

# Kapitel 7: Das Vererbungsprinzip

Prof. Dr. David Sabel

Lehr- und Forschungseinheit für Theoretische Informatik

Institut für Informatik, LMU München

WS 2018/19

Stand der Folien: 9. Januar 2019

Die Inhalte dieser Folien basieren – mit freundlicher Genehmigung – tlw. auf Folien von Prof. Dr. Rolf Hennicker aus dem WS 2017/18 und auf Folien von PD Dr. Ulrich Schöpp aus dem WS 2010/11



- Das **Vererbungsprinzip** der objektorientierten Programmierung verstehen
- Das Vererbungsprinzip in Java umsetzen können
- Folgende Begriffe kennenlernen und anwenden können:
  - Ober- und Unterklassen
  - Subtyping
  - Überschreiben von Methoden
  - Dynamische Bindung
- Die Klasse `Object` kennenlernen
- Abstrakte Klassen und Interface kennenlernen

# Beispiel: Klasse für Bankkonten

---

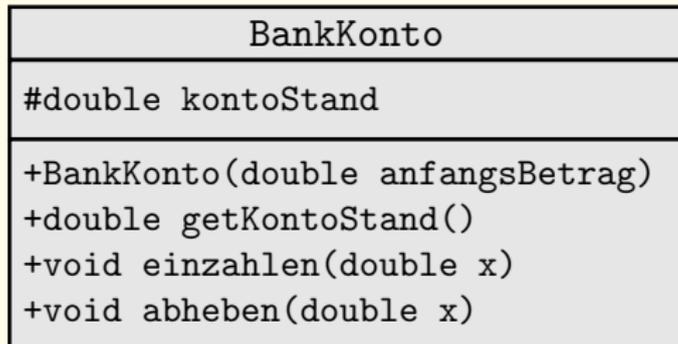
```
public class BankKonto {  
  
    protected double kontoStand;  
  
    public BankKonto(double anfangsBetrag) {  
        this.kontoStand = anfangsBetrag;  
    }  
  
    public double getKontoStand() {  
        return this.kontoStand;  
    }  
  
    public void einzahlen (double x) {  
        this.kontoStand += x;  
    }  
  
    public void abheben(double x) {  
        this.kontoStand -= x;  
    }  
}
```

## Beispiel: Klasse für Bankkonten (2)

```
public class BankKonto {  
    protected double kontoStand;  
  
    public BankKonto(double anfangsBetrag) {  
        this.kontoStand = anfangsBetrag;  
    }  
  
    public double getKontoStand() {  
        return this.kontoStand;  
    }  
  
    public void einzahlen(double betrag) {  
        this.kontoStand += betrag;  
    }  
  
    public void abheben(double betrag) {  
        this.kontoStand -= betrag;  
    }  
}
```

### Grafische Darstellung in UML

(Unified Modeling Language)



**Vererbung** ist ein Mechanismus zur Implementierung von Klassen durch **Erweiterung existierender Klassen**

Beispiel:

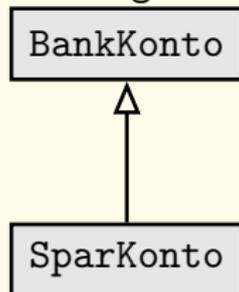
- Die Klasse `BankKonto` stellt die Grundfunktionalität eines Bankkontos bereit.
- Möchte man verschiedene spezielle Arten von Bankkonten implementieren, so kann die Klasse `BankKonto` durch **Vererbung** erweitern:
  - `SparKonto`: hat zusätzlich einen Zinssatz, Zinsen können gut geschrieben werden
  - `GiroKonto`: bei jeder Transaktion werden Gebühren berechnet

# Sparkonto in Java

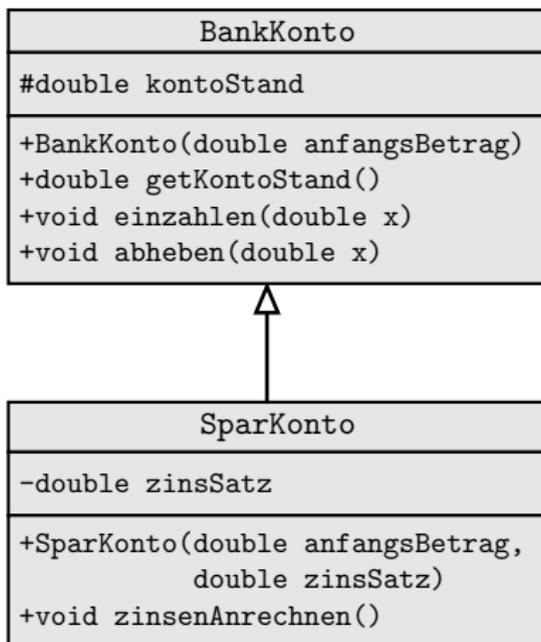
```
public class SparKonto extends BankKonto {  
    private double zinsSatz;  
  
    public SparKonto(double anfangsBetrag, double zinsSatz) {  
        super(anfangsBetrag);  
        this.zinsSatz = zinsSatz;  
    }  
  
    public void zinsenAnrechnen() {  
        double zinsen = this.kontoStand * this.zinsSatz / 100.0;  
        this.kontoStand += zinsen;  
    }  
}
```

- Ein SparKonto **erbt** alle Attribute und Methoden eines BankKontos und fügt eigene hinzu.

Darstellung in UML



# Grafische Darstellung der Vererbung in UML



# Vererbung und Konstruktoren

---

- Im Gegensatz zu Attributen und Methoden werden **Konstruktoren nicht vererbt**
- Die Konstruktoren der Oberklasse können jedoch in neu definierten Konstruktoren der Unterklasse aufgerufen werden:

Aufruf des Konstruktors der Oberklasse:

```
super(); // parameterloser Konstruktor
```

bzw.

```
super(a1, ..., an); //Konstruktor mit Parametern
```

Dieser Aufruf **muss die erste Anweisung** des Konstruktors in der Unterklasse sein.

# Falsche Aufrufe des Oberklassen-Konstruktors (1)

```
1 public class SparKontoFalsch extends BankKonto {
2     private double zinsSatz;
3
4     public SparKontoFalsch(double anfangsBetrag, double zinsSatz) {
5         super(anfangsBetrag);
6         this.zinsSatz = zinsSatz;
7         super(anfangsBetrag);
8     }
9
10    public void zinsenAnrechnen() {
11        double zinsen = this.kontoStand * this.zinsSatz / 100.0;
12        this.kontoStand += zinsen;
13    }
14 }
```

Kompilieren:

```
> javac SparKontoFalsch.java
SparKontoFalsch.java:7: error: call to super must be first
                        statement in constructor
```

```
    super(anfangsBetrag);
        ^
```

```
1 error
```

## Falsche Aufrufe des Oberklassen-Konstruktors (2)

```
1 public class SparKontoFalsch2 extends BankKonto {
2     private double zinsSatz;
3
4     public SparKontoFalsch2(double anfangsbetrag, double zinsSatz) {
5         this.zinsSatz = zinsSatz;
6     }
7     ...
```

Kompilieren:

```
> javac SparKontoFalsch2.java
SparKontoFalsch2.java:4: error: constructor BankKonto in class BankKonto cannot
    be applied to given types;
```

```
public SparKontoFalsch2(double anfangsbetrag, double zinsSatz) {
    ~
    required: double
    found: no arguments
    reason: actual and formal argument lists differ in length
1 error
```

Java fügt **automatisch** `super()` hinzu,  
aber es gibt keinen nullstelligen Konstruktor für `BankKonto`!

