Abschnitt 9: Weitere Konzepte der oo Programmierung in Java

9. Weitere Konzepte der oo Programmierung in Java
9.1 Vererbung, abstrakte Klassen, Polymorphismus
9.2 Interfaces
9.3 Ausnahmen
9.4 Polymorphie versus Typsicherheit

Überblick

9. Weitere Konzepte der oo Programmierung in Java
9.1 Vererbung, abstrakte Klassen, Polymorphismus
9.2 Interfaces
9.3 Ausnahmen
9.4 Polymorphie versus Typsicherheit

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 aber natürlich neue Elemente (Attribute und Methoden) definieren bzw. ein bereits existierendes Element verfeinern (*überschreiben*), denn sie ist ja eine Spezialisierung von A.

Vererbung

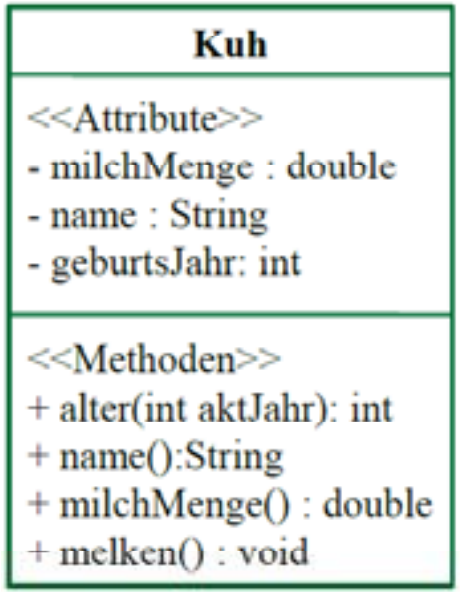
- Vererbung ist ein wichtiges Mittel zur Wiederverwendung von Programmteilen:
  Funktionalitäten, die in der Vaterklasse A bereits implementiert sind, werden automatisch auf die abgeleitete Klasse B vererbt, müssen also in B nicht noch einmal implementiert werden (außer, die betreffende Funktionalität wird spezialisiert, d.h. redefiniert/überschrieben).

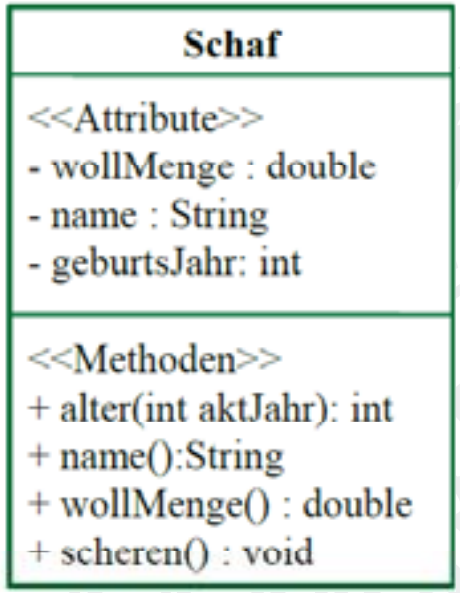
Beispiel

- 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, die wir scheren konnten, etc.
- Dies kann man mit Vererbung entsprechend modellieren (siehe übernächste Folie).

Beispiel

Zunächst ohne Vererbung:

