# Einführung in die Programmierung mit Java

## Teil 8: Vererbung

Martin Hofmann  Steffen Jost

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

5. Dezember 2017

LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

# Teil 8: Vererbung

# Motivation: Vererbung

Oft hat man sehr ähnliche Klassen, welche ähnlichen Code enthalten. Die Duplikation von Code ist aber generell schlecht (erschwert Wartung, Korrektheitsbeweise, etc.).

Java bietet **Vererbung** an, um die Funktionalität einer Klasse ohne Duplikation von Code zu *erweitern* oder auch zu *spezialisieren*.

Die Erben einer Klasse behalten die Funktionalität des Vorfahren, aber können neue Funktionalität anbieten.

# Beispiel: Bankkonto



```
public class Bankkonto {
  private double kontostand;
  private String besitzer;

  public Bankkonto() { this("unbekannt", 5.0); }
  public Bankkonto(String besitzer, double betrag) {
        kontostand    = betrag;
        this.besitzer = besitzer;  }
  public double  getKontostand() { return kontostand; }
  public boolean istUeberzogen() { return kontostand < 0.0; }
  public void einzahlen(double betrag){ kontostand += betrag; }
  public void abheben  (double betrag){ kontostand -= betrag; }
  public void ueberweisen(double betrag, Bankkonto empfaenger){
    this.abheben(betrag);
    empfaenger.einzahlen(betrag);
} }
```

*Hinweis:* Am Anfang eines Konstruktors ruft `this(...)` einen anderen Konstruktor der Klasse mit anderen Argumenten auf.

# Beispiel: Bankkonto beerben

```
public class Sparkonto extends Bankkonto {
  /* Neue Instanzvariablen, z.B. Zinssatz */
  /* Neue Methoden, z.B. Zinsen berechnen */
  /* Neue Konstruktoren, z.B. mit Angabe des Zinssatzes */
}
```

- Alle bisherigen Methoden bleiben verfügbar, als hätte man sie in die Definition von `Sparkonto()` hineinkopiert.
- Bisherige private Instanzvariablen sind ebenfalls vorhanden, sind aber nicht sichtbar in hier neu definierten Methoden.
- Für Konstruktoren gelten besondere Regeln.

# Beispiel: Sparkonto

Ein `Sparkonto` *ist ein* `Bankkonto` mit zusätzlicher Funktionalität:

```
public class Sparkonto extends Bankkonto {
  private double zinssatz;

  public Sparkonto(double satz) {
    zinssatz = satz;
  }

  public void zinsenAnrechnen() {
    double zinsen=this.getKontostand()*zinssatz/100.0;
    this.einzahlen(zinsen); // Aufruf geerbte Methode
  }
}
```



Man könnte hier überall "`this.`" auch weglassen.

# Verwendung

```
Sparkonto johannasSpar = new Sparkonto(1.25);

johannasSpar.einzahlen(200.0);
johannasSpar.zinsenAnrechnen();
System.out.println("Johannas Sparkonto: " +
                   johannasSpar.getKontostand());
```

Ausgabe:

```
Johannas Sparkonto: 207.5625
```

*Erklärung:* $(200 + 5) \cdot (1 + \frac{1.25}{100}) = 207.5625$
Der Konstruktor `Bankkonto()` setzt den Kontostand auf 5.00 (Geschenk der Sparkasse); der geerbte Konstruktor wird also auch noch aufgerufen!

# Funktionsweise

Beim Aufruf

```
johannasSpar.zinsenAnrechnen();
```

wird folgender Code ausgeführt:

```
double zinsen = johannasSpar.getKontostand() *
                                    zinssatz / 100.0;
johannasSpar.einzahlen(zinsen);
```

# Terminologie

- `Bankkonto` ist *die* **Oberklasse** (**superclass**) von `Sparkonto`.
- `Sparkonto` **erbt von** (**inherits from**) `Bankkonto`.
- `Sparkonto` ist *eine* **Unterklasse** (**subclass**) von `Bankkonto`.

In Java hat jede Klasse nur genau eine Oberklasse.

Eine Klasse kann aber mehrere Unterklassen haben.
*Beispiel:* `Bankkonto` könnte noch weitere Unterklassen haben, wie etwa `Girokonto`, `Tagesgeldkonto`, etc.

