vaterklasse zugreifen, kann dies mit dem 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, es zeigt an, dass ein Element der Vaterklasse aufgerufen wird (Vorsicht: es ist *kein* Verweis/Zeiger auf ein ensprechendes Objekt der Vaterklasse).
- Existiert in der Vaterklasse `K` ein Konstruktor `K(<Parameterliste>)`, so kann im Rumpf eines Konstruktors der abgeleiteten Klasse dieser Konstruktor mit **`super`**`(<Parameterliste>)` aufgerufen werden

7.2 Vererbung in Java

- Wir sind nun endlich in der Lage, alle Möglichkeiten zu verstehen, die Sichtbarkeit von Elementen zu spezifizieren.
- 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
- Achtung: Damit ist ein als **`private`** deklariertes Element also auch *nicht* in abgeleiteten Klassen sichtbar!
- Zusätzlich gibt es das Schlüsselwort **`protected`** (in UML „#“)
- Elemente mit Sichtbarkeitsspezifikation **`protected`** sind in der Klasse selbst und *in den Methoden* abgeleiteter Klassen sichtbar.

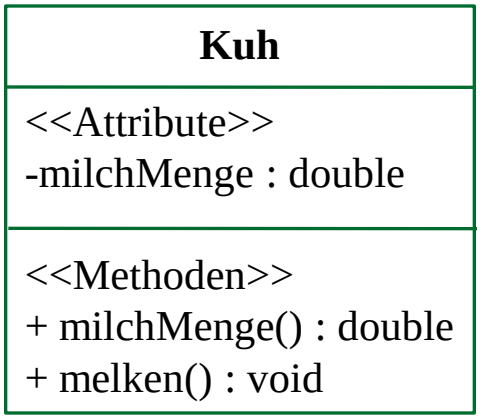
7.2 Vererbung in Java

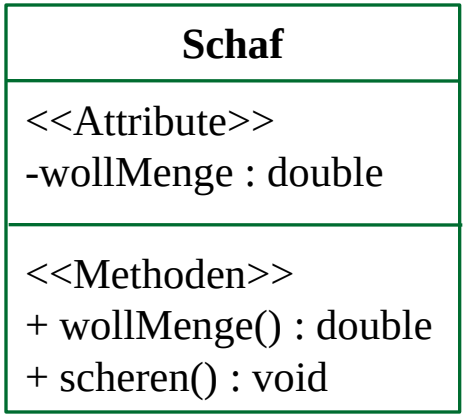

```
public class Tier {
   private int geburtsJahr;
   private String name;

   public Tier(String name, int gebJahr) {
     this.geburtsJahr = gebJahr;
     this.name = name;
   }

   public int alter(int jahr) {
     return jahr – this.geburtsJahr;
   }

   public String name() {
     return this.name;
   }

   public static void printTierListe(Tier[] tiere) {
     for(int i=0; i<tiere.length; i++) {
       System.out.println(“Name: “+tiere[i].name()+”, “+tiere[i].alter(2013)+” Jahre alt”);
      }
   }
}
```



7.2 Vererbung in Java


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

   public Kuh(int gebJahr, String name, double milchMenge) {
     super(name, gebJahr);
     this.milchMenge = milchMenge;
   }

   public double milchMenge() { … }

   public void melken() { … }
}

public class Schaf extends Tier {
   private double wollMenge;

   public Schaf(int gebJahr, String name, double wollMenge) {
     super(name, gebJahr);
     this.wollMenge = wollMenge;
   }

   public double wollMenge() { … }

   public void scheren() { … }
}
```





7.2 Vererbung in Java

- Bemerkung:
  - Die Attribute der Vaterklasse sind hier `private`, daher sind sie in den Klassen `Kuh` und `Schaf` nicht sichtbar. Initialisierung dieser Attribute geht in diesem Beispiel nur über den Konstruktor der Klasse `Tier`, der in den Konstruktoren der abgeleiteten Klassen verwendet wird.
  - Um Zugriff auch in den abgeleiteten Klassen zu gewährleisten, hätten die Attribute in Klasse `Tier` als `protected` spezifiziert werden müssen.
- Überall, wo ein Objekt vom Typ der Vaterklasse verlangt ist, darf nun auch ein Objekt vom Typ der abgeleiteten Klasse stehen.

