



PM2 Java: Klassen – Typen – Interfaces – Abstrakte Klassen



Fahrplan

- Klassendefinitionen in Java
 - Instanz-Variablen und -Methoden
 - Statische Variablen und Methoden (Klassen-Variablen und -Methoden)
 - Konstruktoren
- Typen - statische Typisierung und dynamisches Binden
- Interfaces ein Konzept zur Flexibilisierung statischer Typisierung
- Interfaces als Abstraktion von Implementierungen
- Abstrakte Klassen eine Kombination von Interface- und Implementierungs-vererbung
- Überladen Überschreiben Überschatten

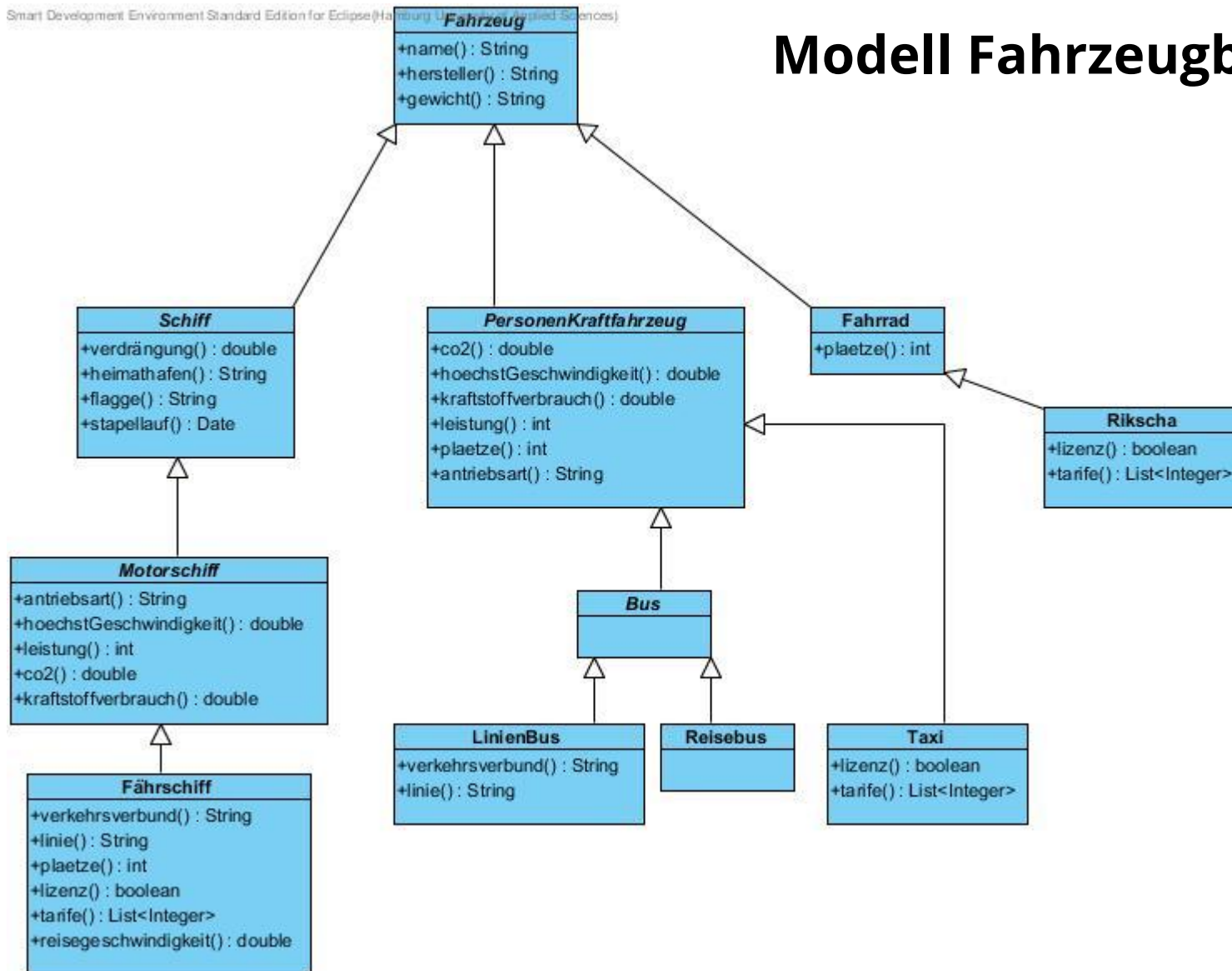


Fahrzeugbeispiel

- **Ausgangspunkt:** Ein Klassenmodell von Fahrzeugen mit den folgenden Beziehungen
 - Schiffe, Personen-Kraftfahrzeuge und Fahrräder sind Fahrzeuge.
 - Motorschiffe sind Schiffe.
 - Fährschiffe sind Motorschiffe.
 - Rikschas sind Fahrräder.
 - Privatfahrzeuge, Taxis oder Busse sind Personen-Kraftfahrzeuge.
 - Linienbusse oder Reisebusse sind Busse.
- Fahrzeuge haben einen Namen, einen Hersteller und ein Gewicht
- Schiffe haben eine Verdrängung, einen Heimathafen, eine Flagge und ein Datum für den Stapellauf
- Motorschiffe und Personen-Kraftfahrzeuge sind motorisiert mit den Eigenschaften Antriebsart, Leistung, Co2, Höchstgeschwindigkeit und Kraftstoffverbrauch.
- Rikschas, Busse und Fährschiffe dienen der Personenbeförderung und haben eine Lizenz und Tarife.
- Fährschiffe und Linienbusse sind öffentliche Verkehrsmittel, die einem Verkehrsverbund angehören mit einer Linienbezeichnung.



Modell Fahrzeugbeispiel





Problem mit dem Fahrzeugbeispiel

- Aufgrund der sich überschneidenden Eigenschaften eine Reihe von Dubletten.
- Vor der Beseitigung der Dubletten, zunächst ein Blick auf Ausschnitte des Modells im Java Quelltext um zu sehen,
 1. wie in Java eine Klassenhierarchie definiert wird.
 2. wie Instanz-Variablen und Methoden definiert werden.
 3. wie Konstruktoren einzuführen sind, um Objekte zu initialisieren
- Im Abschnitt Interfaces wird das Modell durch Refaktorisierung in ein Modell ohne Dubletten überführt.



Die Parent-Klasse aller Fahrzeuge

```
public abstract class Fahrzeug {  
  
    private String name = "";  
    private String hersteller = "";  
    private double gewicht;  
  
    public String name() {  
        return this.name;  
    }  
}
```

- Ist eine abstrakte Klasse markiert durch *abstract* vor dem Klassennamen.
- Abstrakte Klassen können gemeinsames Verhalten und gemeinsame Eigenschaften definieren.
- Es lässt sich von ihnen keine Instanz erzeugen.
- Die Variablen *name ... gewicht* sind Instanz-Variablen der Objekte der Klasse und sollten immer private sein.
- Die Methode *name()* ist eine Instanz-Methode. Da sie einen *String* als Ergebnistyp deklariert, muss mit *return* eine String zurückgegeben werden.



Klassenhierarchien

- Das Ableiten von einer Superklasse wird in Java durch das Schlüsselwort **extends** markiert.
- Die Subklasse „erweitert“ (**extends**) die Superklasse.
- Im Beispiel leitet die **abstrakte** Klasse **Schiff** von der **abstrakten** Klasse **Fahrzeug** ab.

```
public abstract class Schiff extends Fahrzeug
{

    private String heimathafen;
    private String flagge;
    private Date stapellauf;

    public double verdrängung() {
        return gewicht();
    }

    ...

    public String heimathafen() {
        return heimathafen;
    }

}
```