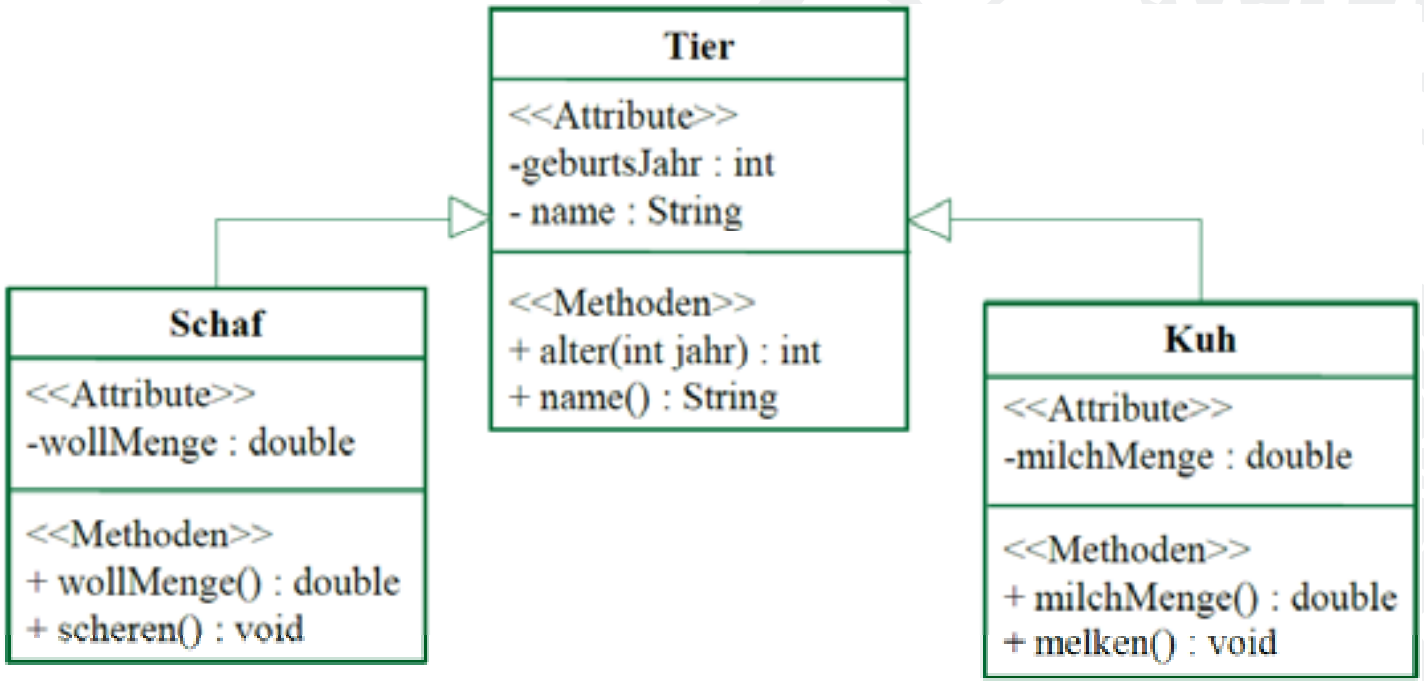




Beispiel

Jetzt mit Vererbung:

Die gemeinsamen Eigenschaften (Attribute) und Funktionalitäten (Methoden) sind in eine gemeinsame 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() { ... }
}
```

- Beispiel: Abgeleitete Klasse

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

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

Ableiten einer Klasse in Java

- Objekte der Klasse `Kuh` erben damit alle Attribute und Methoden der Klasse `Tier`.
- Variablen vom Typ `Tier` können nun auch Objekte vom Typ `Kuh` aufnehmen (aber nicht anders herum!), z.B.:

```
Tier t = new Kuh();
```

oder auch

```
Tier t = new Tier();

Kuh k = new Kuh();

t = k;
```

- Diesen Effekt nennt man *Polymorphismus* (von Objekten, d.h. die Objekte „in“ der Variablen `t` sind polymorph).

Ableiten einer Klasse in Java

- Was bedeutet das Erben von Methoden (und Attributen)?
- 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 `name` aus der Vaterklasse.

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 automatisch 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 (Achtung: in `Object` ist Identität implementiert!).
  - `Object clone()`
    kopiert ein Objekt, d.h. legt eine neues Objekt an, das eine genaue Kopie des ursprünglichen Objekts ist (Achtung: in `Object` ist das eine flache Kopie!).
  - 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 des *Versteckens/Verdeckens* 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: Überschreiben von Methoden

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

```
public class Tier {
    ...
    public String name() {
        return this.name;
    }
}
```

- Und in `Kuh` dagegen:

```
public class Kuh extends Tier {
    ...
    public String name() {
        return "Kuh "+this.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 entsprechendes Objekt so wie z.B. bei **`this`**)

Beispiel: Aufruf von `super`

- Der Zugriff auf den Namen könnte in `Kuh` auch mit Hilfe von **`super`** implementiert sein:

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

- Hier wird nun die Methode `name()` aus `Tier` aufgerufen (statt wie vorher das Attribut `name` aus `Tier`, das in `Kuh` ja nicht versteckt wurde und daher mit **`this.`**`name` in `Kuh` angesprochen wird).