## Beispiel: Benutzung von SparKonto

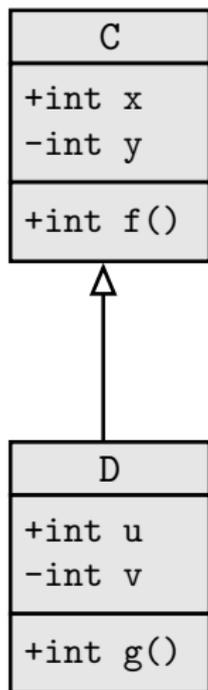
---

```
public class SparKontoTest {
    public static void main(String[] arg){
        SparKonto sk = new SparKonto(450.0,1.2); //450 EUR, 1.2%
            Zinsen
        sk.einzahlen(50.0); //geerbte Methode
        sk.zinsenAnrechnen(); //neue Methode
        System.out.println(sk.getKontoStand());
            //geerbte Methode getKontoStand()
    }
}
```

Kompilieren und Ausführen:

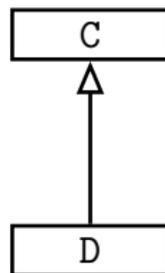
```
> javac SparKontoTest.java
> java SparKontoTest
506.0
```

# Vererbung von Attributen



- Attribute von C: `x` und `y`
- Attribute von D: `u`, `v`, `x` und `y` (`x` und `y` sind geerbt)
- Methoden von C: `f`
- Methoden von D: `g` und `f` (`f` ist geerbt)
- Wenn C Subklasse einer anderen Klasse B ist, dann besitzen C und D auch noch alle Attribute und Methoden von B.
- Man kann von D aus **nicht direkt auf die privaten Attribute** von C zugreifen (hier `y`), sondern nur mittels nicht privater (geerbter) getter-Methoden von C.
- Auf geschützte (`protected`) Attribute kann man in Subklassen schon zugreifen.

- 1 Vererbung drückt eine „**is-a**“-Beziehung aus.
- 2 Die Objekte der Klasse D können wie Objekte der Klasse C benutzt werden.
- 3 Eine Referenz auf ein Objekt der Unterklasse D kann überall dort eingesetzt werden, wo eine Referenz auf ein Objekt der Oberklasse C erwartet wird.



Wir sagen: Der Klassentyp D ist ein **Subtyp** von C  
(D „passt zu“ C)

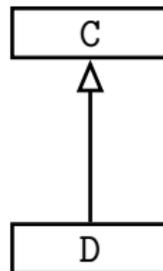
## Subtyping: Beispiel

---

Weise der Variablen `c` mit Typ `C` einen Ausdruck des Subtyps `D` zu:

```
C c = new D();
```

Die Variable `c` verweist dann auf ein Objekt der Klasse `D` (und damit auf ein Objekt von `C`).



## Subtyping: Beispiel

Weise der Variablen `c` mit Typ `C` einen Ausdruck des Subtyps `D` zu:

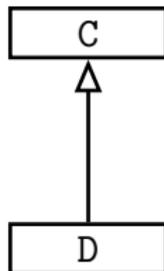
```
C c = new D();
```

Die Variable `c` verweist dann auf ein Objekt der Klasse `D` (und damit auf ein Objekt von `C`).

Umgekehrt: Zuweisung eines Objekts der Oberklasse `C` zu einer Variablen vom Typ `D` ist **nicht möglich**, d.h.

```
D d = new C();
```

**ist verboten**



## Beispiel: Bankkonto und Sparkonto

---

```
1 public class SparKontoTest2 {
2     public static void main(String[] args) {
3         SparKonto sparkonto = new SparKonto(450.0, 1.2);
4         BankKonto konto = sparkonto;
5         konto.einzahlen(50);
6         konto.zinsenAnrechnen();
7         System.out.println(konto.getKontoStand());
8         System.out.println(sparkonto.getKontoStand());
9     }
10 }
```

## Beispiel: BankKonto und SparKonto

```
1 public class SparKontoTest2 {
2     public static void main(String[] args) {
3         SparKonto sparkonto = new SparKonto(450.0, 1.2);
4         BankKonto konto = sparkonto;
5         konto.einzahlen(50);
6         konto.zinsenAnrechnen();
7         System.out.println(konto.getKontoStand());
8         System.out.println(sparkonto.getKontoStand());
9     }
10 }
```

Fehler, da `zinsenAnrechnen()` nur für `SparKonto` definiert ist:

```
> javac SparKontoTest2.java
```

```
SparKontoTest2.java:6: error: cannot find symbol
```

```
    konto.zinsenAnrechnen();
```

```
        ^
```

```
symbol:   method zinsenAnrechnen()
```

```
location: variable konto of type BankKonto
```

```
1 error
```

## Beispiel: Bankkonto und Sparkonto (2)

---

```
1 public class SparKontoTest3 {
2     public static void main(String[] args) {
3         SparKonto sparkonto = new SparKonto(450.0, 1.2);
4         Bankkonto konto = sparkonto;
5         konto.einzahlen(50);
6         SparKonto nocheinSparkonto = konto;
7         System.out.println(konto.getKontoStand());
8         System.out.println(sparkonto.getKontoStand());
9     }
10 }
```

## Beispiel: BankKonto und SparKonto (2)

```
1 public class SparKontoTest3 {
2     public static void main(String[] args) {
3         SparKonto sparkonto = new SparKonto(450.0, 1.2);
4         BankKonto konto = sparkonto;
5         konto.einzahlen(50);
6         SparKonto nocheinsparkonto = konto;
7         System.out.println(konto.getKontoStand());
8         System.out.println(sparkonto.getKontoStand());
9     }
10 }
```

Fehler, da konto nicht unbedingt ein SparKonto ist:

```
> javac SparKontoTest3.java
```

```
SparKontoTest3.java:6: error: incompatible types:
```

```
    BankKonto cannot be converted to SparKonto
```

```
    SparKonto nocheinsparkonto = konto;
```

```
    ^
```

1 error

# Typkonversion durch Einengung

- In den beiden Beispielen wussten **wir** zwar, dass konto auf ein Sparkonto verweist, aber der Java-Compiler weiß es nicht.
- In **solchen Situationen** kann man den Typ ausdrücklich konvertieren:

```
Sparkonto nocheinsparkonto = (Sparkonto) konto;  
nocheinsparkonto.zinsenAnrechnen();
```

- Bei der Auswertung einer Typkonversion (D)e wird **geprüft**, ob das Objekt auf das e im aktuellen Zustand verweist, tatsächlich Instanz von D oder einer Subklasse von D ist.

Ist dies nicht der Fall, so kommt es zu einem **Laufzeitfehler** (ClassCastException)

- Hinweis: Typkonversionen zwischen Klassentypen sollten möglichst vermieden werden.

## Beispiel mit Type Casting

```
public class SparKontoTest4 {
    public static void main(String[] args) {
        SparKonto sparkonto = new SparKonto(450.0, 1.2);
        BankKonto konto = sparkonto;
        konto.einzahlen(50);
        SparKonto nocheinsparkonto = (SparKonto)konto;
        // mit Type Casting
        nocheinsparkonto.zinsenAnrechnen();
        System.out.println(konto.getKontoStand());
        System.out.println(sparkonto.getKontoStand());
        System.out.println(nocheinsparkonto.getKontoStand());
    }
}
```

Kompilieren und Ausführen:

```
> javac SparKontoTest4.java
> java SparKontoTest4
506.0
506.0
506.0
```

## Beispiel mit Type Casting und ClassCastException

```
public class SparKontoTest5 {
    public static void main(String[] args) {
        SparKonto sparkonto = new SparKonto(450.0, 1.2);
        BankKonto konto = new BankKonto(50);
        SparKonto nocheinsparkonto = (SparKonto)konto;
        // mit Type Casting und Laufzeitfehler
        System.out.println(konto.getKontoStand());
        System.out.println(sparkonto.getKontoStand());
        System.out.println(nocheinsparkonto.getKontoStand());
    }
}
```

Kompilieren und Ausführen:

```
> javac SparKontoTest5.java
```

```
> java SparKontoTest5
```

```
Exception in thread "main" java.lang.ClassCastException:
    BankKonto cannot be cast to SparKonto
    at SparKontoTest5.main(SparKontoTest5.java:5)
```

# Überschreiben von Methoden

---

- In vielen Fällen möchte man die Implementierung von Methoden in der Unterklasse abändern.  
Beispiel: Beim Girokonto soll bei jeder Ein- und Auszahlung eine Gebühr vom Kontostand abgezogen werden.
- Deshalb: Man darf in der Unterklasse Methoden der Oberklasse neu implementieren. Das nennt man **überschreiben**.
- Eine überschriebene Methode muss in der Unterklasse denselben Namen und dieselben Parameter (in Anzahl und Typ) wie die Methode der Oberklasse haben. Die Sichtbarkeit darf nicht eingeschränkt werden. Der Ergebnistyp (falls vorhanden) muss zum bisherigen Ergebnistyp passen.
- Überschreiben in UML-Diagrammen: Die überschriebene Methode wird in der Unterklasse nochmal aufgeführt.