Klassenhierarchien

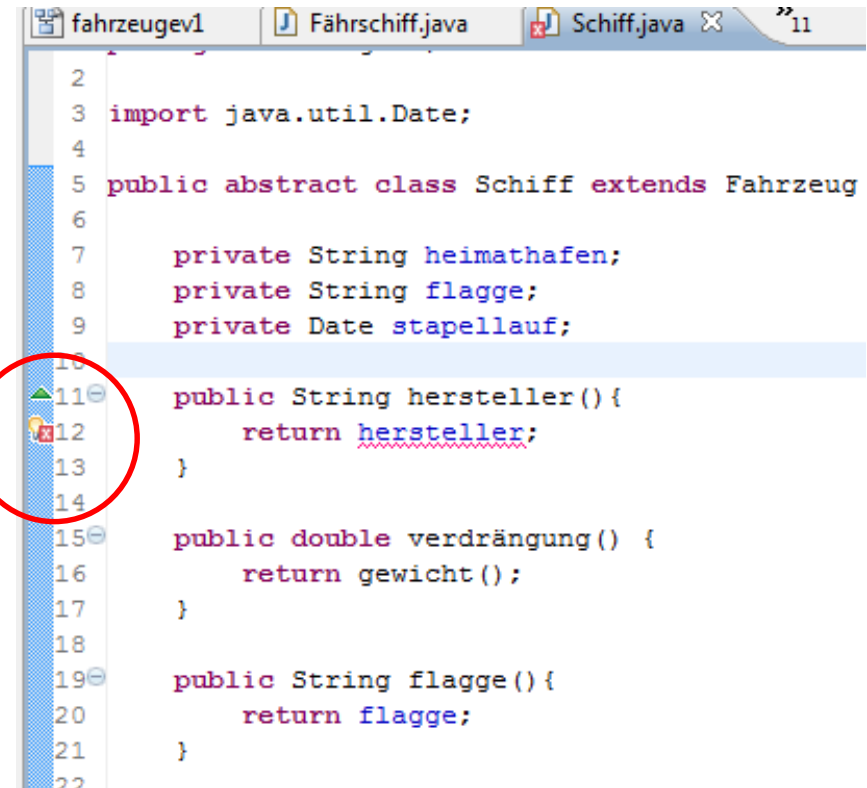
- Natürlich können auch konkrete Klassen von abstrakten Klassen ableiten.
- Im Beispiel leitet die konkrete Klasse *Fäherschiff* von der **abstrakten** Klasse *Motorschiff* ab.

```
public class Fäherschiff extends Motorschiff {  
  
    private String verkehrsverbund;  
    private String linie;  
    private int plaetze;  
    private boolean lizenz;  
    private List<Integer> tarife;  
  
    public String verkehrsverbund() {  
        return verkehrsverbund;  
    }  
  
    public List<Integer> tarife() {  
        return tarife;  
    }  
}
```




Private Instanz-Variablen werden nicht vererbt

- Für Subklassen sind private Instanzvariablen und private Methoden der Superklasse nicht sichtbar.
- Versuchen wir beispielsweise die Attribute *hersteller* der Superklasse *Fahrzeug* in der Subklasse *Schiff* zu referenzieren, erhalten wir einen Compilerfehler.
- Um private Instanz-Variablen für Subklassen sichtbar zu machen, benötigen wir **nicht private Methoden**, die den Inhalt der Instanz-Variablen lesen.



```
1  fahrzeugev1  Fährschiff.java  Schiff.java  11
2
3  import java.util.Date;
4
5  public abstract class Schiff extends Fahrzeug
6
7      private String heimathafen;
8      private String flagge;
9      private Date stapellauf;
10
11     public String hersteller(){
12         return hersteller;
13     }
14
15     public double verdrängung() {
16         return gewicht();
17     }
18
19     public String flagge(){
20         return flagge;
21     }
22
```



Private Instanz-Variablen werden nicht vererbt

- Um private Instanz-Variablen für Subklassen sichtbar zu machen, benötigen wir **nicht private Methoden**, die den Inhalt der Instanz-Variablen lesen.

```
package fahrzeugev1;

import java.util.Date;

public abstract class Fahrzeug {
    private String name = "";
    private String hersteller = "";
    ... //
    public String name() {
        return this.name;
    }
    public String hersteller() {
    return hersteller;
    }
}
```



Public Klassen und Compilation Units

- Quelltextdateien in Java, die auf .java enden heißen Compilation-Units.
- Jede **Compilation-Unit** muss eine Klasse enthalten die den selben Namen der Datei trägt.
- Eine Compilation-Unit **darf nur eine *public*** Klasse enthalten. Diese muss im Namen mit dem Namen der Datei übereinstimmen.

```

1 package fahrzeugev1;
2
3 import java.util.Date;
4
5 public class SchiffV1 {
6
7     private Date stapellaufzeit;
8
9
10    public double verdrängung() {
11        // TODO
12        return 0.0;
13    }
14
15    public Date stapellauf() {
16        return stapellauf;
17    }
18
19    public String getTyp() {
20        return typ;
21    }
22 }

```



METHODEN



Methoden

- Die allgemeine Syntax für Methoden:
 - [<Sichtbarkeits Mod>] [*static* | *abstract*] [*final*] (<Typ> | *void*) <name> (<Typ> param₁, ..., <Typ> param_n)
 - Die in eckigen Klammern ([]) eingeschlossenen Elemente sind optionale Bestandteile
 - <Typ> ist ein Stellvertreter für einen gültigen Java-Typ.
 - Wenn n=0, dann hat die Methode keine Parameter



Variable Argumentlisten (Varargs)

- Die Methode ***format*** der Klasse ***String***, kann mit 1 oder mehreren Argumente aufgerufen werden.
- Das erste Argument ist der Formatstring, die 2-n Argumente die Werte, die im Formatstring in die variablen Stellen ***i\$*** eingesetzt werden.
- Variable Argumentlisten werden in der Methodendefinition notiert, indem dem Typ des Methodenparameter drei Punkte nachgestellt. (***Object... args*** in ***format*** von ***String***)

```
String.format("%1$s,%2$d", "Menge", 5);  
String.format("%1$s,%2$s,%3$s,%4$d ",  
              "Produkt", "Schraube", "Menge", 5);
```

```
// Methode format in String  
public static String format(String  
    format, Object ... args) {  
    return new  
        Formatter().format(format,  
        args).toString();  
}
```



Variable Argumentlisten

- Variable Argumentlisten werden intern in ein Array umgewandelt.
- Daher kann einer Methode mit Varargs auch ein Array übergeben werden: Die zweite Zeile rechts ist identisch mit der ersten.
- Schreibt man eigene Methoden mit einem variablen Parameter, so kann der Parameter wie ein Array behandelt werden.

1.)

```
String.format("%1$s,%2$d", "Menge", 5);  
String.format("%1$s,%2$d", new  
    Object[]{"Menge", 5});
```

2.)

```
static void readVarArgs(Object... args)  
{  
    for (Object object : args) {  
        p(object);  
    }  
}
```

```
readVarArgs("Produkt", "Schraube",  
    "Menge", 5);
```



Produkt
Schraube
Menge
5



Lokale Variablen in Methoden verdecken Instanz-Variablen

- Lokale Variablen in Methoden und Konstruktoren verdecken Instanz-Variablen mit gleichen Namens.
- Im Beispiel rechts verdeckt die lokale Variable *count* in der Methode *setCount* die Instanz-Variable *count*.

➔ **Effekt:** Das *Counter* Objekt inkrementiert nicht korrekt.

```
public class Counter {
    private int count;
    public Counter(){ ... }
    public Counter inc(){
        setCount(count+1);
        return this;
    }
    protected void setCount(int count1){
        int count = count1;
    }
    public int getStep() {...}
    public int getCount(){...}
    public void reset(){...}
}
```

```
Counter c1 = new Counter();
c1.inc().inc();
p("c1 " + c1.getCount());
```

➔ c1 0



Instanz (Objekt) methoden

- Instanz (Objekt)-Methoden sind alle Methoden im direkten Scope der Klasse **ohne** den *static* Modifikator.
- Objektmethoden können nur auf Objekten der Klasse aufgerufen werden.
- Objektmethoden einer Klasse können die **Objektmethoden der selben Klasse** aufrufen, ohne ein Empfängerobjekt voranzustellen. Der Empfänger ist das implizite Objekt *this*.
- Alle Objektmethoden einer Klasse können die **sichtbaren Objektmethoden der Superklassen** aufrufen, ohne ein Empfängerobjekt voranzustellen. Der Empfänger ist auch hier das implizite Objekt *this*.

```
public class Counter {  
    private int count;  
    public Counter(){  
        super();  
        count =0;  
    }  
    public Counter inc(){  
        setCount(count+1);  
        return this;  
    }  
    protected void setCount(int count){  
        this.count = count;  
    }  
    public int getCount(){return count;}  
    public void reset(){setCount(0);}  
}
```



Instanz (Objekt)-Methoden

- Instanz (Objekt)-Methoden sind alle Methoden im direkten Scope der Klasse **ohne** den *static* Modifikator.

Im Bsp.: *getCount, inc, reset, setCount*

- Alle Objektmethode einer Klasse können die Objektmethode der selben Klasse aufrufen, ohne ein Empfängerobjekt voranzustellen. Der Empfänger ist das implizite Objekt *this*.

Im Beispiel: *reset* ruft *setCount(0)*, *inc()* ruft *getStep()* ohne Empfänger auf.

```
public class Counter {  
    private int count;  
    public Counter(){  
        super();  
        count =0;  
    }  
    public Counter inc(){  
        setCount(count+1);  
        return this;  
    }  
    protected void setCount(int count){  
        this.count = count;  
    }  
    public int getStep() {return 1;}  
    public int getCount(){return count;}  
    public void reset(){setCount(0);}  
}
```

Instanz-Methoden können nur auf Objekten aufgerufen werden



- Objektmethoden können nur auf Objekten der Klasse aufgerufen werden, nicht auf der Klasse selber.