Auch von Unterklassen kann man weiter vererben. Zum Beispiel ein `Prämiensparkonto` als Unterklasse von `Sparkonto`, etc.

# Vererbung von Instanzvariablen

- Alle Instanzvariablen der Oberklasse werden vererbt;
- `private`-Instanzvariablen sind in der Unterklasse vorhanden, aber nicht mehr sichtbar!

Allerdings können ererbte Methoden der Oberklasse auf alle vererbten Instanzvariablen zugreifen!

Beispiel Ein Sparkonto hat also die Instanzvariablen `kontostand`, `besitzer` und `zinssatz`. Neu geschriebene Methoden können aber nur `zinssatz` direkt beeinflussen.

Hinweis Man kann eine Instanzvariable als `protected` kennzeichnen. Dann ist sie auch in den Unterklassen sichtbar (und in allen Klassen derselben package). Von Verwendung wird aus diversen Gründen abgeraten!

# Konstruktoren der Unterklasse

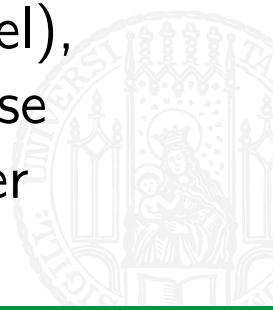
- Konstruktoren werden nicht vererbt.
- In einem Konstruktor kann aber ein Konstruktor der Oberklasse mit `super(...)` aufgerufen werden.

Man sollte einen Konstruktor der Oberklasse immer zu Beginn des Konstruktors der Unterklasse aufrufen, damit die ererbten privaten Instanzvariablen geeignet initialisiert werden!

Beispiel:

```
public Sparkonto(String besitzer, double satz) {
    super(besitzer, 0.0);
    zinssatz = satz;
}
```

Geschieht dies nicht (wie im ursprünglichen `Sparkonto`-Beispiel), so wird der Defaultkonstruktor (ohne Parameter) der Oberklasse automatisch eingefügt. Gibt es keinen Defaultkonstruktor in der Oberklasse er, so gibt es einen Compilerfehler!



# Überschreiben

- `public`-Methoden werden vererbt.
- `private`-Methoden werden nicht vererbt; geerbte `public`-Methoden rufen diese aber weiterhin auf.
- Geerbte Methoden können überschrieben werden

wie bei Interface-Implementierung

## Überschreiben

Definiert man in der Unterklasse eine Methode, die es in der Oberklasse schon gibt (Name und Parameterzahl und -typen stimmen überein), so wird die ererbte Methode **überschrieben** (engl. **overriding**). Die neudefinierte Methode ersetzt die alte *überall*, d.h. auch in unveränderten *geerbten Methoden* **(!!!)**, welche die überschriebene Methode verwenden.

Unbedingt `@Override` Annotation vor der Definition einer überschriebenen Methode schreiben, damit uns der Kompiler vor versehentlichem Overloading warnen kann!

# Beispiel: Girokonto

```
public class Girokonto extends Bankkonto {
    private int anzahlTransaktionen;
    private double gebuehrensatz;

    public Girokonto(double satz) {
        super();
        anzahlTransaktionen = 0;
        gebuehrensatz = satz;
    }

    @Override
    public void einzahlen(double betrag) {
        super.einzahlen(betrag);
        anzahlTransaktionen++;
    }
                    ⋮
```

```
    ⋮

    @Override
    public void abheben(double betrag) {
        super.abheben(betrag);
        anzahlTransaktionen++;
    }

    public void gebuehrenAbziehen() {
        double gebuehren = anzahlTransaktionen
                                        * gebuehrensatz;
        super.abheben(gebuehren);
        anzahlTransaktionen = 0;
    }
}
```

- Jede Ein-/Auszahlung erhöht nun einen internen Zähler!
- `super` ermöglicht Aufruf der geerbten Methode.