# Überschreiben von Methoden

---

- In vielen Fällen möchte man die Implementierung von Methoden in der Unterklasse abändern.  
Beispiel: Beim Girokonto soll bei jeder Ein- und Auszahlung eine Gebühr vom Kontostand abgezogen werden.
- Deshalb: Man darf in der Unterklasse Methoden der Oberklasse neu implementieren. Das nennt man **überschreiben**.
- Eine überschriebene Methode muss in der Unterklasse denselben Namen und dieselben Parameter (in Anzahl und Typ) wie die Methode der Oberklasse haben. Die Sichtbarkeit darf nicht eingeschränkt werden. Der Ergebnistyp (falls vorhanden) muss zum bisherigen Ergebnistyp passen.
- Überschreiben in UML-Diagrammen: Die überschriebene Methode wird in der Unterklasse nochmal aufgeführt.

# Überschreiben von Methoden

---

- In vielen Fällen möchte man die Implementierung von Methoden in der Unterklasse abändern.  
Beispiel: Beim Girokonto soll bei jeder Ein- und Auszahlung eine Gebühr vom Kontostand abgezogen werden.
- Deshalb: Man darf in der Unterklasse Methoden der Oberklasse neu implementieren. Das nennt man **überschreiben**.
- Eine überschriebene Methode muss in der Unterklasse **denselben Namen** und **dieselben Parameter (in Anzahl und Typ)** wie die Methode der Oberklasse haben. Die Sichtbarkeit darf nicht eingeschränkt werden. Der Ergebnistyp (falls vorhanden) muss zum bisherigen Ergebnistyp passen.
- Überschreiben in UML-Diagrammen: Die überschriebene Methode wird in der Unterklasse nochmal aufgeführt.

# Überschreiben von Methoden

---

- In vielen Fällen möchte man die Implementierung von Methoden in der Unterklasse abändern.  
Beispiel: Beim Girokonto soll bei jeder Ein- und Auszahlung eine Gebühr vom Kontostand abgezogen werden.
- Deshalb: Man darf in der Unterklasse Methoden der Oberklasse neu implementieren. Das nennt man **überschreiben**.
- Eine überschriebene Methode muss in der Unterklasse **denselben Namen** und **dieselben Parameter (in Anzahl und Typ)** wie die Methode der Oberklasse haben. Die Sichtbarkeit darf nicht eingeschränkt werden. Der Ergebnistyp (falls vorhanden) muss zum bisherigen Ergebnistyp passen.
- Überschreiben in UML-Diagrammen: Die überschriebene Methode wird in der Unterklasse nochmal aufgeführt.

# Beispiel: GiroKonto

---

```
public class GiroKonto extends BankKonto {
    private double gebuehr;

    public GiroKonto(double anfangsBetrag, double gebuehr) {
        super(anfangsBetrag);
        this.gebuehr = gebuehr;
    }

    public void abheben(double x) {
        this.kontoStand -= x+this.gebuehr;
    }

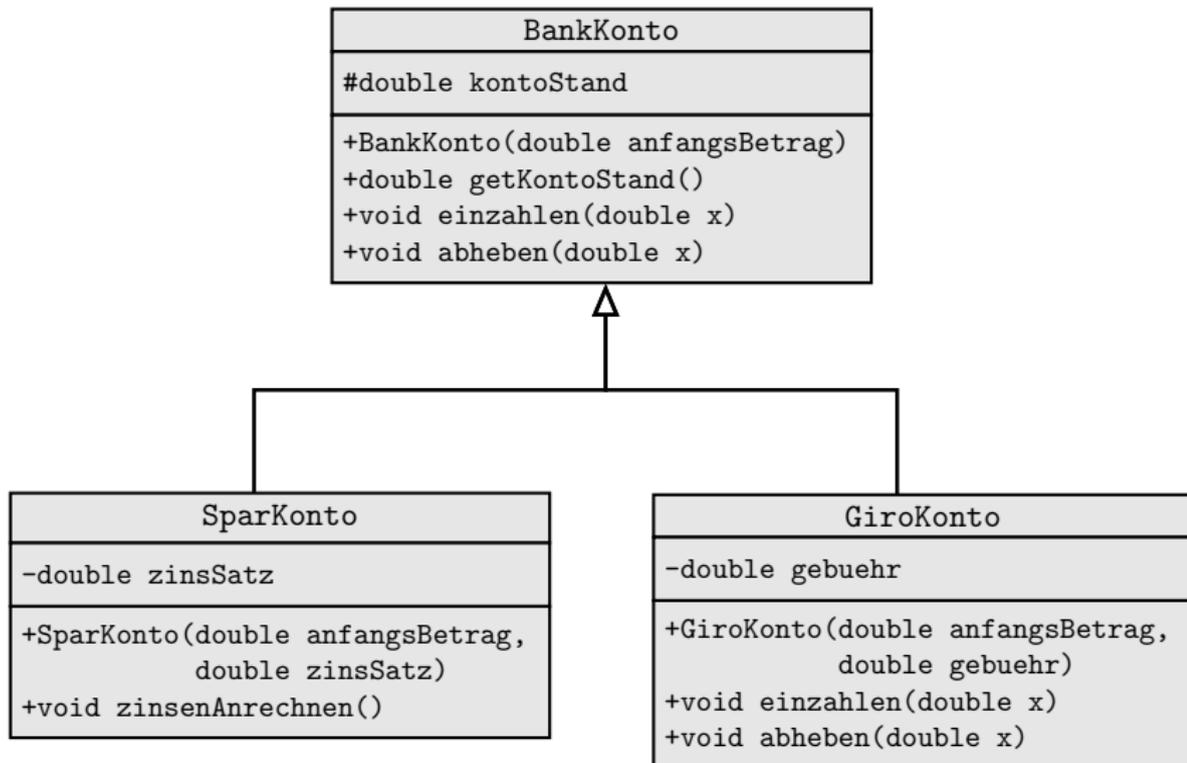
    public void einzahlen(double x) {
        this.kontoStand += x-this.gebuehr;
    }
}
```

## Beispiel: GiroKonto verwenden

---

```
public class GiroKontoTest {
    public static void main(String[] args) {
        BankKonto sparkonto = new SparKonto(100, 1.2);
        BankKonto girokonto = new GiroKonto(100.0, 0.02);
                                // 2 Cent Transaktionsgebuehr
        sparkonto.einzahlen(100.0);
        sparkonto.abheben(100.0);
        girokonto.einzahlen(100.0);
        girokonto.abheben(100.0);
        System.out.println("SPARKONTO:" +
            sparkonto.getKontoStand());
        System.out.println("GIROKONTO:" +
            girokonto.getKontoStand());
    }
}
```

# UML-Diagramm zu den Konten-Klassen



Eine Variable `o` vom Typ `C` kann auf Instanzen verschiedener Klassen verweisen:

Objekte der Klasse `C`, Objekte einer Unterklasse `D` von `C`, Objekte einer Unterklasse `E` von `D`, ...

Beim Aufruf einer Methode `o.m()` muss eine Implementierung der Methode `m` ausgewählt werden. Durch Überschreiben kann es mehrere geben!

Die Auswahl der auszuführenden Methode richtet sich nach der Klasse des Objekts, auf welches `o` zeigt, und nicht nach dem Typ von `o`.

Man spricht in diesem Zusammenhang von **Polymorphie** (Vielgestaltigkeit).

# Beispiel: Dynamische Bindung

```
public class GiroKontoTest2 {
    public static void main(String[] args) {
        BankKonto konto = new GiroKonto(100.0, 0.02); // 2 Cent Gebuehr
        konto.einzahlen(100.0);
        System.out.println(konto.getKontoStand());
        konto = new BankKonto(100.0);
        konto.einzahlen(100.0);
        System.out.println(konto.getKontoStand());
    }
}
```

Ausführung:

```
java GiroKontoTest2
199.98000000000002
200.0
```

Die **dynamische Bindung** bewirkt, dass beim ersten Aufruf von `konto.einzahlen(100.0)` der Rumpf von `einzahlen` der Klasse `GiroKonto` und beim zweiten Aufruf von `konto.einzahlen(100.0)` der Rumpf der Klasse `BankKonto` verwendet wird.