Im Bsp.:

Der Aufruf der Methode `Counter.inc()` in der statischen `main` Methode erzeugt einen Compilerfehler, da für die Klasse die Objektmethode nicht definiert ist.

Der Aufruf der Methode `c.inc()` auf einem `Counter` Objekt hingegen ist ok.

```
public class Counter {  
    private int count;  
    public Counter(){...}  
    public Counter inc(){...}  
    protected void setCount(int  
        count){...}  
    public int getStep() {return 1;}  
    public int getCount(){...}  
    public void reset(){setCount(0);}  
}  
  
public static void main(String[]  
    args) {  
    Counter c = new Counter();  
    c.inc();  
    Counter.inc();  
    //Compilerfehler  
}  
}
```



Aufruf von Instanz (Objekt)-Methoden der Superklassen

- Alle Objektmethode einer Klasse können die **sichtbaren Objektmethode der Superklassen** aufrufen, ohne ein Empfängerobjekt voranzustellen. Der Empfänger ist auch hier das implizite Objekt *this*.

Im Beispiel: Die Methode *inc* von *StepCounter* ruft die Methoden *setCount* und *getCount* von Counter auf.

```
public class StepCounter extends
    Counter {
    private int step;
    public StepCounter(int step){
        super();
        this.step = step;
    }
    public int getStep(){return step;}
    public Counter inc(){
        setCount(getCount()+step);
        return this;
    }
}
```



Aufruf von überschriebenen Methoden der Superklassen

- Wenn eine Methode der Superklasse überschrieben wird, dann kann die überschriebene Methode der Superklasse über *super* erreicht werden.
- *super* ist hier eine Objektreferenz
Methodensuche in der Superklasse. Hier StepCounter
- **Im Beispiel:** In der Methode *inc* der Klasse *LimitedStepCounter* wird die Methode *inc* der Superklasse aufgerufen (*super.inc()*).

```
public class LimitedStepCounter
    extends StepCounter {

    private int limit;

    public LimitedStepCounter(int
        step,int limit){
        super(step);
        this.limit = limit;
    }

    public Counter inc() {
        if (getCount()+ getStep() >
            limit) {
            reset();
        }
        return super.inc();
    }
}
```



this /super versus this(...) /super(...)

`this(ard1...)` sind Konstruktoraufrufe

- **this** ist eine Referenz auf das Objekt, das die aktuelle Methode abarbeitet
- **super** ist eine Referenz auf ein Objekt. Die Methoden- und Attributsuche beginnt in der Superklasse der Klasse, deren Methodendefinition den Aufruf **super** enthält.
- Die Verwendung von **this** und **super** in der Bedeutung einer Objektreferenz ist in allen Objektmethoden und Konstruktoren erlaubt.
- **this(arg1, ..., argn)** ist der Aufruf eines Konstruktors der selben Klasse. Dies ist nur in Konstruktoren erlaubt. Konstruktorkaskade
- **super(arg1, ... argn)** ist der Aufruf eines Konstruktors der direkten Superklasse. Dies ist nur in Konstruktoren erlaubt.



Klassenmethoden und Klassenvariablen

STATISCHE METHODEN UND STATISCHE VARIABLEN



Klassenmethoden – statischen Methoden

- Manchmal gibt es Methoden in einer Klasse, die unabhängig vom Zustand konkreter Objekte der Klasse sind:
 - **Konvertierungsmethoden zwischen Datentypen:** Beispiele sind hier die Methode *valueOf, format* der Klasse *String* oder die *parse* Methoden der Wrapperklassen.
 - **Objekterzeugungsmethoden:** Beispiel *getDateInstance* Methode der Klasse *DateFormat*
 - **Utilitymethoden:** Beispiele sind hier die Methoden der Klassen *Math* und *Arrays, Collections*
- Diese Methoden arbeiten alle unabhängig vom Zustand der Objekte der Klasse und sind daher **Klassenmethoden** besser **statische Methoden**.
- Statische Methoden tragen immer den Modifikator *static*, der dem Sichtbarkeits-Modifikator folgt.



Beispiele für statische Methoden

```
public final class Math {
    public static double sin(double a) {
        return StrictMath.sin(a); // default impl. delegates to StrictMath
    }
}

public class Arrays {
    public static void sort(int[] a) {
        DualPivotQuicksort.sort(a, 0, a.length - 1, null, 0, 0);
    }
}

public final class String
    implements java.io.Serializable, Comparable<String>, CharSequence {

    public static String format(String format, Object... args) {
        return new Formatter().format(format, args).toString();
    }
    implemets kündigt Interface an
}

public final class Double extends Number implements Comparable<Double> {
    public static double parseDouble(String s) throws NumberFormatException {
        return FloatingDecimal.parseDouble(s);
    }
}
```



Klassenvariablen

- Manchmal gibt es Zustandsinformation, die nicht zu einem sondern zu allen Objekten der Klasse gehört. Bsp.:
 - eine Variable, die sich die Instanzen einer Klasse merkt.
 - eine Variable, die sich das einzige Objekt der Klasse merkt (Singleton Pattern).
 - eine Variable, die sich den Score Wert in einem Game merkt.
- Diese Zustandsinformation wird von allen Objekte gemeinsam benötigt bzw. manipuliert.
- **Klassenvariablen**, besser **statische Variablen**, sind Variablen, die einen gemeinsamen Speicher für alle Objekte der Klasse bereitstellen.
- Eine statische Variable gibt es **nur genau einmal** pro Klasse.
- Alle Objekte einer Klasse haben Zugriff auf dieselbe statische Variable.
- Statische Variablen werden initialisiert, wenn die Klasse geladen wird.
- Die häufigste Anwendung von Klassenvariablen sind die Klassenkonstanten:
 - die Konstanten **PI** und **E** der Klasse **Math**
 - die Konstanten **MIN_VALUE**, **MAX_VALUE** der Wrapperklassen.



Beispiele für Klassenkonstanten

```
public final class Math {  
    public static final double E = 2.7182818284590452354;  
    public static final double PI = 3.14159265358979323846;  
}  
  
public final class Double extends Number implements Comparable<Double> {  
    public static final double MAX_VALUE = 0x1.fffffffffffffP+1023;  
                                           // 1.7976931348623157e+308  
    public static final double MIN_VALUE = 0x0.0000000000001P-1022;  
                                           // 4.9e-324  
}
```



Eine statische Variable für den Score-Wert in einem Spiel

- Die Variable **score** merkt sich über alle Spieler (**Player**), den maximal erreichten Punktestand.
- Der **score** wird einmal initialisiert, wenn die Klasse **Player** geladen wird.
- Der **score** wird aktualisiert, wenn ein **Player** seinen Punktestand erhöht (**win()**).
- Alle Objekte schreiben in die gemeinsame Variable **score** in der Methode **win()**.
- Die Objektvariable **points** hält den Punktestand eines einzelnen **Player**.

```
public class Player {  
    private int points;  
    private static int score=0;  
  
    public static int getScore() {return  
        score;}  
  
    public Player(){points = 0;}  
  
    public Player win(){  
        points++;  
        score = Math.max(points, score);  
        return this;  
    }  
    public Player loose(){  
        points = Math.max(--points, 0);  
        return this;  
    }  
    public int getPoints() {return points;}  
}
```



Eine statische Variable für den Score in einem Spiel

```
public class PlayGame {  
  
    public static void main(String[] args) {  
        Player p1 = new Player();  
        Player p2 = new Player();  
        Player p3 = new Player();  
  
        p1.win().win().win();  
        p("score= " + Player.getScore());  
        p2.loose().loose().win();  
        p("score= " + Player.getScore());  
        for(int i = 0; i < 10; i++) {  
            p3.win();  
        }  
        p("score= " + Player.getScore());  
    }  
}
```

*Wir verändern für drei
Player deren individuellen
Punktstand durch
wiederholten Aufruf von
win oder loose.*

*score enthält für alle
Spieler den größten
erreichten Punktestand*



**score= 3
score= 3
score= 10**



Initialisieren von statischen Variablen

- Statische Variablen werden in **statischen Initialisierungsblöcken** initialisiert.
- Statischen Initialisierungsblöcken wird der Modifikator *static* vorangestellt.
- Statische Initialisierungsblöcke sind dann nützlich, wenn beim Laden einer Klasse statische Variablen vorbelegt werden müssen, sich die Vorbelegung aber nicht in einem Ausdruck formulieren lässt.
- Das nachfolgende Beispiel, eine Klassendefinition für Jahreszeiten, die eine Nachfolgerbeziehung enthalten, demonstriert die Notwendigkeit von statischen Initialisierungsblöcken.