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!!!)
  - default/package-scoped: das Element ist in allen Klassen desselben Packages sichtbar.
- Zusätzlich gibt es das Schlüsselwort **`protected`**: diese Elemente sind in der Klasse selbst und in den Methoden aller abgeleiteter Klassen sichtbar (und zusätzlich in allen Klassen des selben Packages).

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.
- 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.
- *Dynamisches Binden* (*late binding*) bei Methodenaufrufen: Erst zur Laufzeit wird entschieden, welcher Methodenrumpf nun ausgeführt wird, d.h. von welcher Klasse das aufrufende Objekt nun tatsächlich ist (wegen Polymorphismus ist das erst zur Laufzeit klar!).
- Dabei wird die Methode des *speziellsten* Typs ausgewählt.

Überschreiben von Methoden

- Was bedeutet „speziellster“ Typ?
- Eine Variable vom Typ `A` kann Objekte vom Typ `A`, Objekte vom Typ `B extends A` oder Objekte vom Typ `C extends B` enthalten.
- Wird für die Variable eine Methode `m` aufgerufen, die in `B` überschrieben wurde, kann erst zur Laufzeit entschieden werden, welchen Typ das Objekt besitzt, das von der Variable aktuell referenziert wird:
  - Ist das Objekt vom Typ `A` wird `m` aus `A` ausgeführt.
  - Ist das Objekt vom Typ `B` wird `m` aus `B` ausgeführt.
  - Ist das Objekt vom Typ `C` wird `m` aus `B` ausgeführt, denn diese Version ist die speziellste, die für die Klasse `C` „gefunden werden kann“.
- Überschreiben von Methoden und dynamisches Binden ist ein wichtiges Merkmal von Polymorphismus.

Überschreiben von Methoden

- Beispiel: eine Variable vom Typ `Tier` kann Objekte vom Typ `Tier`, Objekte vom Typ `Kuh` oder Objekte vom Typ `Schaf` enthalten.
- Wird für die Variable eine Methode `m` aufgerufen, die in den abgeleiteten Klassen überschrieben wurde, kann erst zur Laufzeit entschieden werden, welchen Typ das Objekt besitzt, das von der Variable aktuell referenziert wird, und damit, welche Methode die speziellste ist und ausgeführt wird.

Beispiel: Late Binding

```
...

boolean b;

...

Tier t = new Tier();
Kuh k = new Kuh();
if(b)
{
   t=k;
}
System.out.println(t.name());      // (*)
```

Welche Methode wird bei `(*)` ausgeführt?

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 (da dort auch das entsprechende Attribut vereinbart ist) und nicht mehr in der Klasse `AmphibienFahrzeug` überschrieben werden.
- Problem:

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

- Welche Methode `getPS()` wird ausgeführt?

Warum keine Mehrfachvererbung?

- 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

Konstruktoren sind im Kontext Verebung nochmal genauer zu betrachten:

- 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.
- 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.

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 explizit implementierten Konstruktor sein!
- Der Aufruf muss die Signatur eines Konstruktors der Vaterklasse erfüllen.

Konstruktoren und Vererbung

- Wie gesagt: 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!).
- Das bedeutet andersherum: steht der Default-Konstruktor in der Vaterklasse nicht zur Verfügung, muss in der abgeleiteten Klasse einer der expliziten Konstruktoren der Vaterklasse aufgerufen werden!

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 (wir hätten ihn natürlich auch explizit aufrufen können).

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).