# Dynamische Bindung

---

- Bei einem Methodenaufruf  $e.m(a_1, \dots, a_n)$  muss eine Implementierung der Methode  $m$  ausgewählt und ausgeführt werden.
- Die Auswahl der Implementierung richtet sich nach der Klasse  $C$  des Objekts, auf welches  $e$  im aktuellen Zustand zeigt und **nicht** nach dem Klassentyp des Ausdrucks  $e$ .
- Ist die Methode  $m$  in der Klasse  $C$  definiert, so wird diese Implementierung ausgeführt. Ansonsten wird nach der **kleinsten Oberklasse von**  $C$  gesucht, in der die Methode definiert ist und diese Implementierung wird dann ausgeführt.
- Die Entscheidung, welche Implementierung der Methode benutzt wird, wird **dynamisch (d.h. erst zur Laufzeit)** getroffen. Man spricht von **dynamischer Bindung** (des Methodennamens an die auszuführende Implementierung).

# Dynamische Bindung

---

- Bei einem Methodenaufruf  $e.m(a_1, \dots, a_n)$  muss eine Implementierung der Methode  $m$  ausgewählt und ausgeführt werden.
- Die Auswahl der Implementierung richtet sich nach der Klasse  $C$  des Objekts, auf welches  $e$  im aktuellen Zustand zeigt und **nicht** nach dem Klassentyp des Ausdrucks  $e$ .
- Ist die Methode  $m$  in der Klasse  $C$  definiert, so wird diese Implementierung ausgeführt. Ansonsten wird nach der **kleinsten Oberklasse von**  $C$  gesucht, in der die Methode definiert ist und diese Implementierung wird dann ausgeführt.
- Die Entscheidung, welche Implementierung der Methode benutzt wird, wird **dynamisch (d.h. erst zur Laufzeit)** getroffen. Man spricht von **dynamischer Bindung** (des Methodennamens an die auszuführende Implementierung).

# Dynamische Bindung

---

- Bei einem Methodenaufruf  $e.m(a_1, \dots, a_n)$  muss eine Implementierung der Methode  $m$  ausgewählt und ausgeführt werden.
- Die Auswahl der Implementierung richtet sich nach der Klasse  $C$  des Objekts, auf welches  $e$  im aktuellen Zustand zeigt und **nicht** nach dem Klassentyp des Ausdrucks  $e$ .
- Ist die Methode  $m$  in der Klasse  $C$  definiert, so wird diese Implementierung ausgeführt. Ansonsten wird nach der **kleinsten Oberklasse von**  $C$  gesucht, in der die Methode definiert ist und diese Implementierung wird dann ausgeführt.
- Die Entscheidung, welche Implementierung der Methode benutzt wird, wird **dynamisch** (d.h. erst zur Laufzeit) getroffen. Man spricht von **dynamischer Bindung** (des Methodennamens an die auszuführende Implementierung).

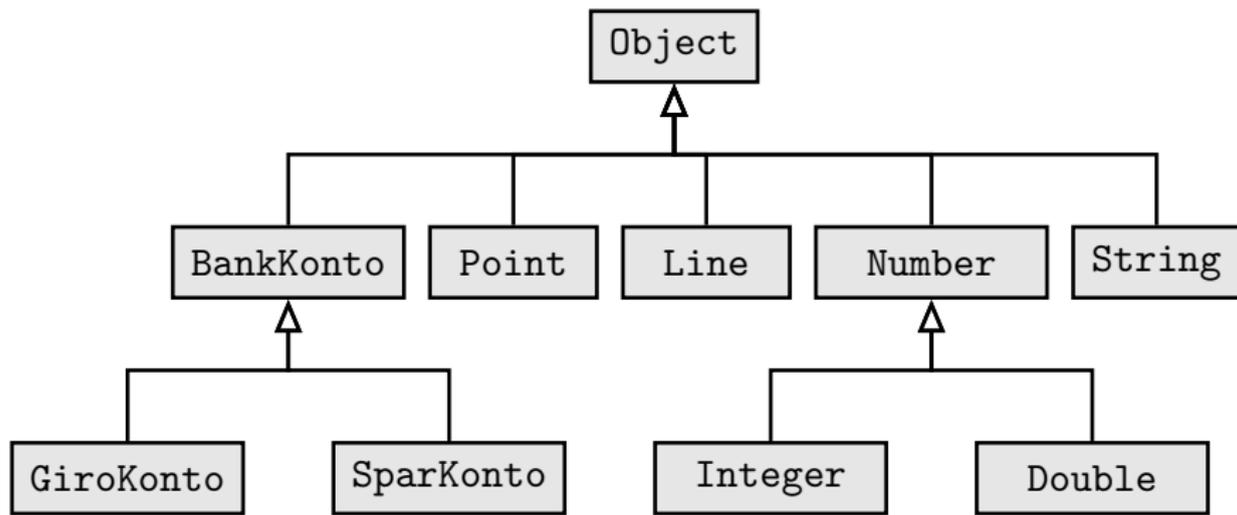
# Dynamische Bindung

---

- Bei einem Methodenaufruf  $e.m(a_1, \dots, a_n)$  muss eine Implementierung der Methode  $m$  ausgewählt und ausgeführt werden.
- Die Auswahl der Implementierung richtet sich nach der Klasse  $C$  des Objekts, auf welches  $e$  im aktuellen Zustand zeigt und **nicht** nach dem Klassentyp des Ausdrucks  $e$ .
- Ist die Methode  $m$  in der Klasse  $C$  definiert, so wird diese Implementierung ausgeführt. Ansonsten wird nach der **kleinsten Oberklasse von**  $C$  gesucht, in der die Methode definiert ist und diese Implementierung wird dann ausgeführt.
- Die Entscheidung, welche Implementierung der Methode benutzt wird, wird **dynamisch (d.h. erst zur Laufzeit)** getroffen. Man spricht von **dynamischer Bindung** (des Methodennamens an die auszuführende Implementierung).

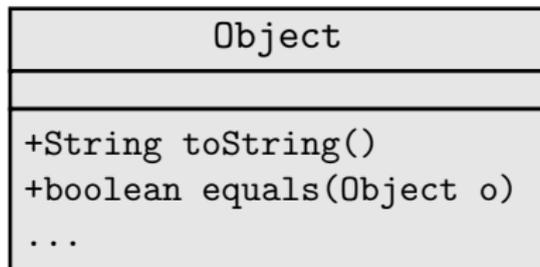
# Javas Klassenhierarchie – die Klasse Object

- In Java gibt es eine **oberste Klasse** namens Object
- Jede Klasse ist implizit auch eine (nicht notwendigerweise direkte) Unterklasse von Object



# Die Klasse Object

---



- Methode `toString` liefert eine `String`-Repräsentation des Objekts.
- Methode `equals` zum Vergleich von Objekten.
- Die Unterklassen überschreiben beide Methoden meistens.
- `equals` wird meist so überschrieben, dass es auf Gleichheit von Objektzuständen und nicht auf Gleichheit von Objektreferenzen testet

# Beispiel: Überschreiben von equals für Point

```
public class Point {
    ...
    @Override
    public boolean equals(Object o) {
        if ( o == null ) return false;
        if ( o == this ) return true;
        // teste ob o zur aktuellen Klasse geh"ort:
        if ( ! o.getClass().equals(getClass()) )
            return false; // nein, dann false
        else {
            Point that = (Point) o; // ansonsten: sicheres type casting
            return this.getX() == that.getX()
                && this.getY() == that.getY();}
        }
    }
}
```

Bemerkung zu @Override:

- Die Annotation @Override kennzeichnet, dass eine Methode überschrieben wird. Der Compiler prüft, dass die Oberklasse diese Methode enthält und gibt anderenfalls einen Fehler aus.
- Nicht notwendig, aber hilft bei der Fehlervermeidung!

## Beispiel: Überschreiben von equals für Point (2)

---

Beachte: Die Dokumentation verlangt:

“...to maintain the general contract for the hashCode method, which states that equal objects must have equal hash codes.”