# Bemerkungen

- Die Pseudovariable `super` bezeichnet das aktuelle Objekt aufgefasst als Objekt der Oberklasse.
- `super.super.foo()` ist nicht erlaubt!
  $\Rightarrow$ Jede Klasse hat nur eine Oberklasse!
- Hätten wir in `gebuehrenAbziehen` statt `super.abheben(betrag);` einfach nur `this.abheben(gebuehren);` geschrieben, so wären für das Abziehen der Gebühren gleich wieder welche fällig geworden.
- Wir können in `Girokonto` nicht schreiben

  ```
  kontostand = kontostand - gebuehren;
  ```

  da `kontostand` nicht sichtbar ist.
- Wir könnten natürlich eine neue Instanzvariable mit Namen `kontostand` deklarieren, dann hätte aber jedes Objekt zwei verschiedene Instanzvariablen des Namens `kontostand` ↯

# Subtyping

Ein Objekt der Unterklasse kann immer dort verwendet werden, wo ein Objekt der Oberklasse erwartet wird.

Ganz analog zu Interface-Typen!

Beispiel:

```
Sparkonto johannasSpar = new Sparkonto(1.25);
Bankkonto matthiasGiro = new Bankkonto();
matthiasGiro.ueberweisen(200, johannasSpar);
```

*Zur Erinnerung:*

```
public void ueberweisen(double betrag,Bankkonto empfaenger)
```

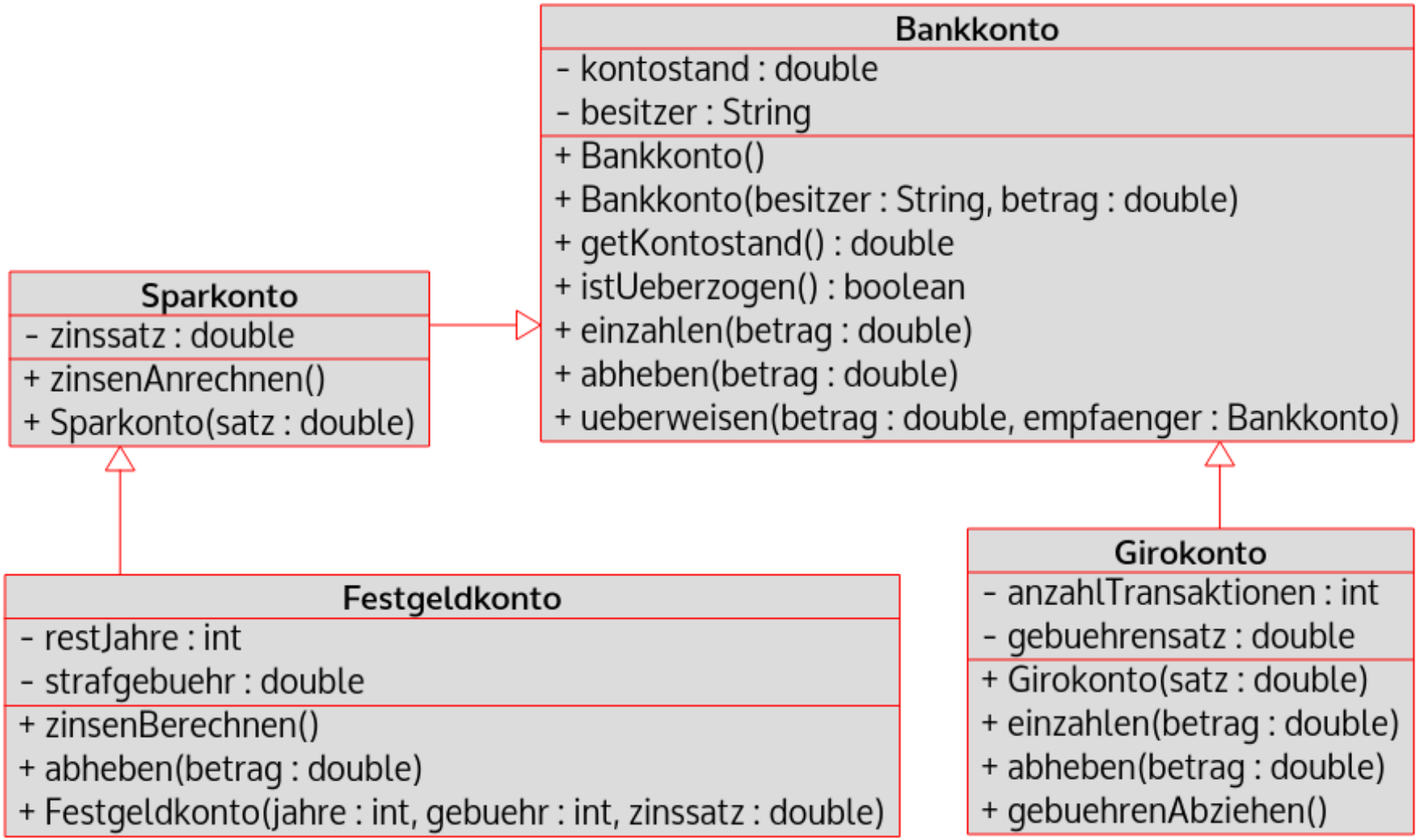
Zwischen der Unterklasse und der Oberklasse besteht eine **"ist-ein"**-Beziehung (engl. **"is a"**-relationship).