Beispiel zur Vererbung

- Auch die Klasse `Rechteck` kann man als Spezialisierung von `Punkt` betrachten.
- Ein Rechteck ist ein Punkt mit zusätzlicher Ausdehnung entlang der Koordinatenachsen.

```
public class Punkt {
    private double x;
    private double y;

    public Punkt(double x, double y) {
        this.x = x;
        this.y = y;
    }

    public double getX() { return this.x; }

    public double getY() { return y; }

    public void verschiebe(double xOff, double yOff) { ...  // wie bisher }

}
```

Beispiel zur Vererbung

- `Rechteck` hat damit automatisch `verschiebe(...)`.

```
public class Rechteck extends Punkt {
    private double xExt;
    private double yExt;

    public Rechteck(double xCoord, double yCoord, double xExt, double yExt) {
        super(xCoord, yCoord);
        this.xExt = xExt;
    }

    public Rechteck(Punkt p, double xExt, double yExt) {
        super(p.getX(), p.getY());
    }

    public Punkt getOrigin()  {
        return new Punkt(super.getX(), super.getY());
    }

    public boolean istEnthalten(Punkt p) {
        return (super.getX() <= p.getX() && p.getX() <= super.getX()+xExt)
               &&
               (super.getY() <= p.getY() && p.getY() <= super.getY()+yExt);
    }
}
```

Beispiel zur Vererbung

- Wären die Attribute der Klasse `Punkt` (`x` und `y`) nicht **`private`** sondern **`protected`**, so könnte man in `Rechteck` auch direkt auf die Attribute der Oberklasse zugreifen:

```
public class Punkt {
    protected double x;
    protected double y;
    ...
}

public class Rechteck extends Punkt {
    ...
    public Rechteck(Punkt p, double xExt, double yExt) {
        super(p.x, p.y);
    }

    public Punkt getOrigin()  {
        return new Punkt(super.x, super.y);
    }

    public boolean istEnthalten(Punkt p) {
        return (super.x <= p.getX() && p.getX() <= super.x+xExt)
               &&
               (super.y <= p.getY() && p.getY() <= super.y+yExt);
    }
}
```

Umsetzung von Aufzählungstypen

- Die Vereinbarung

```
public enum AmpelFarben { ROT, GELB, GRUEN };
```

setzt der Compiler wie folgt um:

```
public final class AmpelFarbe extends Enum {
    public static final AmpelFarbe ROT = new AmpelFarbe("ROT", 0);
    public static final AmpelFarbe GELB = new AmpelFarbe("GELB", 0);
    public static final AmpelFarbe GRUEN = new AmpelFarbe("GRUEN", 0);

    private AmpelFarbe(String s, int i) {
         super(s,i);
    }

}
```

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.

Abstrakte Methoden

- Syntax:

```
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 Methoden spezifizieren daher eine gemeinsame Funktionalität, die alle abgeleiteten konkreten Klassen zur Verfügung stellen (aber möglicherweise unterschiedlich implementieren).

Abstrakte Klassen

- 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);
    }
}
```

Überblick

9. Weitere Konzepte der oo Programmierung in Java
9.1 Vererbung, abstrakte Klassen, Polymorphismus
9.2 Interfaces
9.3 Ausnahmen
9.4 Polymorphie versus Typsicherheit

Schnittstellen

- Die Objekt-Methoden einer Klasse definieren die Verhaltensweisen von Objekten dieser Klasse.
- Aus Sicht eines Anwenders der Klasse können diese Verhaltensweisen auch als *Funktionalitäten* oder *Dienstleistungen* bezeichnet werden.
- Dabei interessiert sich der Anwender (z.B. eine andere Klasse) nicht wirklich für die Implementierungsdetails, er muss nur die Signatur einer Methode kennen, um sie verwenden zu können.
- Andererseits hängen die Implementierungsdetails nicht von der konkreten Verwendung durch einen Anwender ab.

Schnittstellen

- Der Implementierer möchte also nur wissen, welche Funktionalitäten bereitgestellt werden müssen.
- Der Anwender hingegen möchte nur wissen, welche Funktionalitäten bereitgestellt werden.
- Beide richten sich nach einer gemeinsamen „Schablone“ (*Schnittstelle*, *Interface*), der Implementierer „von innen“, der Anwender „von außen“.