Initialisieren von statischen Variablen

```
public final class Jahreszeit {  
    private String name;  
    private Jahreszeit succ;  
    public static final Jahreszeit FRUEHJAHR;  
    public static final Jahreszeit SOMMER;  
    public static final Jahreszeit HERBST;  
    public static final Jahreszeit WINTER;  
    static {  
        hier einmalige Initialisierung  
        WINTER = new Jahreszeit("Winter", null);  
        HERBST = new Jahreszeit("Herbst", WINTER);  
        SOMMER = new Jahreszeit("Sommer", HERBST);  
        FRUEHJAHR = new Jahreszeit("Frühjahr", SOMMER);  
        WINTER.setSucc(FRUEHJAHR);  
    }  
    private Jahreszeit(String name, Jahreszeit succ){  
        this.name = name;  
        this.succ = succ;  
    }  
  
    private void setSucc(Jahreszeit succ) {  
        this.succ = succ;  
    }  
}
```

Nachfolger von WINTER
noch nicht initialisiert →
Nachfolger nach
Initialisierung von
FRUEHJAHR setzen

WINTER.setSucc(FRUEHJAHR)
im direkten Scope der Klasse
nicht erlaubt → setzen im
static Initialisierungsblock



Von *final* Klassen darf nicht abgeleitet werden

- Im vorhergehenden Beispiel ist die Klasse *Jahreszeit final*, um das Erzeugen von Jahreszeitenobjekten über den Umweg von Subklassen zu unterbinden.
- Darf von einer Klasse nicht abgeleitet werden, dann muss der Klassendefinition der Modifikator *final* vorangestellt werden.
- Ein Versuch, von einer *final* Klasse abzuleiten, führt zu einem Compilerfehler.

```
class MyString extends String {  
    // Compilerfehler The type MyString cannot subclass the final  
    // class String  
}
```

- Die Java-Bibliothek enthält eine Reihe finaler Klassen, z.B., *String, StringBuffer, StringBuilder, Integer, Double, Math, System*

Regeln für statische Methoden und -variablen



- Statische Methoden und Variablen werden durch Voranstellen von *static* gekennzeichnet.
- Statische Variablen werden beim Laden der Klassen definiert. (Das nutzen wir für die Initialisierung von *score* mit 0 zu Spielbeginn.)
- Objekte der Klasse und deren Subklassen haben in den Objektmethoden Zugriff auf statische Methoden und Variablen, wenn diese sichtbar sind. (siehe Methode *Player.win*)
- Von außen muss auf statische Methoden und Variablen durch Voranstellen des Klassennamens zugegriffen werden. (siehe *Player.getScore* in *PlayGame*).
- Statische Methoden haben **keinen** Zugriff auf *this*, da *this* auch in statischen Methoden ein Objekt der Klasse bezeichnet und nicht die Klasse selber. (In Ruby war *self* im Klassen-Scope eine Referenz auf das Klassenobjekt)
- Statische Methoden haben **keinen** Zugriff auf *super*. Die Begründung ist analog *this*.



Statische Methoden haben keinen Zugriff auf *this*

- Der Aufruf einer Objektmethode in einer statischen Methode (z.B. *getPoints()* in *getPlayerPoints()*) ist äquivalent mit dem Aufruf der Methode auf *this* (*this.getPoints()*).
- Da *this* immer eine Referenz auf ein Objekt ist, Klassen in Java aber keine Objekte sind, ist der Aufruf fehlerhaft.
- Aber es ist möglich in statischen Methoden Objekte zu erzeugen und auf diesen dann Objektmethoden aufzurufen. (Im *main* der *PlayerDemo* werden Methoden auf *Player* Objekten aufgerufen.)

```
public class Player {  
  
    private int points;  
    private static int score=0;  
  
    public static int getScore()  
        {return score;}  
  
    public static int  
        getPlayerPoints() {  
        return getPoints();}  
  
    public int getPoints() {return  
        points;}  
    }  
}
```



Objekterzeugung in statischen Methoden

```
public class PlayGame {  
  
    public static void main(String[] args) {  
        Player p1 = new Player();  
        Player p2 = new Player();  
        Player p3 = new Player();  
  
        p1.win().win().win();  
        p("score= " + Player.getScore());  
        p2.loose().loose().win();  
        p("score= " + Player.getScore());  
        for(int i = 0; i < 10; i++) {  
            p3.win();  
        }  
        p("score= " + Player.getScore());  
    }  
}
```

Erzeugen von Player-Objekten und Aufruf von Objektmethoden auf Player-Objekten.



Nicht polymorphe Initialisierung von Objekten

KONSTRUKTOREN



Konstruktoren

- Konstruktoren sind spezielle Methoden einer Klasse **ohne** einen **Rückgabotyp**. Der implizite Rückgabewert ist ein Objekt der Klasse.
- Konstruktoren **müssen** wie die Klasse heißen.
- Konstruktoren können beliebig viele Parameter haben.
- Konstruktoren werden nach der Erzeugung eines Objektes **zur Initialisierung** aufgerufen.
- Der Name Konstruktor ist irreführend. Konstruktoren erzeugen keine Objekte sie initialisieren diese nur.
- Die aktuellen Parameter eines Konstruktors werden bei der Initialisierung den Instanz-Variablen zugewiesen.
- Vor der Initialisierung der Instanz-Variablen **muss** immer ein Konstruktor der Superklasse aufgerufen werden.
- Enthält ein Konstruktor einen Konstruktor-Aufruf, dann muss dieser immer an erster Stelle stehen.



Ein Konstruktor mit 2 Argumenten

```
public class LinienBus extends Bus {
```

```
    private String verkehrsverbund;  
    private String linie;
```

```
    public LinienBus(String verkehrsverbund, String linie) {  
        super();  
        this.verkehrsverbund = verkehrsverbund;  
        this.linie = linie;  
    }
```

Aufruf des
Konstruktors
der Superklasse

Initialisierung der Instanz-Variablen.
Da die formalen Parameter des Konstruktors
den gleichen Namen wie die Instanz-Variablen
haben, müssen die Instanz-Variablen durch
Vorstellen von this angesprochen werden.

Jede Klasse ohne Konstruktor hat einen Default-Konstruktor



- Ein Default-Konstruktor ist ein Konstruktor ohne Parameter.
- Jede Klasse **muss** einen Konstruktor enthalten!
- Der Compiler generiert für jede Klasse, die keinen Konstruktor enthält, einen Default-Konstruktor.
- Jeder Konstruktor muss **an erster Stelle** den Aufruf eines Konstruktors enthalten. Der Compiler generiert in jeden Konstruktor, der keinen Aufruf eines Konstruktors enthält, den Aufruf des Default-Konstruktors der Superklasse
- **Grund:** Damit soll die Initialisierung nach einer vorgegebenen Reihenfolge sichergestellt werden.

Jede Klasse ohne Konstruktor hat einen Default-Konstruktor



```
public class Fahrrad extends Fahrzeug
{
    public Fahrrad() {
        super();
    }
}
```



Der Default-Konstruktor in *Fahrrad* ist überflüssig: Er wird generiert.



Der Aufruf des Konstruktors der Superklasse in *Fahrrad* ist überflüssig: Er wird generiert.

Custom-Konstruktoren müssen in Subklassen explizit aufgerufen werden



- Custom-Konstruktoren sind Konstruktoren, die mindestens einen Parameter haben.
- Wenn eine Klassen einen Custom-Konstruktor hat, dann generiert der Compiler keinen Default-Konstruktor für die Klasse.
- Subklassen **müssen** dann den Custom-Konstruktor explizit aufrufen.
- Im Beispiel, wenn die Klasse *PersonenKraftfahrzeug* einen Custom-Konstruktor enthält, dann muss ein Konstruktor der Klasse *Bus* diesen Konstruktor explizit aufrufen.

Custom-Konstruktoren müssen in Subklassen explizit aufgerufen werden



```
public abstract class PersonenKraftfahrzeug extends Fahrzeug {
```

```
private String antriebsart = "";
```

```
...
```

```
public PersonenKraftfahrzeug(String antriebsart, double co2,  
    double hoechstgeschwindigkeit, double kraftstoffverbrauch,  
    int leistung, int plaetze) {  
    super();
```



Custom-Konstruktor, einziger
Konstruktor der Klasse.

```
this.antriebsart = antriebsart;
```

```
this.co2 = co2;
```

```
this.hoechstgeschwindigkeit = hoechstgeschwindigkeit;
```

```
this.kraftstoffverbrauch = kraftstoffverbrauch;
```

```
this.leistung = leistung;
```

```
this.plaetze = plaetze;
```

```
}
```

Custom-Konstruktoren müssen in Subklassen explizit aufgerufen werden



```
public class Bus extends PersonenKraftfahrzeug {  
  
    private boolean lizenz;  
    private List<Integer> tarife;  
  
    public Bus(boolean lizenz, List<Integer> tarife) {  
        this.lizenz = lizenz;  
        this.tarife = tarife;  
    }  
}
```



kein Konstruktor-Aufruf.
Der Compiler ergänzt super().
Superklasse hat nur einen
Custom-Konstruktor →
Compilerfehler.

```
Bus.java X PersonenKraftfahrzeug Schiff.java »11  
1 package fahrzeugev1;  
2  
3 import java.util.List;  
4  
5 public abstract class Bus extends PersonenKraftfa  
6  
7     private boolean lizenz;  
8     private List<Integer> tarife;  
9  
10    public Bus(boolean lizenz, List<Integer> tarife) {  
11        this.lizenz = lizenz;  
12        this.tarife = tarife;  
13    }  
}
```