D.h. wenn equals überschrieben wird muss man i.a. auch hashCode überschreiben.

```
import java.util.Objects;

public class Point {
    ...
    @Override
    public int hashCode() {
        // Verwendung einer Default-Methode zum Hashen mehrerer
        // Werte
        int hash = Objects.hash(this.getX(), this.getY());
        return hash;
    }
}
```

In Java ist es möglich, Methoden zu **überladen**:

- Der selbe Methodennamen wird für unterschiedliche Argumentanzahlen und -typen verwendet.
- Das geht auch bei Konstruktoren.
- Java entscheidet sich für die am besten passende Methode

# Beispiel zur Überladung

```
public class SparKonto extends BankKonto {
    private double zinsSatz;

    // Ueberladener Konstruktor
    public SparKonto(double anfangsBetrag, double zinsSatz) {
        super(anfangsBetrag);
        this.zinsSatz = zinsSatz;
    }
    public SparKonto(double anfangsBetrag) {
        super(anfangsBetrag);
        this.zinsSatz = 0.0;
    }

    // Ueberladene Methode
    public void zinsenAnrechnen() {
        double zinsen = this.kontoStand * this.zinsSatz / 100.0;
        this.kontoStand += zinsen;
    }
    public void zinsenAnrechnen(double neuerZinsSatz) {
        this.zinsSatz = neuerZinsSatz;
        zinsenAnrechnen();
    }
}
```

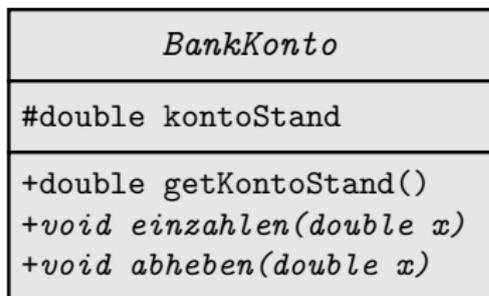
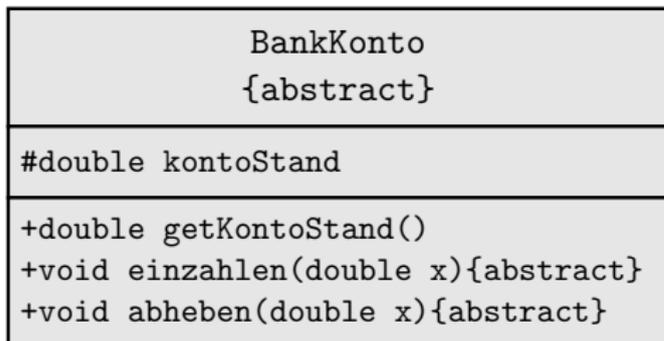
# Abstrakte Klassen und abstrakte Methoden

- Beschreibung von abstrakten Begriffen, deren Realisierungen nur in speziellen Ausprägungen sinnvoll sind.  
Z.B. Bankkonten treten nur in einer spezielleren Form auf (Girokonto, Sparkonto), d.h. „Bankkonto“ ist eigentlich ein abstrakter Begriff
- **Abstrakte Klassen** sind Klassen, für die keine Objekte direkt erzeugt werden können, die also nicht direkt „instanzierbar“ sind. Instanzen können dann für konkrete Subklassen einer abstrakten Klasse erzeugt werden.
- **Abstrakte Methoden** sind Methoden, die in einer (abstrakten) Klasse keine Implementierung haben.
- Jede Klasse, die eine abstrakte Methode hat, muss abstrakt sein (jedoch nicht umgekehrt: Abstrakte Klassen können auch nicht-abstrakte Methoden haben).
- Implementierungen von abstrakten Methoden werden in Unterklassen angegeben.

# Beispiel

```
public abstract class BankKonto {  
    protected double kontoStand;  
    public double getKontoStand() {  
        return this.kontoStand;  
    }  
    public abstract void einzahlen (double x);  
    public abstract void abheben(double x);  
}
```

Darstellung in UML: Mit {abstract} und/oder in kursiver Schrift:



# Beispiel: Implementierung in der Unterklasse SparKonto

```
public class SparKonto extends BankKonto {
    private double zinsSatz;
    public SparKonto(double anfangsBetrag, double zinsSatz) {
        this.kontoStand = anfangsBetrag; // neu, da kein Konstruktor
        fuer die Oberklasse
        this.zinsSatz = zinsSatz;
    }
    public void zinsenAnrechnen() {
        double zinsen = this.kontoStand * this.zinsSatz / 100.0;
        this.kontoStand += zinsen;
    }
    public void einzahlen(double betrag) {
        this.kontoStand += betrag;
    }
    public void abheben(double betrag) {
        this.kontoStand -= betrag;
    }
}
```

# Beispiel: Implementierung in der Unterklasse GiroKonto

---

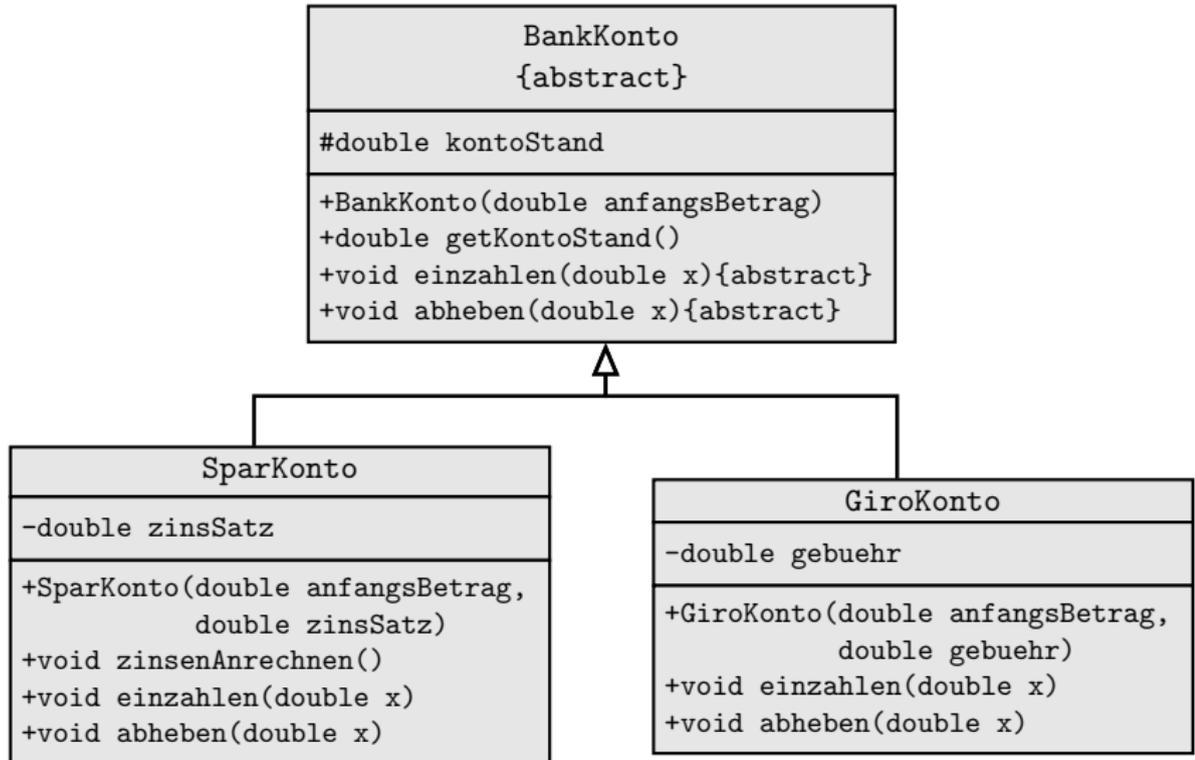
```
public class GiroKonto extends BankKonto {
    private double gebuehr;

    public GiroKonto(double anfangsBetrag, double gebuehr) {
        this.kontoStand = anfangsBetrag;
        this.gebuehr = gebuehr;
    }

    public void abheben(double betrag) {
        this.kontoStand -= betrag+this.gebuehr;
    }

    public void einzahlen(double betrag) {
        this.kontoStand += betrag-this.gebuehr;
    }
}
```

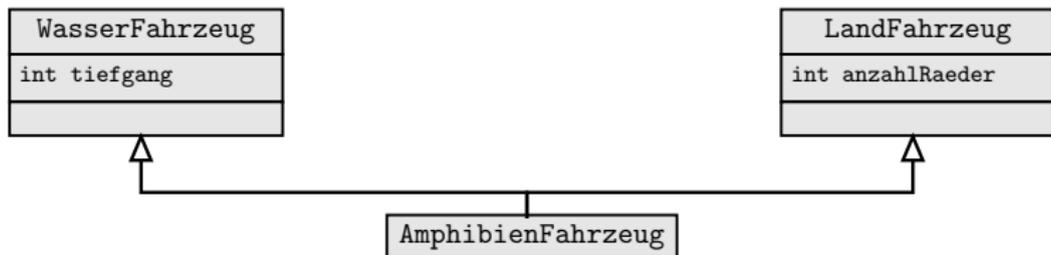
# Abstrakte und konkrete Klassen



# Mehrfachvererbung

Man spricht beim objektorientierten Programmieren von **Mehrfachvererbung**, wenn eine Klasse **mehr als eine Oberklasse** besitzt.