Schnittstellen

- Eine Schnittstelle definiert also Dienstleistungen, die für Anwender (z.B. aufrufende Klassen) zur Verfügung stehen müssen und die der Implementierer bereitstellen muss.
- Dabei werden in der Schnittstelle die Implementierungsdetails der Dienstleistungen (im Ggs. zu Klassen) *nicht* festgelegt.
- Es werden *funktionale Abstraktionen* in Form der Methodensignaturen bereitgestellt, die das WAS, aber nicht das WIE festlegen.

Schnittstellen

- Interfaces bestehen i.d.R. nur aus Methodensignaturen, d.h. sie besitzen insbesondere keine Methodenrümpfe und keine Attribute (in Java dürfen sie zusätzlich statische Konstanten spezifizieren).
- Interfaces sind daher ähnlich zu abstrakten Klassen, die ausschließlich abstrakte Methoden besitzen.
- Interfaces sind gültige Objekttypen für Variablen, es gibt aber keine Objekte (Instanzen) dieses Typs.

Realisierungsbeziehung

- Ein Interface spezifiziert also eine gewisse Funktionalität, ist selbst aber nicht instanziierbar.
- Implementiert eine (nicht-abstrakte) Klasse K ein Interface I, müssen alle Methoden des Interfaces in der Klasse implementiert werden.
- Zwischen K und I besteht dann eine Realisierungsbeziehung.

Schnittstellen/Interfaces in Java

- In Java werden Schnittstellen mit dem Schlüsselwort **`interface`** anstelle von **`class`** vereinbart.
- Alle Eigenschaften für den Namensraum von Klassen bzgl. Packages gelten analog für Interfaces.
- Alle Methoden eines Interfaces sind grundsätzlich **`public`**, daher ist keine Sichtbarkeitsspezifikation nötig (**`public`** darf aber explizit geschrieben werden).

Beiepiel: Interfaces in Java

- Als Beispiel definieren wir ein Interface `RaeumlichesObjekt`, das den Zugriff auf die Ausdehnung eines (3-dimensionalen) räumlichen Objekts festlegt.

```
public interface RaeumlichesObjekt {
    /** Die L&auml;nge des Objekts in mm. */
    int laenge();

    /** Die H&ouml;he des Objekts in mm. */
    int hoehe();

    /** Die Breite des Objekts in mm. */
    int breite();
}
```

Interfaces in Java

- In einem zweiten Beispiel definieren wir ein Interface `Farbig`, das den Zugriff auf die Farbe eines Objekts festlegt und einige wichtige Farben als Konstanten definiert.
- Alle *Attribute* eines Interfaces sind grundsätzlich globale, statische Konstanten, daher werden die Zusätze **`public static final`** nicht benötigt (dürfen aber natürlich wieder explizit geschrieben werden).

Interfaces in Java

- Die Definition des Interface `Farbig`

```
import java.awt.*; // fuer die Klasse Color

public interface Farbig {
    /** Die Farbe (als RGB-Zahl) des Objekts. */
    int farbe();

    /** Die Farbe "schwarz". */
    int SCHWARZ = Color.BLACK.getRGB();
    /** Die Farbe "weiss". */
    int WEISS = Color.WHITE.getRGB();
    /** Die Farbe "rot". */
    int ROT = Color.RED.getRGB();
    /** Die Farbe "gr&uuml;n". */
    int GRUEN = Color.GREEN.getRGB();
    /** Die Farbe "blau". */
    int BLAU = Color.BLUE.getRGB();
}
```

Implementierung von Interfaces in Java

- Das Interface `RaeumlichesObjekt` beschreibt zunächst nur die gewünschte Funktionalität, stellt diese aber noch nicht zur Verfügung.
- Die Funktionalität kann nur von einer konkreten Klasse zur Verfügung gestellt werden.
- Dazu muss die entsprechende Klasse das Interface *implementieren* – dies wird mit dem Schlüsselwort **`implements`** `<InterfaceName>` nach dem Klassennamen angezeigt.
- Die Klasse muss dann Methodenrümpfe für alle Methoden des Interfaces definieren.