Implicit super constructor PersonenKraftfahrzeug() is undefined. Must explicitly invoke another constructor

Custom-Konstruktor müssen in Subklassen explizit aufgerufen werden



```
fahrzeugev1  Bus.java  Fährschiff.java  PersonenKraftfahrzeug  Schiff.java  »9
1 package fahrzeugev1;
2
3 import java.util.List;
4
5 public abstract class Bus extends PersonenKraftfahrzeug {
6
7     private boolean lizenz;
8     private List<Integer> tarife;
9
10    public Bus(boolean lizenz, List<Integer> tarife, String antriebsart,
11               double co2, double hoechstGeschwindigkeit,
12               double kraftstoffverbrauch, int leistung, int plaetze) {
13        super(antriebsart, co2, hoechstGeschwindigkeit, kraftstoffverbrauch,
14              leistung, plaetze);
15        this.lizenz = lizenz;
16        this.tarife = tarife;
17    }
18 }
```



Korrekte, korrigierte
Implementierung des
Konstruktors in der Klasse
Bus.



Konstruktor-Kaskaden

- Konstruktoren können Konstruktoren der gleichen Klasse mit der Methode *this()* aufrufen.
- *this()* darf beliebig viele Argumente enthalten.
- **Im Beispiel:** Der 4 stellige Konstruktor *Bus* ruft mit *this(lizenz,..., plaetze)* den 11-stelligen Konstruktor von *Bus* auf.

```
fahrzeugev1 | Bus.java | Fährschiff.java | PersonenKraftfahrzeu | Schiff.java | »9
18
19 public Bus(boolean lizenz, List<Integer> tarife, double co2, int plaetze) {
20     this("Unknown", "Unknown", 1000, lizenz, tarife, "Diesel", co2, 10, 20,
21         250, plaetze);
22
23 }
24
25 public Bus(String name, String hersteller, double gewicht, boolean lizenz,
26     List<Integer> tarife, String antriebsart, double co2,
27     double hoechstgeschwindigkeit, double kraftstoffverbrauch,
28     int leistung, int plaetze) {
29     super(name, hersteller, gewicht, antriebsart, co2,
30         hoechstgeschwindigkeit, kraftstoffverbrauch, leistung, plaetze);
31     this.lizenz = lizenz;
32     this.tarife = tarife;
33 }
34
```



Konstruktor-Regeln zusammengefasst

1. Jede Klassendefinition muss **mindestens einen** Konstruktor enthalten.
2. Der Compiler generiert den Default-Konstruktor für eine Klasse, dann und nur dann, wenn eine Klasse **keinen** Konstruktor enthält.
3. Jeder Konstruktor **muss** an erster Stelle einen gültigen Konstruktor aufrufen, entweder einen der eigenen oder einen der Super-Klasse.
4. Der Compiler generiert den Aufruf des Default-Konstruktors der Superklasse in die Konstruktoren einer Klasse, die Regel 3 nicht beachten.
5. Jeder Konstruktor darf **nur genau einen** Konstruktor aufrufen.



Typisierung



Typisierung

- **Definitionen:**

- Ein Typ ist eine genaue Charakterisierung der gemeinsamen Eigenschaften einer Menge von Einheiten (Objekten) (**Booch**).
- Typisierung erzwingt, dass Objekte unterschiedlicher Typen nicht oder nur eingeschränkt gegeneinander ersetzt werden können.
- **Regel für das eingeschränkte Ersetzen - Liskov'sches Substitutions-Prinzip:** Wenn $S \subset T$ (lies S ein Subtyp von T ist), dann kann für jede Instanz von T eine Instanz von S eingesetzt werden, ohne dass sich das Verhalten des Programmes ändert.

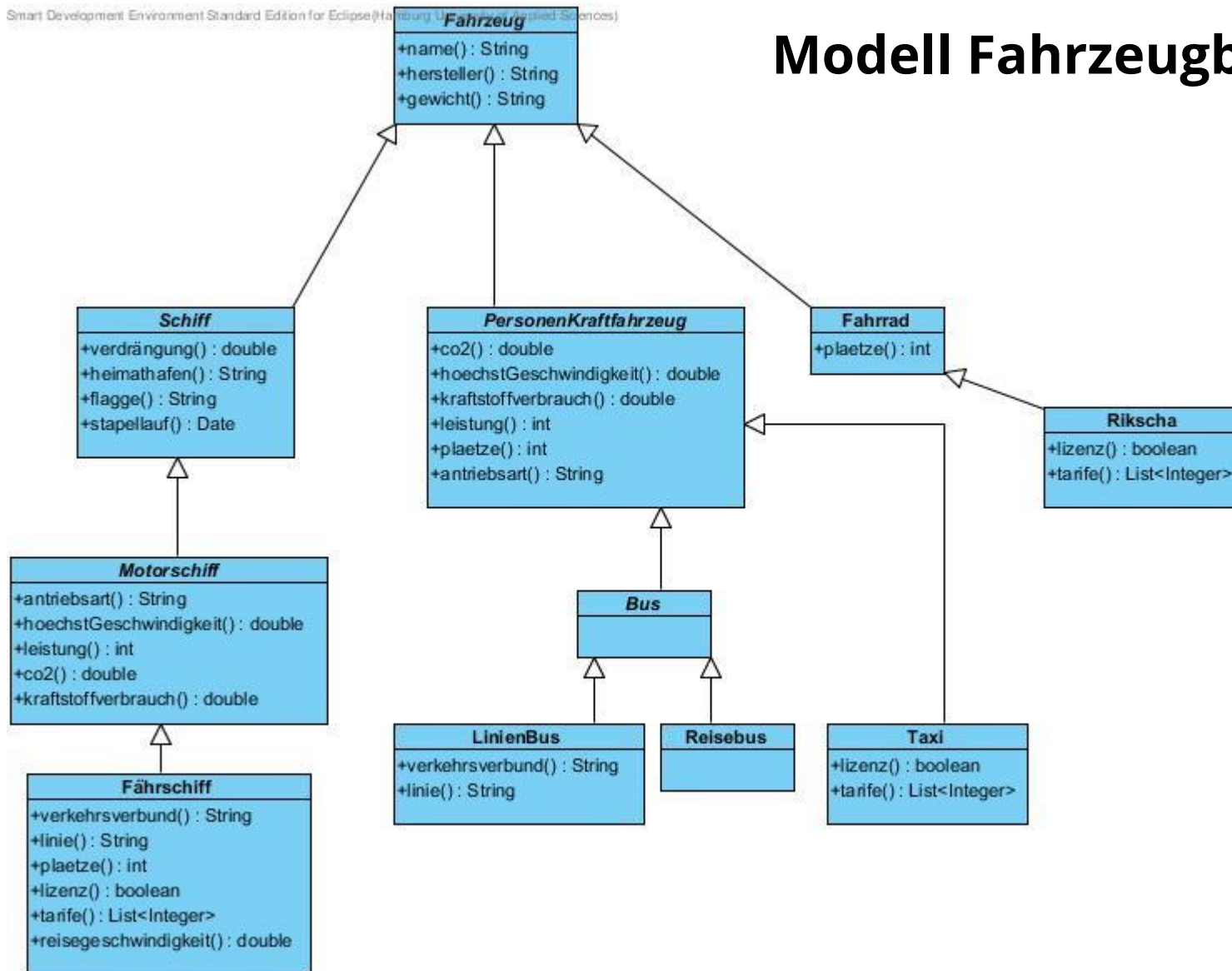


Wozu Typisierung

- Ein Typ spezifiziert das Verhalten von Objekten.
 - Es dürfen keine illegalen Operationen für ein Objekt ausgeführt werden.
 - Legal sind nur Operationen, die das Objekt aufgrund seiner Typzugehörigkeit anbietet.
- **Frage:** Zu welchem Zeitpunkt wird die Typsicherheit sichergestellt?
 - **Statische Typisierung:** zur Compilezeit
 - **Dynamische Typisierung:** zur Laufzeit



Modell Fahrzeugbeispiel





Beispiel: Die Typen einer Instanz von *Linienbus*

- Erzeugen wir ein Objekt der Klasse *Linienbus*, dann hat dieses Objekt, den Typ *Linienbus*, *Bus*, *PersonenKraftfahrzeug* und *Fahrzeug*.
- Wir sagen der Typ des Objektes ist kompatibel zu den genannten Typen.
- Es darf daher Variablen aller 4 genannten Typen zugewiesen werden.
- Eine Zuweisung auf eine Variable vom Typ Schiff hingegen ist nicht möglich.

```
*LinienbusTypenDemo. X Bus.java Fährschiff.java PersonenKraftfahrzeu Schiff.j
1 package fahrzeugev1;
2
3 import java.util.ArrayList;
4
5 public class LinienbusTypenDemo {
6
7     public static void main(String[] args) {
8
9         LinienBus lb = new LinienBus("HVV", "N13", true,
10             new ArrayList<Integer>(), "Diesel", 30, 250 ,100,
11             130);
12         Bus b = lb;
13         PersonenKraftfahrzeug pkf = lb;
14         Fahrzeug fz = lb;
15
16         Schiff sc = lb;
17     }
18 }
```