Beispiel: `Bankkonto`-Array kann auch `Sparkonten` enthalten!

In den Anwendungen ergibt sich durch Subtyping oft eine komplexe **Klassenhierarchie**.

# Klassenhierarchie visualisiert

Im nächsten Vorlesungskapitel lernen wir **Unified Modelling Language (UML)** zur Visualisierung solcher Klassenhierarchien:



# Geschachtelte Vererbung

Ist $B$ eine Unterklasse von $A$, so kann man ohne weiteres eine Unterklasse $C$ von $B$ definieren.

Jedes Objekt der Klasse $C$ ist dann automatisch ein Objekt der Klasse $B$ und als solches auch ein Objekt der Klasse $A$.

Die Oberklasse von $C$ ist aber nur $B$.

# Beispiel: Festgeldkonto

```
public class Festgeldkonto extends Sparkonto {
  private int    restJahre;
  private double strafgebuehr;
  @Override
  public void zinsenBerechnen() {
      restJahre = restJahre-1;
      super.zinsenAnrechnen();
  }
  @Override
  public void abheben(double betrag) {
      if (restJahre > 0)
          super.abheben(strafgebuehr);
      super.abheben(betrag);
  }
  public Festgeldkonto(int jahre,int gebuehr,double zinssatz){
      super(zinssatz);
      strafgebuehr = gebuehr;
      restJahre = jahre;
} }
```

# Dynamische Bindung

Jedes Objekt gehört trotz Subtyping zu einer festen Klasse. Welche Methode bei einem Objekt aufgerufen wird, richtet sich immer nach seiner echten Klasse, auch dann, wenn es per Subtyping als Objekt der Oberklasse benutzt wird. Dies bezeichnet man als **dynamische Bindung** (engl. **dynamic dispatch**).

## Beispiel

```
Bankkonto matthiasGiro = new Girokonto(0.25);
Sparkonto johannasSpar = new Sparkonto(200.0);
johannasSpar.ueberweisen(100.0, matthiasGiro);
```

Hier fallen bei `matthiasGiro` Gebühren an, obwohl die Methode `einzahlen` in `Bankkonto` eigentlich keine Gebühren beaufschlagt. Es wird aber die überschriebene Methode verwendet!

# Typecast und instanceof

Hat ein Ausdruck den Typ einer Oberklasse, so kann man ihn explizit auf die Unterklasse konvertieren:

```
Bankkonto matthiasGiro = new Girokonto(0.25);
Girokonto y = (Girokonto) matthiasGiro;
```

Wird eine **Typkonversion** (**type cast**) der Form `(A)e` ausgewertet, so wird überprüft, ob die tatsächliche Klasse von `e` Unterklasse (evtl. über mehrere Ecken) von `A` ist:

*Falls ja:* Auswertung wird fortgesetzt

*Falls nein:* Programmabbruch

Der Typ des Ausdrucks "`(A)e`" ist `A`

Der Ausdruck "`e instanceof A`" hat den Wert `true` genau dann, wenn die tatsächliche Klasse von `e` eine Unterklasse von `A` ist.