Implementierung von Interfaces in Java

- Eine Klasse kann auch mehrere Interfaces implementieren (dabei muss sie dann *alle* Methoden *aller* Interfaces implementieren).
- Im folgenden sehen wir drei Beispielklassen, die alle das Interface `RaeumlichesObjekt` implementieren.
- Zwei Klassen implementieren zudem das Interface `Farbig`.

Implementierung von Interfaces

```
public class Auto implements RaeumlichesObjekt, Farbig {
    private int laenge;
    private int hoehe;
    private int breite;
    private int farbe = WEISS;
    // weitere Attribute ...

    public int laenge() {
        return this.laenge;
    }

    public int hoehe() {
        return this.hoehe;
    }

    public int breite() {
        return this.breite;
    }

    public int farbe() {
        return this.farbe;
    }
}
```

Implementierung von Interfaces

```
public class FussballPlatz implements RaeumlichesObjekt, Farbig {
    public int laenge() {
        return 105000;
    }

    public int hoehe() {
        return 0;
    }

    public int breite() {
        return 65000;
    }

    public int farbe() {
        return GRUEN;
    }
}
```

Implementierung von Interfaces

```
public class PapierBlatt implements RaeumlichesObjekt {
    private final int FORMAT;

    public int laenge() {
        int erg = 0;
        switch(FORMAT)
        {
            case 0 : erg = 1189; break;
            case 1 : erg = 841; break;
            case 2 : erg = 594; break;
            case 3 : erg = 420; break;
            case 4 : erg = 297; break;
            // usw. ...
        }
        return erg;
    }

    ... // Fortsetzung naechste Seite
```

Implementierung von Interfaces

```
    ...

    public int hoehe()
    {
        return 0;
    }

    public int breite()
    {
        int erg = 0;
        switch(FORMAT)
        {
            case 0 : erg = 841; break;
            case 1 : erg = 594; break;
            case 2 : erg = 420; break;
            case 3 : erg = 297; break;
            case 4 : erg = 210; break;
            // usw. ...
        }
        return erg;
    }
}
```

Verwendung von Interfaces

- Nützlich sind Interfaces u.a. dann, wenn Eigenschaften einer Klasse beschrieben werden sollen, die nicht direkt in ihrer “normalen” Vererbungshierarchie abgebildet werden können.
- Hätten wir `RaeumlichesObjekt` als abstrakte Vaterklasse definiert, und `Auto`, `FussballPlatz` und `PapierBlatt` daraus abgeleitet, ergäbe das eine etwas unnatürliche Vererbungshierarchie.
- Durch die Implementierung des Interfaces `RaeumlichesObjekt` können die drei Klassen die Methoden `laenge`, `hoehe` und `breite` dagegen *unabhängig* von ihrer Vererbungslinie garantieren (dies wird später auch bei der Umsetzung einer Mehrfachverebung genutzt).

Verwendung von Interfaces

- Definierte Interfaces können ähnlich wie abstrakte Klassen verwendet werden.
- Das folgende einfache Beispiel illustriert die Anwendung des Interfaces `RaeumlichesObjekt`:

  Eine Methode `volumen` (die z.B. eine Hilfsmethode einer `main`-Methode sein kann), die das Volumen von räumlichen Objekten berechnet, kann nun wie folgt definiert sein:

```
public static int volumen(RaeumlichesObjekt obj) {
    return obj.laenge() * obj.hoehe * obj.breite();
}
```

Verwendung von Interfaces

- Die Methode `volumen` akzeptiert als Eingabe nun nur Objekte von Klassen, die das Interface `RaeumlichesObjekt` implementieren, also in unserem Beispiel nur Objekte der Klassen `Auto`, `FussballPlatz` und `PapierBlatt`.
- So ist sichergestellt, dass die Methoden `laenge`, `hoehe` und `breite` tatsächlich für den Eingabeparameter `obj` zur Verfügung stehen.
- Wie bereits erwähnt, kann man für Interfaces keine Objekte instanziieren, d.h. die Methode `volumen` kann nicht für ein Objekt vom Typ `RaeumlichesObjekt` aufgerufen werden, sondern nur für Objekte vom Typ einer Klasse, die `RaeumlichesObjekt` implementiert.
- Dennoch ist ein Interface ein gültiger Objekttyp für Variablen.