Statische Typisierung

- Die Typinformation wird mittels Deklarationen im Programm-Quelltext an Variablen gebunden, in
 - Variablendeklaration oder
 - in formalen Methoden-Parameter
 - `String str; public boolean equals (Object o)`
- Der Compiler überprüft zur **Übersetzungszeit**, ob ein Variablen-Typ und ein Objekt-Typ kompatibel sind. Zwei Typen sind kompatibel, wenn
 - die Namen der Typen exakt gleich sind, oder
 - der Typ der Variable ein Supertyp des Objekttyps ist.
- Das gewährleistet zur Definitionszeit des Programmes, dass nur legale Operationen auf Objekten aufgerufen werden: Es können nur die Methoden aufgerufen werden, die in der eigenen Klasse oder einem der Supertypen definiert sind.
- Der bei der Deklaration einer Variablen verwendete Typ ist der **statische Typ** einer Variable.

Statische Typisierung gewährleistet, dass nur legale Operationen verwendet werden können



- Der statische Typ von *a* ist *A*. Methode *mb()* ist in *A* nicht definiert. Der Aufruf von *mb()* auf *a* erzeugt daher einen Compiler-Fehler.

```
class A {  
    void ma() {p("Legal for type A");}}  
class B extends A {  
    void mb() {p("Legal for type B");}}
```

```
class StaticTypeLegalOps {  
    public static void main(String[] args) {  
        statischer Typ A a = new B();  
        a.ma();    //ok  
        a.mb();    // Compilerfehler  
        // The method mb() is undefined for the type A  
    }  
}
```



Dynamische Typisierung und dynamisches Binden

- Typ-Prüfung wird beim Methodenaufruf zur Laufzeit implizit durchgeführt. D.h.
 - Methoden müssen in der Empfängerklasse oder einer Oberklasse gefunden werden.
- Das Objekt, das einer Variablen zugewiesen wird, entscheidet zur Laufzeit über den Typ der Variable: Wir sprechen auch vom **dynamischen** Typ der Variable.
- Der Typ des Objektes entscheidet darüber, welche Methodendefinition zur Laufzeit gewählt wird. (**dynamisches Binden**).
- **Java:** Der **statische Typ** einer Variablen gewährleistet, dass nur legale Operationen auf der Variablen ausgeführt werden. Der Compiler prüft dies bei der Übersetzung.
- Dennoch werden die Methoden (bis auf einige Ausnahmen) zur Laufzeit **dynamisch gebunden**.



Dynamische Typisierung und dynamisches Binden

- Die Typ-Prüfung wird beim Methodenaufruf zur Laufzeit durchgeführt. D.h. Bei der Methodensuche müssen die verwendeten Methoden in der Empfängerklasse oder einer Superklasse gefunden werden.
- Das Objekt, das einer Variablen zugewiesen wird bestimmt zur Laufzeit den Typ der Variable. Wir sprechen auch vom **dynamischen** Typ der Variable.
- Der Typ des Objektes, also der dynamische Typ der Variable bestimmt, wo die Methodensuche beginnt. Die erste Methodendefinition, die entlang der Vererbungshierarchie gefunden wird, wird ausgeführt. Dieses Verhalten wird **dynamisches Binden** genannt.

Statische Typisierung und dynamisches Binden



- **Java ist statisch typisiert:**
 - Bei der Definition des Quelltextes muss jede Variable mit einem statischen Typ deklariert werden.
 - Der **statische Typ** gewährleistet, dass zur Definitionszeit nur legale Methoden aufgerufen werden können. Der Compiler prüft dies bei der Übersetzung des Quelltextes.
 - Dennoch werden die Methoden (bis auf einige Ausnahmen) zur Laufzeit **dynamisch gebunden**. Die virtuelle Maschine führt nach den gleichen Prinzipien, die wir aus Ruby kennen, zur Laufzeit eine Methodensuche durch.

In Java werden Objektmethoden dynamisch gebunden



- Der dynamische Typ von *c1* ist *D* es wird *mc* in *D* aufgerufen.
- Der dynamische Typ von *c2* ist *C*: es wird *mc* in *C* aufgerufen.

```
public class DynamicBindOps {  
    public static void main(String[] args) {  
        C c1 = new D();  
        c1.mc();  
        C c2 = new C();  
        c2.mc();  
    }  
}
```



mc in D
mc in C

```
class C {  
    void mc() {p("mc in C");}}  
class D extends C {  
    void mc() {p("mc in D");}}}
```



Statisches Binden

- Statisches Binden legt die **Methoden und Attribut-Auswahl zur Compilezeit** fest.
- Der Typ der Variable oder der in einem **Cast** konstatierte Typ eines Objektes entscheidet über die verwendete Methodendefinition oder das Attribut.
- Statisch gebunden werden in Java: ^{KA}
 - **statische Attribute und Methoden:** richten sich an eine Klasse und nicht an ein Objekt.
 - **Konstruktoren**
 - **private Attribute und Methoden:** sind außerhalb der Klasse nicht sichtbar.
weil private Methoden nicht vererbt werden
 - **alle Attribute:** alle Instanz-Variablen unabhängig von ihrer Sichtbarkeit. Beim Duplizieren von Attributen verdecken sich die Attribute der Super und Subklasse gegenseitig. (siehe nachfolgendes Beispiel)
 - **final** Methoden und Attribute, da diese in Subklassen nicht verändert / überschrieben werden können.

Statisch gebunden werden statische Attribute und Methoden



s.h.Foto mit Object Person

- Das Attribut *data* und die Methode *pMe* werden vom Compiler statisch gebunden.
- Der Compiler legt fest, welche Methode zur Laufzeit gewählt wird.
- Ein Type-Cast bewirkt, dass der Compiler *sb* als *StaticDerived* behandelt.
- Daher werden hier auch die statischen Methoden und Attribute der Klasse *StaticDerived* verwendet.

```
StaticBase sb = new StaticDerived();  
p(sb.data);  
p(((StaticDerived)sb).data);  
sb.pMe();  
((StaticDerived)sb).pMe();
```

1
2
this is StaticBase
this is StaticDerived


```
class StaticBase{  
    static int data = 1;  
    static void pMe(){  
        p("this is StaticBase");  
    }  
}  
  
class StaticDerived extends StaticBase {  
    static int data = 2;  
    static void pMe(){  
        p("this is StaticDerived");  
    }  
}
```

Attribute unabhängig von der Sichtbarkeit werden statisch gebunden



- **Alle** Attribute / Instanz-Variablen werden statisch gebunden. (1'te / 4'te Ausgabe)
- Objektmethoden werden **nicht** statisch gebunden. (2'te / 3'te Ausgabe)
- **Vorsicht:** Beim Duplizieren von Attributen verdecken sich die Attribute der Super- und Subklasse gegenseitig.

```
Base b = new Derived();  
p(b.data);  
b.md();  
p(((Derived) b).data);
```



1
2
1
2

```
class Base {  
    int data = 1;  
    public void md() {  
        p(data);  
    }  
}  
  
class Derived extends Base {  
    int data = 2;  
    public void md() {  
        p(data);  
        super.md();  
    }  
}
```



Statische und dynamische Typprüfung in Java

Typkorrekte Zuweisung

```
Person person = new Person();  
Student student = new Student();  
person = student;
```

Laufzeit: *person* und *student* referenzieren
exakt dasselbe *Student*-Objekt (Aliase)

Compile-Zeit: in *person* ist der *student*-
Anteil nicht mehr sichtbar.

Object obj = student: lässt nur noch den
Object-Anteil des *student*-Objektes
sichtbar stehen.

```
class Person {  
    String name, vorname;  
    ...  
}
```

```
class Student extends Person {  
    String matnr, semester;  
    ...  
}
```



Statische und dynamische Typprüfung

```
package binding;
import static util.Printer.*;

class StaticDynamicBinding {
    public static void main(String[] args) {
        Person person = new Person();
        Student student = new Student();

        person = student; // ok: Typ Student kompatibel zu Typ Person

        p(person.getVorname()); // ok: Typ Person hat Methode getVorname

        p(((Student)person).getMatnr());
        // ok: Downcast macht die Student Aspekte von person sichtbar
        // aber Vorsicht kann zur Laufzeit zu einer ClassCastException führen

        p(student.getName()); // ok: Methode getName im Supertyp Person
        p(((Person)student).getName());
        // ok Upcast (Widening): Typ Person hat Methode getName()
    }
}
```



Type Casts

```
SomeType st;  
st = (SomeType) anObject;
```

- Ein Type-Cast verändert den **statisch** Typ einer Variablen zur **Compile-Zeit**.
Objkttyp (dynamischer Typ wird nicht verändert)
- Der Type-Cast ist eine Anweisung an den Compiler:
➔ interpretiere *anObject* wie ein Objekt vom Typ *SomeType*
- Der Type-Cast auf *SomeType* ist nur legal, wenn *anObject* zur Laufzeit an ein Objekt vom Typ *SomeType* gebunden ist.
- Ob dies der Fall ist, kann nur zur Laufzeit geprüft werden.