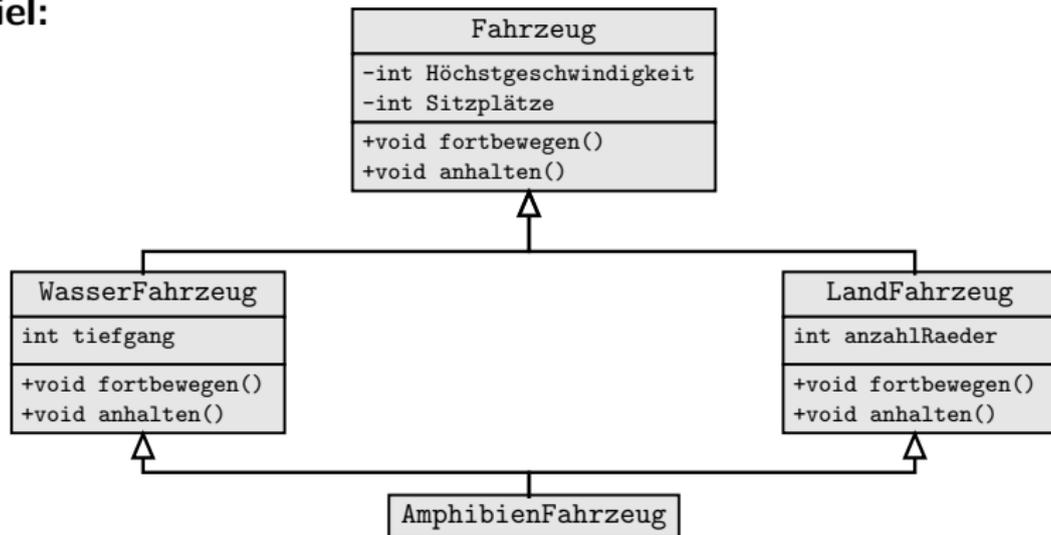
**Beispiel:**



# Mehrfachvererbung

Man spricht beim objektorientierten Programmieren von **Mehrfachvererbung**, wenn eine Klasse **mehr als eine Oberklasse** besitzt.

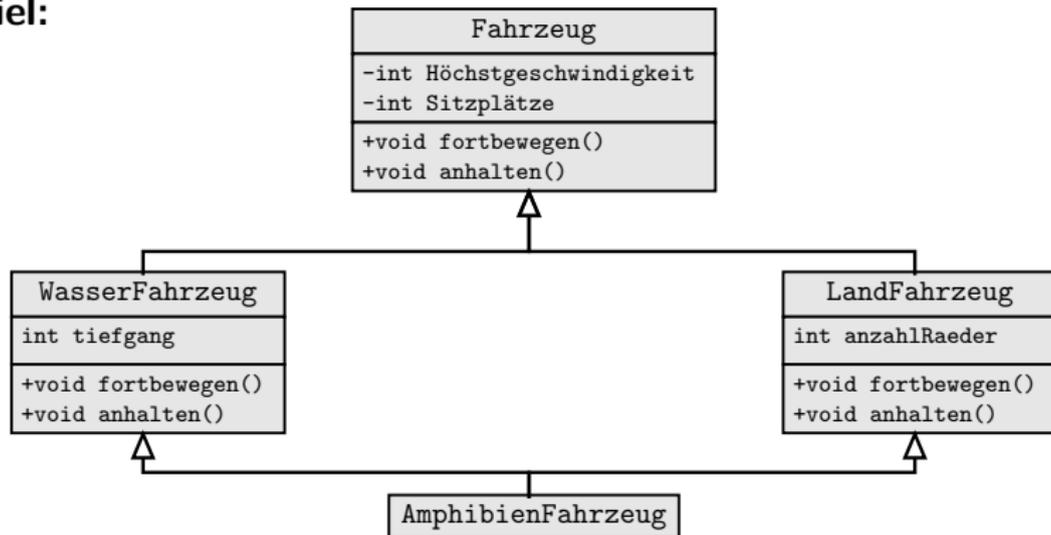
**Beispiel:**



# Mehrfachvererbung

Man spricht beim objektorientierten Programmieren von **Mehrfachvererbung**, wenn eine Klasse **mehr als eine Oberklasse** besitzt.

**Beispiel:**

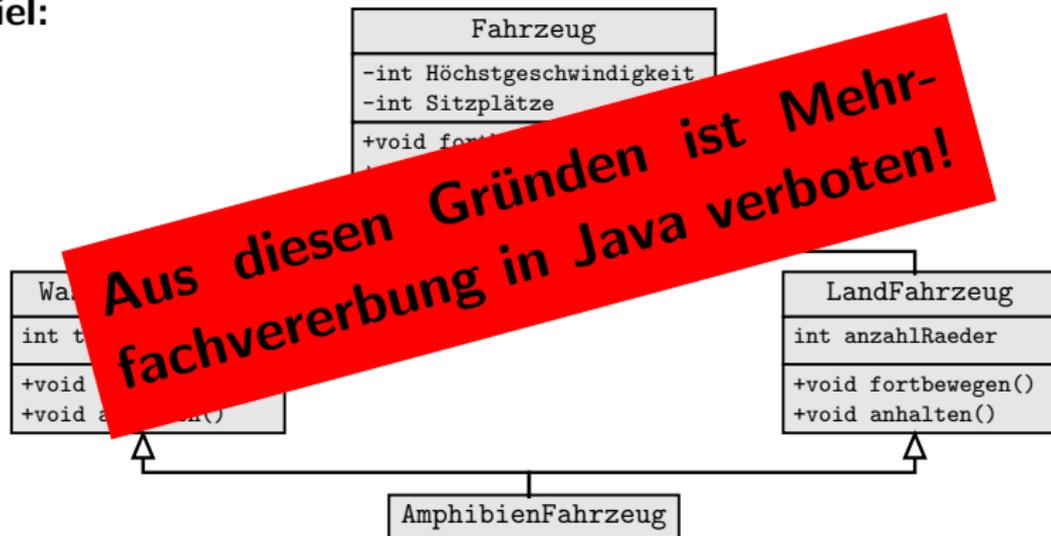


**Probleme:** Welche Attribute besitzt AmphibienFahrzeug? Welche Implementierung von fortbewegen() / anhalten() wird für AmphibienFahrzeuge verwendet?

# Mehrfachvererbung

Man spricht beim objektorientierten Programmieren von **Mehrfachvererbung**, wenn eine Klasse **mehr als eine Oberklasse** besitzt.

**Beispiel:**



**Probleme:** Welche Attribute besitzt AmphibienFahrzeug? Welche Implementierung von fortbewegen() / anhalten() wird für AmphibienFahrzeuge verwendet?