```
Kuh milla = new Kuh("Milla", 2010, 3.7);
Kuh resi = new Kuh("Resi", 2011, 3.1);
Schaf sven = new Schaf("Sven", 2009, 7.7);
Tier[] tiere = {milla, resi, sven};
Tier.printTierListe(tiere);    // milla, resi und sven haben
                               // die Methoden alter und name
                               // geerbt.
```

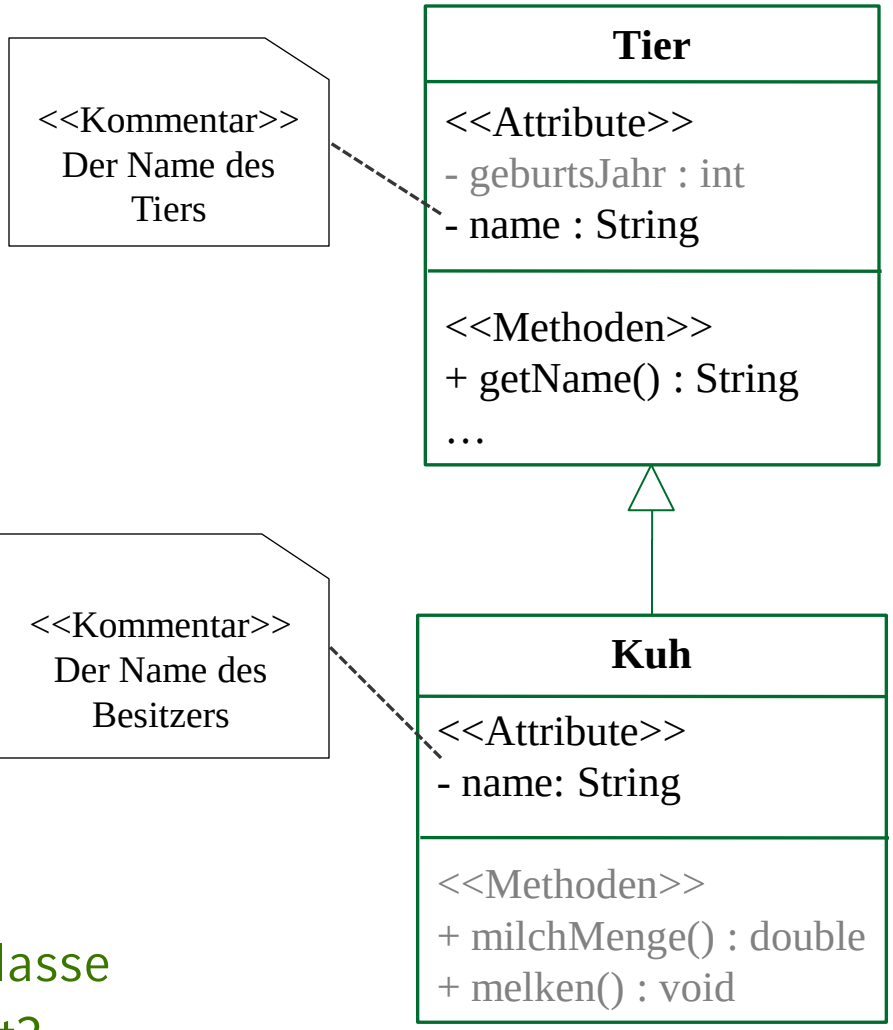
7.2 Vererbung in Java

- Nochmal genauer: 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.

7.2 Vererbung in Java

- Beispiel:

```
public class Tier {
      /** Der Name des Tiers.*/
      private String name;
      …
      public String getName() {
            return this.name;
      }
}