Type Casts

- Wir unterscheiden zwei Arten von Casts.
 - **Downcast:** Der Typ eines Objektes wird auf einen kleineren Typen gecastet. Bsp.:
Person p = new Person();
Student s = (Student)~~person~~;
 - **Upcast:** Der Typ eines Objektes wird auf einen größeren Typen gecastet. Bsp.:
Student s = new Student();
Person p = (Person) s;
- Zur Compilezeit wird nur überprüft, ob grundsätzlich ein Cast möglich ist
- Casts sind nur entlang der Vererbungshierarchie erlaubt.

Antibeispiel:

```
Person p = new Person();  
String s = (String)person; // FEHLER
```

hier nu p. gleicher Fehler oben

Person & String stehen nebeneinander, es besteht also keine Vererbungshierarchie. Würde hier Object stehen wäre das Casten möglich



Typobjekte

- Jedem Typ ist ein eindeutiges Typobjekt *type.class* zugeordnet.
- Daher ist der Vergleich auf Identität (*==*) ausreichend.
- Das Typobjekt zur Klasse eines Objektes erhält man durch *object.getClass()*
- Typobjekte sind Objekte und selbst Instanzen vom Typ *Class*.

```
class staticdynamicbinding.Derived
class staticdynamicbinding.Base
false
true
true
true
true
class java.lang.Class
true
```

```
Base b1 = new Derived();
Base b2 = new Base();
Class cB1 = Derived.class;
Class cB2 = Base.class;
p(cB1);p(cB2);
p(cB1==cB2);
p(cB1== b1.getClass());
p("hello".getClass()== String.class);
p(cB1 instanceof Class);
p("hello".getClass() instanceof Class);
p(cB1.getClass());
p(cB1.getClass() instanceof Class);
```





Zusammenfassung

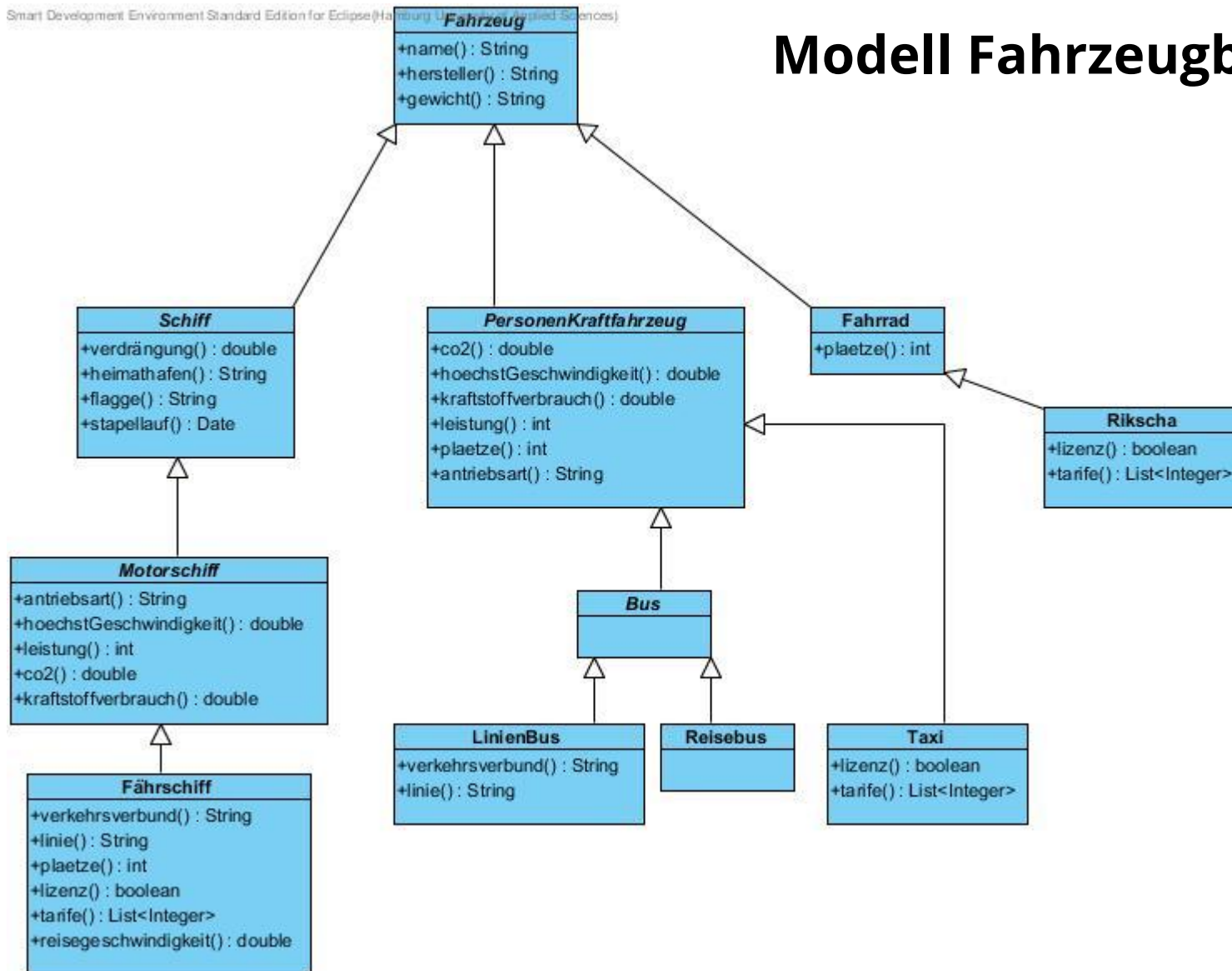
- **Typisierung** = Konformität der Typen in einem Ausdruck
- **Statische Typisierung** = Konformität wird zur Compilezeit überprüft.
- Entschärfung durch dynamisches Binden auch *late binding*.
- **Dynamische Typisierung** = Konformität wird zur Laufzeit überprüft.
- Flexibilität durch **Polymorphie**
 - Polymorphie erlaubt neben der Verwendung des passenden Typs auch andere, kompatible Typen.
- **Ausblick:** Interfaces sind ein Konzept, zur Flexibilisierung statischer Typisierung in Java



INTERFACES



Modell Fahrzeugbeispiel



Redundanzen beseitigen ohne Mehrfachvererbung im Fahrzeugbeispiel



- Wir haben im ersten Klassenentwurf für Fahrzeuge eine Reihe von Redundanzen entdeckt.
- **Lösungsidee:** Redundanzen in Klassen abstrahieren und Subklassen von mehreren Klassen ableiten lassen. (Lösen wir in Ruby über Mixins.)
- **Aber:** Java erlaubt **keine Mehrfachvererbung von Klassen.**

Redundanzen beseitigen ohne Mehrfachvererbung im Fahrzeugbeispiel



- Gehen wir noch einen Schritt weiter und überlegen uns drei Anwendungen für das Fahrzeugbeispiel.
 1. Die Stadt möchte alle **Personenbeförderungen** auf gültige Lizenzen überprüfen. Ergebnis ist eine Liste mit ungültigen Lizenzen.
 2. Das Umweltamt führt eine Erhebung zum potentiellen CO2 Ausstoß aller **motorisierten** Fahrzeuge durch.
 3. Ein ÖPNV Unternehmen möchte überprüfen, ob das Fahrgastaufkommen für Fahrzeuge einer Strecke durch die **öffentlichen Verkehrsmittel** abgedeckt werden.
- In diesen drei Anwendungen werden unterschiedliche Klassen unter 3 verschiedenen Aspekten betrachtet.
- Dabei kann eine Klasse durchaus in alle 3 Kategorien gehören. So ist ein **Fährschiff** sowohl motorisiert als auch eine Personenbeförderung als auch ein öffentliches Verkehrsmittel.