# Interfaces (Schnittstellen)

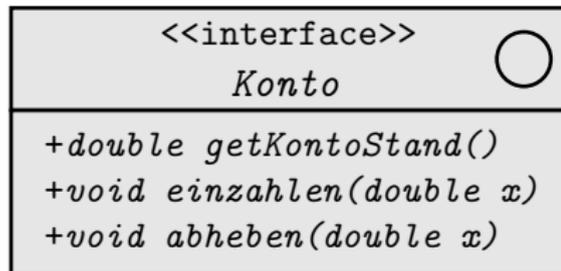
---

- Interfaces (Schnittstellen) sind ein Hilfsmittel zur Aufteilung von Systemen in einzelne Komponenten (bzw. zur Entkopplung der starren Klassenhierarchie)
- Interfaces bieten Dienste (in Form von Methoden) an, die von bestimmten Klassen implementiert und von anderen Klassen benutzt werden können.
- Interfaces haben **nur öffentliche, abstrakte Methoden** und **keine** Objektattribute
- Mit jeder Interface-Deklaration wird ein **Interfacetyp** definiert, der wie ein Klassentyp verwendet werden kann.
- Klassen können **mehrere** Interfaces implementieren  
Daher erlauben Interfaces **eine eingeschränkte Form der Mehrfachvererbung**.

# Beispiel und Darstellung in UML

```
public interface Konto {  
    public double getKontoStand();  
    public void einzahlen(double x);  
    public void abheben(double x);  
}
```

Darstellung in UML: alles kursiv, Schlüsselwort <<interface>>, Kreis als Icon, nur abstrakte Methoden



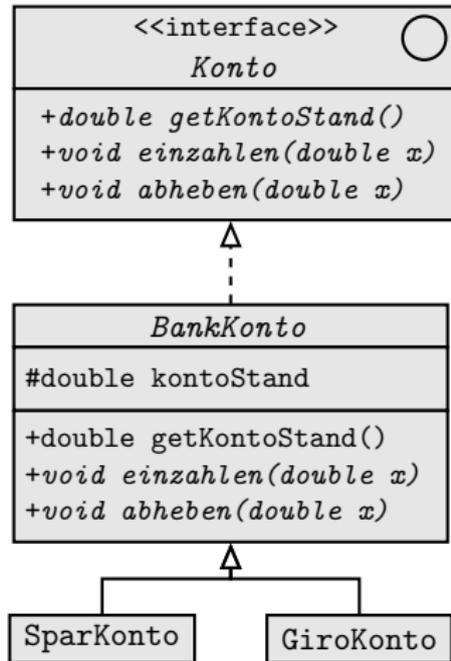
# Beispiel: Implementierung von Interfaces

```
public interface Konto{
    ...
}

public abstract BankKonto
    implements Konto {
    ...
}

public GiroKonto extends BankKonto{
    ...
}

public SparKonto extends BankKonto{
    ...
}
```



# Erweiterte Grammatik für Klassendeklarationen in Java

ClassDecl =  
["public"] ["abstract"] "class" [Super][Interfaces] Identifier ClassBody

Super = "extends" ClassType

Interfaces = "implements" InterfaceType { ", " InterfaceType }

InterfaceType = Identifier

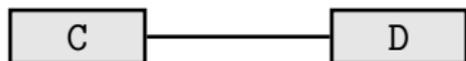
ClassBody = ...

Ab ClassBody wie vorher (vgl. Kap. 5) aber mit abstrakten Methoden.

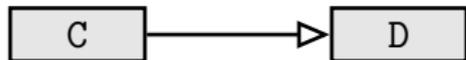
## Beachte:

- In Java kann eine Klasse **höchstens eine** Oberklasse erweitern, eine Klasse kann aber **mehrere** Interfaces implementieren.
- `D implements C` induziert (genauso wie `D extends C`) eine Subtypbeziehung: D ist Subtyp von C.

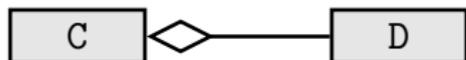
# Einige Beziehungen zwischen Klassen in UML



Assoziation: Die Klassen C und D stehen in Beziehung



Vererbung (Generalisierung): C ist Unterklasse von D

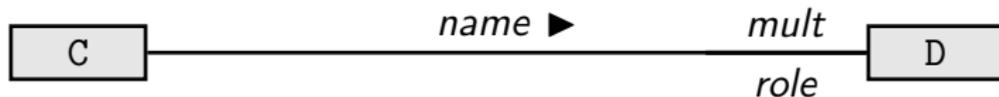


Aggregation: Jedes Objekt von C enthält Objekte von D



Komposition (starke Aggregation): Objekte von D existieren **ausschließlich** innerhalb von C

# Assoziation mit Namen, Rollen, Multiplizitäten

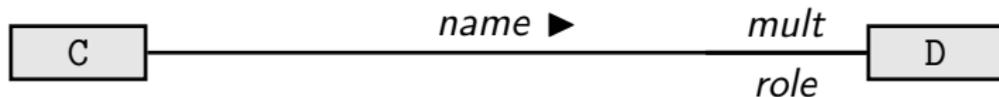


- Das Diagramm illustriert, dass jedes Objekt  $o$  der Klasse  $C$  mit „*mult*“ vielen Objekten der Klasse  $D$  in Beziehung stehen, welche für  $o$  die Rolle *role* spielen.
- *name* gibt den Namen der Assoziation an, ► zeigt die Leserichtung.
- Rolle *role* ist ein Name.
- Multiplizität *mult* ist
  - eine natürliche Zahl  $n$ ,
  - ein Stern  $*$  (repräsentiert beliebig viele Objekte) oder
  - ein Intervall der Form  $n..m$  oder  $n..*$

## Beispiel:



# Assoziation mit Namen, Rollen, Multiplizitäten



- Das Diagramm illustriert, dass jedes Objekt  $o$  der Klasse  $C$  mit „*mult*“ vielen Objekten der Klasse  $D$  in Beziehung stehen, welche für  $o$  die Rolle *role* spielen.
- *name* gibt den Namen der Assoziation an,  $\blacktriangleright$  zeigt die Leserichtung.
- Rolle *role* ist ein Name.
- Multiplizität *mult* ist
  - eine natürliche Zahl  $n$ ,
  - ein Stern  $*$  (repräsentiert beliebig viele Objekte) oder
  - ein Intervall der Form  $n..m$  oder  $n..*$

## Beispiel:



Rollen und Multiplizitäten können an beiden Enden existieren

# Beispiel Aggregation und Komposition

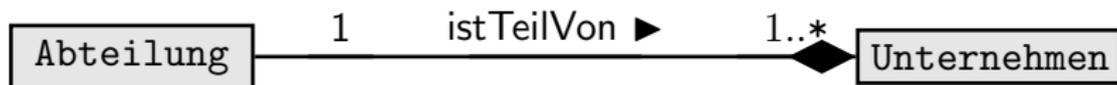
## Beispiel Aggregation

- Eine Linie besteht aus genau zwei Punkten.
- Punkte können jedoch auch unabhängig von Linien existieren



## Beispiel Komposition

- Ein Unternehmen besteht aus mindestens einer Abteilung
- Eine Abteilung existiert nicht ohne ein Unternehmen.
- Eine Abteilung ist Teil genau eines Unternehmens.



# Umsetzung der Assoziation in Java



```
public class C {
    private D role;
}
```



```
public class C {
    private D[] role; // oder andere Container-Klassen
                    // (siehe Kapitel 9)
}
```

# Beispiel: Objektorientierter Entwurf

Prozess der Modellierung und Problemlösung:



Einige Prinzipien beim Entwurf

- Substantive der Problembeschreibung werden oft durch Klassen abgebildet.
- Verben der Problembeschreibung werden oft zu Methoden im Modell.
- Klassen sollten nicht zu sehr gekoppelt sein, d.h. sich möglichst nicht gegenseitig benutzen.
- Klassen nicht zu groß machen, sondern in Teile zerlegen