Interfaces und Vererbung

- Eine Klasse, die ein Interface implementiert, kann auch Vaterklasse für ein oder mehrere abgeleitete Klassen sein.
- Dann erben alle abgeleiteten Klassen natürlich auch alle Methoden des Interfaces (die ja in der Vaterklasse implementiert wurden und ggf. nochmals überschrieben werden können).
- Dadurch “implementieren” auch alle abgeleiteten Klassen die Interfaces, die von der Vaterklasse implementiert werden.
- Auch Interfaces selbst können abgeleitet werden.
- Das abgeleitete Interface erbt alle Methoden des Vater-Interface.
- Eine implementierende Klasse muss damit auch alle Methoden aller Vater-Interfaces implementieren.

Beispiel: Vererbung von Interfaces

```
public interface EinDimensional {
  int laenge();
}

public interface ZweiDimensional extends EinDimensional {
  int breite();
}

public interface DreiDimensional extends ZweiDimensional {
  int hoehe();
}
```

Beispiel: Vererbung von Interfaces

```
public class Auto2 implements DreiDimensional {

  private int laenge;
  private int hoehe;
  private int breite;
  // weitere Attribute ...

  public int laenge() {
    return this.laenge;
  }

  public int hoehe() {
    return this.hoehe;
  }

  public int breite() {
    return this.breite;
  }
}
```

Interfaces und abstrakte Klassen: Unterschiede

- Offensichtlich haben Interfaces und abstrakte Klassen ähnliche Eigenschaften, z.B. können keine Objekte von Interfaces und abstrakten Klassen instanziiert werden.
- Im Unterschied zu Interfaces können abstrakte Klassen aber auch konkrete Methoden enthalten, d.h. Methoden mit Rumpf.
- Es ist sogar möglich, dass abstrakte Klassen nur konkrete Methoden spezifizieren; mit dem Schlüsselwort `abstract` in der Klassendeklaration ist die Klasse dennoch abstrakt (und daher nicht instanziierbar).
- Alle Methoden eines Interfaces sind dagegen immer abstrakt.
- Abstrakte Klassen dürfen im Gegensatz zu Interfaces Attribute spezifizieren.

Interfaces und abstrakte Klassen: Verwendung

- Welches Konzept sollte man nun verwenden?
- Interfaces sind flexibler und können in unterschiedlichen Klassenhierarchien verwendet werden, da sie keinerlei Möglichkeiten bereitstellen, Implementierungsdetails festzulegen, sondern lediglich abstrakte Funktionalitäten.
- Implementierungsdetails können allerdings in der Dokumentation vorgeschrieben werden – ob diese Vorschrift wirklich eingehalten wird, kann aber der Compiler nicht überprüfen.
- Abstrakte Klassen bieten darüber hinaus die Möglichkeit, einen Teil der Implementierungsdetails bereits festzulegen und damit die Wiederverwendbarkeit von Code-Teilen zu unterstützen, wo dies möglich/sinnvoll ist.

Interfaces und Mehrfachvererbung

- Wie bereits angedeutet, kann man mit Hilfe von Interfaces auch Mehrfachvererbung in Java modellieren.
- Beispiel: `AmphibienFahrzeug` wird von `WasserFahrzeug` und `Landfahrzeug` abgeleitet.
- Problem war, dass beide Vaterklassen eine Methode `getPS` implementieren, die nicht überschrieben wird.
- Falls die Methode `getPS` für ein Objekt der Klasse `AmphibienFahrzeug` aufgerufen wird, kann nicht entschieden werden, welche ererbte Version ausgeführt werden soll.