# Die Klasse Object

Die Klasse `Object` steht an der Spitze jeder Klassenhierarchie.
Sie definiert unter anderem die Methoden:

```
String  toString()    /* in einen String konvertieren */
boolean equals(Object other) /* auf Gleichheit testen */
Object  clone()         /* Kopie des Objektes anlegen */
                            ⋮
```

Diese Methoden kann man in einer eigenen Klasse überschreiben.

Dies sollte man auch tun, damit diese Methoden immer sinnvoll funktionieren!

# Übungen I

Was ist `b.getKontostand()` am Ende ?

```
Sparkonto b = new Sparkonto(10);
b.einzahlen(4995);
b.abheben(b.getKontostand() / 2);
b.zinsenAnrechnen();
```

# Übungen II

Was sollte hier Unterklasse sein, was Oberklasse:

- Angestellter-Vorgesetzter
- Polygon- Dreieck
- Doktorstudent-Student
- Person-Student
- Angestellter-Doktorstudent
- Bankkonto-Girokonto
- Fahrzeug-Pkw
- Fahrzeug-Van
- Pkw-Van
- Lkw-Fahrzeug

# Übungen III

Die Klasse `Sub` sei Unterklasse von `Sandwich`. Welche der folgenden Zuweisungen sind erlaubt?

```
Sandwich x = new Sandwich();
Sub y = new Sub();
x = y;
y = x;
y = new Sandwich();
x = new Sub();
```

# Übungen IV

```
class A {
  public void exec() { this.doIt(); }
  public void doIt() { System.out.println("A"); }
}
class B extends A {
  public void doit() { System.out.println("B"); }
}
class C extends B {
  public void doIt() { System.out.println("C"); }
}
```

Was wird ausgegeben bei folgendem Code?

```
A a = new B();
B b = new C();
a.exec();
b.exec();
```

# Behavioural Subtyping

Hat man in einer Oberklasse Methoden durch Vor- und Nachbedingungen spezifiziert, so sollten überschreibende Versionen davon auch dieser Spezifikation genügen.

Der Vorteil ist dann, dass Spezifikationen für sämtliche geerbte Methoden fortgelten.

Ebenso sollten Invarianten, die in der Oberklasse vereinbart wurden, auch von den Methoden der Unterklasse eingehalten werden.

Liegen all diese Bedingungen vor, so sagt man, die Unterklasse sei ein **behavioural subtype** der Oberklasse.

# Verstärkung von Invarianten bei Vererbung

- Eine Unterklasse kann eine stärkere Invariante verlangen als die Oberklasse
  z.B. positive `breite` & `hoehe` müssen auch noch gleich sein
- Die Vererbungsregeln für Konstruktoren unterstützen dies.

Beispiel: Quadrat als Unterklasse von Rechteck

```
public class Rechteck {
    private int x, y, breite, hoehe;
    public Rechteck(int x, int y, int breite, int hoehe) {
        this.x=x;this.y=y;this.breite=breite;this.hoehe=hoehe;
    }
    public void verschieben(int dx, int dy) {
        x=x+dx;y=y+dy;
}}
public class Quadrat extends Rechteck {
    public Quadrat(int x, int y, int seite) {
        super(x,y,seite,seite);
}}
```

# Verstärkung von Invarianten bei Vererbung

- Eine Unterklasse kann eine stärkere Invariante verlangen als die Oberklasse
  - z.B. positive `breite` & `hoehe` müssen auch noch gleich sein
- Die Vererbungsregeln für Konstruktoren unterstützen dies.

**Mögliches Problem:**
Was, wenn `Rechteck` Änderung von `breite` und `hoehe` erlaubt?

Beispiel: Quadrat als Unterklasse von Rechteck

```
public class Rechteck {
    private int x, y, breite, hoehe;
    public Rechteck(int x, int y, int breite, int hoehe) {
        this.x=x;this.y=y;this.breite=breite;this.hoehe=hoehe;
    }
    public void verschieben(int dx, int dy) {
        x=x+dx;y=y+dy;
}}
public class Quadrat extends Rechteck {
    public Quadrat(int x, int y, int seite) {
        super(x,y,seite,seite);
}}
```

# Abstrakte Klassen und Methoden

Man kann eine Methodendeklaration in einer Klasse mit dem Schlüsselwort `abstract` kennzeichnen und keine Implementierung angeben.