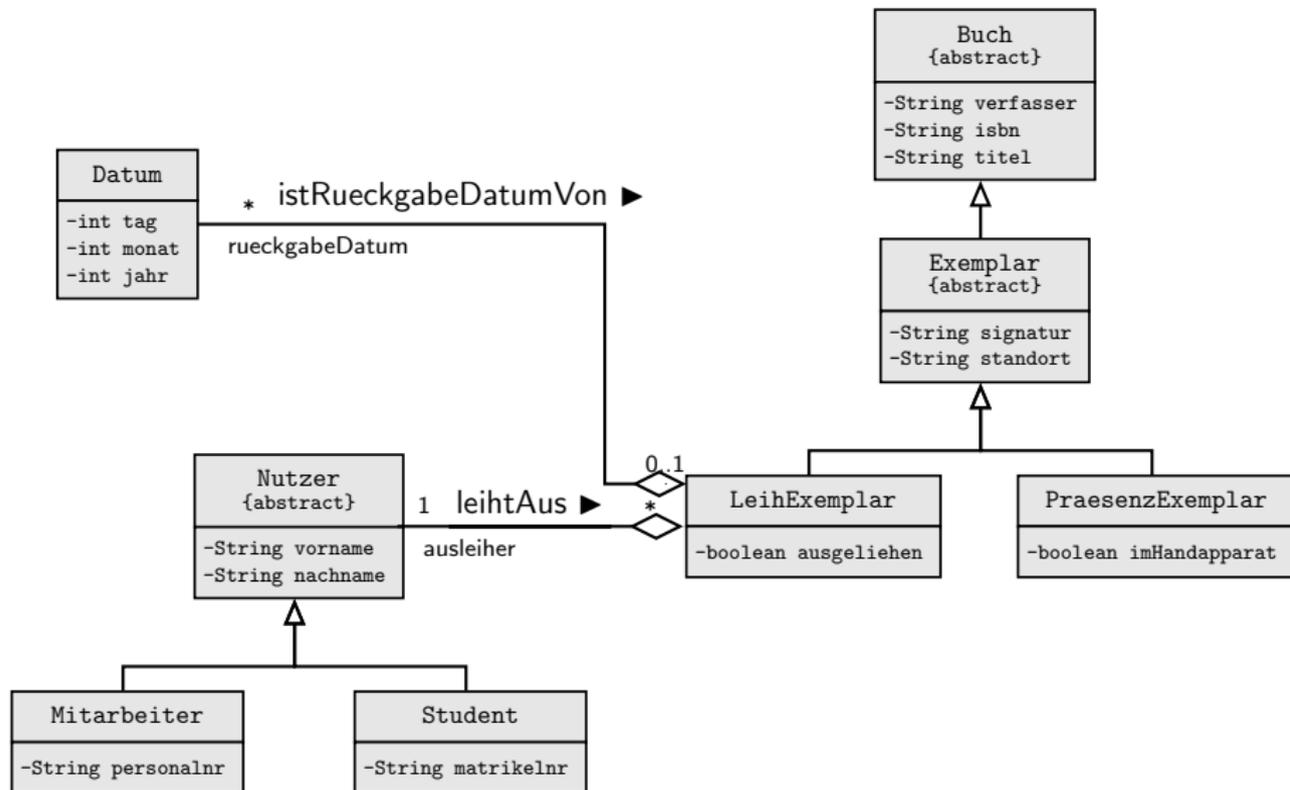
# Beispiel: Bibliothekssystem

---

## Problembeschreibung:

- Ein Bibliothekssystem dient zur Verwaltung der Bücher, der Nutzer und der Entleihvorgänge.
- Ein Buch hat eine ISBN, einen oder mehrere Verfasser und einen Titel.
- Von ein und demselben Buch kann es mehrere Exemplare geben. Jedes Exemplar hat eine eindeutige Signatur und einen Standort.
- Es gibt Präsenzexemplare, die nicht ausleihbar sind, und ausleihbare Exemplare. Ein Präsenzexemplar kann im Handapparat eines Lehrstuhls stehen. Ein ausleihbares Buch kann ausgeliehen sein oder nicht.
- Nutzer der Bibliothek sind Studierende und Mitarbeiter.
- Nutzer haben eine Vor- und einen Nachnamen.
- Studierenden ist eine Matrikelnummer zugeordnet, Mitarbeitern ist eine Personalnummer zugeordnet.
- Ein Kunde kann mehrere Bücher ausleihen. Für jedes entliehene Buch, gibt es ein Rückgabedatum.

# Mögliche Modellierung



# Mögliche Implementierung (Auszüge)

```
public abstract class Buch {
    private String verfasser, isbn, titel;
    public Buch(String verfasser, String isbn, String titel) {
        this.verfasser = verfasser;
        this.isbn = isbn;
        this.titel = titel;
    }
}

public abstract class Exemplar extends Buch {
    private String signatur, standort;
    public Exemplar(String verfasser, String isbn, String titel, String signatur,
        String standort){
        super(verfasser, isbn, titel);
        this.signatur = signatur;
        this.standort = standort;
    }
}

public class PraesenzExemplar extends Exemplar {
    private boolean imHandapparat;
    public PraesenzExemplar(String verfasser, String isbn, String titel, String
        signatur, String standort, boolean imHandapparat) {
        super(verfasser, isbn, titel, signatur, standort);
        this.imHandapparat = imHandapparat;
    }
}
```

## Mögliche Implementierung (Auszüge) (2)

```
public class LeihExemplar extends Exemplar {
    private boolean ausgeliehen;
    private Nutzer ausleiher;
    private Datum rueckgabeDatum;
    public LeihExemplar(String verfasser, String isbn, String titel,String
        signatur, String standort) {
        super(verfasser, isbn,titel,signatur,standort);
        this.ausgeliehen = false;
        this.ausleiher = null;
        this.rueckgabeDatum = null;
    }
    public void leiheAus(Nutzer ausleiher,Datum rueckgabeDatum) {
        if (this.ausgeliehen) {
            System.out.println("Buch ist schon ausgeliehen!");
            System.out.println("Das Buch wird am " + this.rueckgabeDatum.toString() + "
                zurueckgegeben");
            // besser waere ein Fehler (siehe spaeter...)
        }
        else {
            this.ausleiher = ausleiher;
            this.rueckgabeDatum = rueckgabeDatum;
            ausgeliehen = true;}
    }
    public void rueckgabe() {
        if (this.ausgeliehen) {
            this.ausgeliehen = false;
            this.ausleiher = null;
            this.rueckgabeDatum = null;
        }
    }
}
```

# Mögliche Implementierung (Auszüge) (3)

---

```
public abstract class Nutzer {
    private String vorname, nachname;
    public Nutzer(String vorname, String nachname) {
        this.vorname = vorname;
        this.nachname = nachname;
    }
}

public class Student extends Nutzer {
    private final String matrikelnr; // aendern der Matrikelnummer nicht erlaubt
    public Student(String vorname, String nachname, String matrikelnr) {
        super(vorname, nachname);
        this.matrikelnr = matrikelnr;
    }
}

public class Mitarbeiter extends Nutzer {
    private String personalnr;
    public Mitarbeiter(String vorname, String nachname, String personalnr) {
        super(vorname, nachname);
        this.personalnr = personalnr;
    }
}
```

# Mögliche Implementierung (Auszüge) (4)

```
public class Datum {
    private int tag,monat,jahr;
    public Datum(int tag, int monat, int jahr) {
        if (tag < 1 || tag > 31) {
            System.out.println("Tag nicht im Intervall [1..31]");
            System.out.println("Setze Tag auf 1");
            this.tag=1;
        }
        else {this.tag=tag;}
        if (monat < 1 || monat > 12) {
            System.out.println("Monat nicht im Intervall [1..12]");
            System.out.println("Setze Monat auf 1");
            this.monat=1;
        }
        else {this.monat=monat;}
        this.jahr = jahr;
    }
    @Override
    public String toString() {
        return (this.tag + "." + this.monat + "." + this.jahr);
    }
}
```

# Mögliche Implementierung (Auszüge) (5)

---

```
public class Main {
    public static void main(String[] args) {
        LeihExemplar b1 = new LeihExemplar("Peter Gumm, Manfred
            Sommer", "978-3486706413", "Einfuehrung in die Informatik", "0001/8
            10-7752", "Zentralbibliothek");
        Nutzer n1 = new Mitarbeiter("David", "Sabel", "12345678");
        Nutzer n2 = new Student("Niko", "Laus", "08154711");
        Datum d1 = new Datum(28,11,2018);
        b1.leiheAus(n1,d1);
        b1.leiheAus(n2,d1);
        b1.rueckgabe();
        b1.leiheAus(n2,d1);
    }
}
```

- Vererbung als Strukturierungsprinzip, Vererbung stellt eine „is-a-Beziehung dar.
- Ober- und Unterklassen, `super`
- Überschreiben von Methoden
- Überladung
- Abstrakte Klassen und Methoden
- Mehrfachvererbung (gibt es nicht in Java)
- Interfaces
- Syntax in Java
- UML: Assoziation, Aggregation, Komposition, Generalisierung