Interfaces und Mehrfachvererbung

- Lösung: Nur eine der Vaterklassen wird als Klasse realisiert, alle anderen (in unserem Fall nur eine) werden als Interfaces angegeben.
- Die (von der übriggebliebenen Vater-Klasse) abgeleitete Klasse muss nun zusätzlich zur Vererbung alle Interfaces implementieren.

Interfaces und Mehrfachvererbung

- Beispiel: `WasserFahrzeug` wird als Interface spezifiziert, wohingegen `Landfahrzeug` eine Klasse ist.
- Die Klasse `AmphibienFahrzeug` ist von `Landfahrzeug` abgeleitet und implementiert `WasserFahrzeug`.
- Die Methode `getPS` muss (falls sie im Interface `WasserFahrzeug` verlangt wird) in der abgeleiteten Klasse implementiert werden.
- Wird die Methode allerdings bereits von der Vaterklasse ererbt, muss `getPS` in `AmphibienFahrzeug` *nicht* implementiert werden (kann aber selbstverständlich überschrieben werden).
- In beiden Fällen ist die Methode `getPS` für Objekte der Klasse `AmphibienFahrzeug` eindeutig bestimmt.

Interfaces und Mehrfachvererbung

- Achtung: Die Realisierung von Mehrfachvererbung in Java mittels Interfaces schränkt das eigentliche Konzept der Mehrfachvererbung ein.
- Offensichtlich ist es z.B. nicht möglich, Objekte aller Vaterklassen zu erzeugen.
- In unserem Beispiel ist es nicht möglich, Objekte vom Typ `WasserFahrzeug` zu instanziieren.
- Ebenfalls eingeschränkt ist die Möglichkeit, Funktionalitäten in Vaterklassen zu implementieren und dann zu vererben (Wiederverwendung).
- Diese Einschränkungen sind allerdings nötig, um die oben angesprochenen Probleme der Mehrfachvererbung zu lösen.

Exkurs: Marker-Interfaces

- Interfaces, die weder Methoden noch Konstanten definieren, also einen leeren Rumpf haben, werden *Marker-Interfaces* (auch: *Flag-Interfaces*, *Tagging-Interfaces*) genannt.
- Marker-Interfaces sind dazu gedacht, gewisse (teilweise abstrakte) Eigenschaften von Objekten sicher zu stellen, die typischerweise im Kommentar des Interfaces spezifiziert sind.
- Implementiert eine Klasse ein Marker-Interface, sollte sich der Implementierer an diese Spezifikationen halten.
- Der Programmierer signalisiert durch `implements <MarkerInterface>`, dass seine Klasse die Eigenschaften hat, die im angegebenen Marker-Interface spezifiziert sind.
- Vorsicht bei geerbten Marker-Interfaces!

Beispiel: Das Interface `Cloneable`

- Zur Erinnerung: `clone()` erzeugt eine Kopie des aktuellen Objekts.
- Die ursprüngliche Fassung von `clone()` in `Object` erzeugt eine flache Kopie.
- Damit ist nicht sichergestellt, dass die Kopie unabhängig vom ursprünglichen Objekt ist!

Beispiel: Das Interface `Cloneable`

- Die Java-API stellt das Marker-Interface `Cloneable` zur Verfügung, das für die Methode `clone()` eine spezielle Eigenschaft spezifiziert.
- Implementiert eine Klasse das Interface `Cloneable`, so zeigt der Implementierer an, dass die Methode `clone()` eine tiefe Kopie erzeugt.
- Die Methode `clone()` muss dazu entsprechend überschrieben werden.

Zusammenfassung

Sie kennen jetzt:

- das Konzept der Interfaces als Möglichkeit abstrakte Funktionalitäten (d.h. komplett ohne Implementierungsdetails) zu spezifizieren,
- die Umsetzung von Interfaces in Java, insbesondere das Definieren, Implementieren und Verwenden von Interfaces,
- den Unterschied zwischen Interfaces und abstrakten Klassen,
- die Möglichkeit, Mehrfachvererbung mit Hilfe von Interfaces in Java zu simulieren,
- das Konzept der Marker-Klassen zur Zusicherung von gewissen Eigenschaften von Objekten.