Die gesamte Klasse muss dann auch das Schlüsselwort `abstract` tragen. Man kann dann von dieser Klasse keine Instanzen erzeugen (`new` ist also verboten).

Allerdings kann man von der Klasse erben und dann die "abstrakten" Methoden konkret implementieren.

Verbleiben abstrakte Methode, muss die Klasse abstrakt bleiben.

Abstrakte Klassen wurden inzwischen von Interfaces weitestgehend abgelöst. Ein Interface bietet sich meist an, wenn Instanzvariablen für die konkreten Methoden irrelevant sind.

# Beispiel AbstraktesBankkonto I

```
public abstract class AbstraktesBankkonto {
   private static int naechsteNummer = 0;
   private int kontonummer;

   public AbstraktesBankkonto() {
      kontonummer = naechsteNummer;
      naechsteNummer++;
   }

   public int getKontonummer() {
        return kontonummer;
   }

   public abstract void writeLog(String s);
```

# Beispiel AbstraktesBankkonto I

```
                    ⋮

    public void abheben(double betrag) {
         writeLog("Abhebung: "+betrag);
    }

    public void einzahlen(double betrag) {
         writeLog("Einzahlung: "+betrag);
    }

    public void ueberweisen(AbstraktesBankkonto b,double betrag){
         this.abheben(betrag);b.einzahlen(betrag);
    }
}
```

# Beispiel AbstraktesBankkonto II

```
public class Bankkonto extends AbstraktesBankkonto {
    private double kontostand;

    public double getKontostand() {
        return kontostand;
    }

    public Bankkonto() {
        super();
        kontostand = 0;
    }
```

⋮

```
    ⋮

    @Override
    public void writeLog(String s) {
        System.out.println
            ("Kontonr.: " + getKontonummer() + "\n" +
             s +
             "\nNeuer Kontostand: " + getKontostand());
    }
    @Override
    public void einzahlen(double betrag) {
        kontostand += betrag;
        super.einzahlen(betrag);
    }
    @Override
    public void abheben(double betrag) {
        kontostand -= betrag;
        super.abheben(betrag);
    }
}
```

# Verwendung

```
b.einzahlen(3);
c.einzahlen(6);
b.ueberweisen(c,4);
```

Ausgabe:

```
Kontonr.: 0
Einzahlung: 3.0
Neuer Kontostand: 3.0
Kontonr.: 1
Einzahlung: 6.0
Neuer Kontostand: 6.0
Kontonr.: 0
Abhebung: 4.0
Neuer Kontostand: -1.0
Kontonr.: 1
Einzahlung: 4.0
Neuer Kontostand: 10.0
```

# Zusammenfassung: Vererbung

- Man kann zu einer gegebenen Klasse Unterklassen definieren:

  `class` *NameDerUnterklasse* `extends` *NameDerOberKlasse*{
  *neue Konstruktoren*
  *neue Instanzvariablen*
  *neue Methoden*}

- Instanzvariablen, Methoden der Oberklasse werden geerbt.
- Private Instanzvariablen der Oberklasse sind in der Unterklasse nicht sichtbar.
- `public` und `protected` Methoden der Oberklasse kann man überschreiben. Die neue Definition ersetzt die alte überall, auch in Definitionen von Methoden der Oberklasse.
- Mit `super` kann man auf Methoden und Konstruktoren der Oberklasse zugreifen.
- Konstruktoren werden nicht vererbt.

  Ausnahme: Defaultkonstruktor ohne Argumente.

- Vererbung wird benutzt um "*is a*"-Beziehungen softwaretechnisch zu realisieren.
- `Object` ist automatisch Oberklasse jeder Klasse. Die Methoden `equals, toString` der Klasse `Object` können überschrieben werden.
- Behavioural subtyping: Spezifikationen aus der Oberklasse werden von der Unterklasse respektiert.
- Abstrakte Methoden in abstrakten Klassen werden nicht implementiert, sie müssen in konkreten Unterklassen überschrieben werden.
- Der Einsatz von Interfaces hat inzwischen Vererbung in vielen Fällen ersetzt.