public class Kuh extends Tier {
      /** Der Name des Besitzers.*/
      private String name;
      …
// getName wird nicht ueberschrieben
}
```



- Was passiert, wenn ein Objekt der Klasse Kuh die Methode `getName()` aufruft?

7.2 Vererbung in Java

- Problem dieser Modellierung:
  - Die Klasse `Tier` definiert ein 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.
  - Wenn nun ein Objekt der Klasse `Kuh` die Methode `getName()` aufruft, wird der Name der Kuh ausgegeben und nicht der Name des Besitzers.

7.2 Vererbung in Java

- Nochmal genauer: Überschreiben von Methoden
  - Das Überschreiben von Methoden ist dagegen ein gewünschter Effekt, denn eine abgeleitete Klasse zeichnet sich 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 tatsächlich 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 sind wichtige Merkmale des Konzeptes *Polymorphismus*.

7.2 Vererbung in Java

- Warum keine Mehrfachvererbung in Java?
  - Bei Mehrfachvererbung können verschiedene Probleme auftreten.
  - Ein solches Problem kann im Zusammenhang mit Methoden entstehen, die aus beiden Vaterklassen 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.
  - Beispiel:
    - Klasse `AmphibienFahrzeug` wird von den beiden Klassen `LandFahrzeug` und `WasserFahrzeug` abgeleitet.
    - Die Methode `getPS()`, die die Leistung des Fahrzeugs (als `int`) zurückgibt, könnte bereits in den beiden Vaterklassen implementiert sein und nicht mehr in der Klasse AmphibienFahrzeug überschrieben werden.
    - Welche Methode `getPS()` wird hier ausgeführt?

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

7.2 Vererbung in Java

Nocheinmal 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

7.2 Vererbung in Java

- Dies muss aber etwas präzisiert werden:
  - Wenn ein Objekt mittels des **new**-Operators und eines entspr. Konstruktors erzeugt wird, wird grundsätzlich auch (explizit oder implizit) der Konstruktor der Vaterklasse aufgerufen.
  - Explizit kann man dies durch Aufruf von **`super`**`(<Parameterliste>)` passieren, so wie wir das kennengelernt haben.
  - Vorsicht: 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.
  - 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!).

7.2 Vererbung in Java

- Beispiel: In `Tier` ist *ein* expliziter Konstruktor

  `Tier(String name, int geburtsJahr)`

  definiert (so, wie wir es auf Folie 16 implementiert haben).

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

Da der Default-Konstruktor in der Vaterklasse `Tier` nicht existiert, muss einer der expliziten Konstruktoren der Klasse `Tier` als erster Befehl im Konstruktor der Unterklasse aufgerufen werden (in unserem Beispiel gibt es nur einen solchen expliziten Konstruktor).

7.2 Vererbung in Java

- Beispiel: In `Tier` ist *kein* expliziter Konstruktor definiert.

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

}
```

Da es den Default-Konstruktor in der Vaterklasse `Tier` gibt, muss kein Konstruktor für die Vaterklasse als erster Befehl im Konstruktor der Unterklasse aufgerufen werden.

Beim Aufruf des Konstruktors, z.B. `Kuh erni = new Kuh(3.5);` wird aber, bevor Zeile (*) ausgeführt wird, zunächst der Konstruktor `Tier()` implizit aufgerufen.

7.2 Vererbung in Java

- 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, falls die Attribute der Vaterklasse das Sichtbarkeitsattribut **`protected`** besitzen (ist aber meist wenig sinnvoll). Falls die Attribute der Vaterklasse als **`private`** deklariert sind, ist dies nicht möglich.
- Konstruktoren werden nicht vererbt, müssen aber (implizit oder explizit) in abgeleiteten Klassen verwendet werden (können).

Übersicht

7.3 Abstrakte Klassen und Polymorphismus

- *Abstrakte* Methoden enthalten im Gegensatz zu konkreten Methoden nur die Spezifikation der Signatur.
- Abstrakte Methoden enthalten also keinen Methodenrumpf, der die Implementierung der Methode vereinbart.
- Abstrakte Methoden werden mit dem Schlüsselwort **`abstract`** versehen und anstelle der Blockklammern für den Methodenrumpf 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.

7.3 Abstrakte Klassen und Polymorphismus

- 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 Klasse wird konkret, wenn alle abstrakten Methoden der Vaterklasse implementiert sind. Die Konkretisierung kann auch über mehrere Vererbungsstufen erfolgen. Natürlich werden alle nicht abstrakten Elemente (insb. Attribute) "ganz normal" vererbt.
- Es können *keine* Objekte (Instanzen) von abstrakten Klassen erzeugt werden! *WARUM*?

7.3 Abstrakte Klassen und Polymorphismus

- Beispiel: Wir wollen die Mitarbeiter der LMU verwalten, insbesondere deren brutto Monatsgehalt berechnen können.
- Dazu wird zunächst die abstrakte Klasse `Mitarbeiter` definiert, die alle grundlegenden Eigenschaften eines Mitarbeiters modelliert.
- Insbesondere definieren wir eine Methode `monatsBrutto`, die zunächst abstrakt ist (weil wir für die allgemeine Klasse `Mitarbeiter` noch keine Implementierung angeben können)
- In abgeleiteten Klassen werden dann die einzelnen Mitarbeitertypen `Arbeiter`, `Angestellter` und `Beamter` abgebildet und konkret implementiert (d.h. insbesondere die abstrakte Methode `monatsBrutto`).
- Die Klasse `Gehaltsberechnung` verwendet diese Klassen *polymorph*.

7.3 Abstrakte Klassen und 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 abstrakt

7.3 Abstrakte Klassen und 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;
    }

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

Konkretisierung

7.3 Abstrakte Klassen und 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;
    }

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

Konkretisierung

7.3 Abstrakte Klassen und 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;
    }

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

Konkretisierung

7.3 Abstrakte Klassen und 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();
        }

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

Nur möglich, da die Klasse Mitarbeiter diese Methode bereitstellt (als abstrakte M.)