Anwendung 1

- Die Methode *lizenzenPruefen* prüft Lizenzen auf Listen von Objekten, die vom Typ *PersonenBefoerderung* sind.
 - In unserem Beispiel sind dies *Fährschiffe*, *Linienbusse*, *Taxis* und *Rikschas*.
- ➔ die genannte Klassen sollten den gemeinsamen Typ *PersonenBefoerderung* haben, damit nur eine Methode *lizenzenPruefen* für alle Klassen existiert.

```
private static List<PersonenBefoerderung> lizenzenPruefen(  
    List<PersonenBefoerderung> lpb) {  
    List<PersonenBefoerderung> mitLizenz = new ArrayList<PersonenBefoerderung>();  
    for (PersonenBefoerderung pb : lpb) {  
        if(! pb.lizenz()){  
            ohne mitLizenz.add(pb);  
        }  
    }  
    return mitLizenz;  
}
```



Anwendung 2

- Die Methode *co2Ausstoss* summiert den Schadstoffausstoß der motorisierten *Fahrzeuge*. In unserem Klassenmodell sind das *Motorschiffe* und *PersonenKraftfahrzeuge*.
- ➔ die genannten Klassen sollten den gemeinsamen Typ *Motorisiert* haben.
- *Taxis*, *Linienbusse*, *Fährschiffe* sollen zwei weitere Typen haben: *Motorisiert* und *PersonenBefoerderung*.
- *Reisebusse* und *Rikschas* jeweils nur einen zusätzlichen Typ.

```
private static double co2Ausstoss(List<Motorisiert> lm) {  
    int ausstoss = 0;  
    for (Motorisiert motorisiert : lm) {  
        ausstoss += motorisiert.co2();  
    }  
    return ausstoss;  
}
```




Anwendung 2

- Die Methode *co2Ausstoss* summiert den Schadstoffausstoß der motorisierten *Fahrzeuge*. In unserem Klassenmodell sind das *Motorschiffe* und *PersonenKraftfahrzeuge*.
- die genannten Klassen sollten den gemeinsamen Typ *Motorisiert* haben.

```
private static double co2Ausstoss(List<Motorisiert> lm) {  
    int ausstoss = 0;  
    for (Motorisiert motorisiert : lm) {  
        ausstoss += motorisiert.co2();  
    }  
    return ausstoss;  
}
```

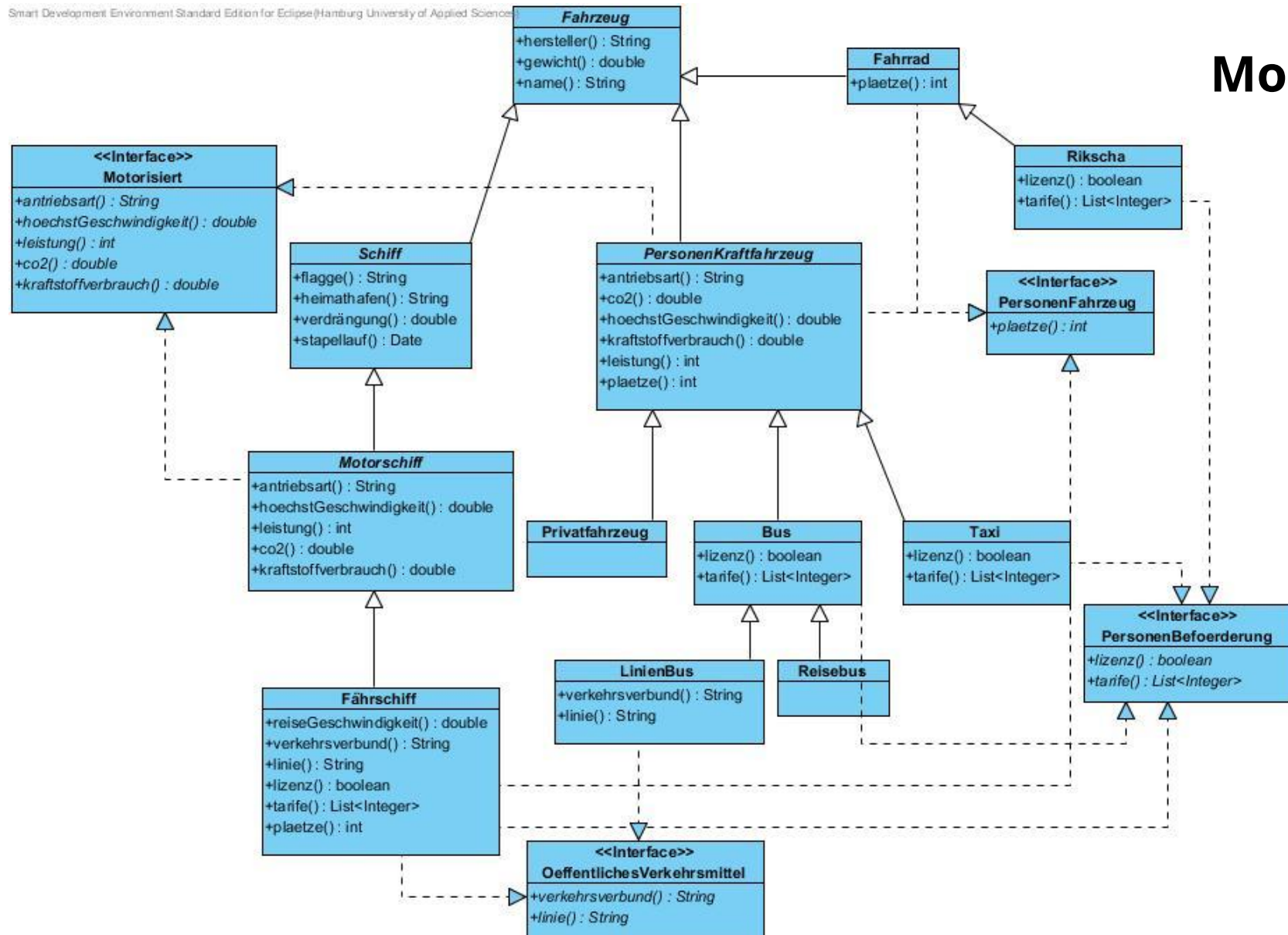


Vorbereitung Anwendung 3

- Zuerst werfen wir erneut einen Blick auf die Klassen des Modells. Die Eigenschaft *plaetze*, die die Anzahl der Passagiere eines Fahrzeugs beschreibt, ist sicher nicht für LKW's oder Containerschiffe gültig, sondern nur für *PersonenFahrzeug*.
- Wir extrahieren diese Eigenschaft als separaten Typ, so dass wir später Kraftfahrzeuge in Personenkraftfahrzeuge und Nutzfahrzeuge unterscheiden können.
- Die Klassen *Faehrschiff*, *Fahrrad* und *PersonenKraftfahrzeug* sind alle vom Typ *PersonenFahrzeug*.
- Alle bisher neu eingeführten Typen bilden wir in Java als Interfaces ab.
- Die Mehrfach-Typeigenschaften bilden wir ab, indem die Klassen Interfaces implementieren. (Pfeile mit gestrichelter Linie)
- Die Vererbung zwischen Klassen werden wie zuvor als Pfeile mit durchgezogener Linie dargestellt.



Modell





Auszug aus den Klassen und Interfaces

Package fahrzeugev2

```
public class Faehrschiff extends Motorschiff
    implements OeffentlichesVerkehrsmittel,
        PersonenBefoerderung, PersonenFahrzeug {...}
```

```
public interface OeffentlichesVerkehrsmittel {
    public String verkehrsverbund();
    public String linie();
}
```

- Klassen „leiten“ von Interfaces ab, indem sie diese implementieren (*implements*).
- Klassen können **mehr als ein** Interface implementieren.
- Ein **Interface** ist eine Sammlung von Methoden-Rümpfen ohne Implementierung. Es spezifiziert ein Protokoll für implementierende Klassen. Der Compiler stellt sicher, dass konkrete Klassen die Methoden eines Interfaces implementieren.



Anmerkung zu den folgenden Codebeispielen

- In dem nachfolgenden Beispielcode verfügen alle Klassen aus **Gründen der Vereinfachung in der Darstellung** über einen Default-Konstruktor, der alle Instanz-Variablen vorbelegt.
- Wir werden abschließend die Technik der **Delegation** kennenlernen, um die Parameter-inflation in den Konstruktoren des ersten Modells zu reduzieren.



Anwendung 1

Klasse *Anwendungenv2.java*

- Wir erzeugen Objekte der Klassen *Linienbus*, *Faehrschiff*, *Rikscha* und *Taxi*, die alle das Interface *PersonenBefoerderung* implementieren.
- Objekte der unterschiedlichen Klassen können dann in eine Liste mit Komponententyp *PersonenBefoerderung* eingetragen werden und der Methode *lizenzenPruefen/1* übergeben werden.

```
private static void lizenzenPruefen() {  
    List<PersonenBefoerderung> lpb = new ArrayList<PersonenBefoerderung>();  
    LinienBus b1 = new LinienBus();  
    Faehrschiff f1 = new Faehrschiff();  
    Rikscha rk = new Rikscha();  
    Taxi tx = new Taxi();  
  
    lpb.add(b1);  
    lpb.add(f1);  
    lpb.add(rk);  
    lpb.add(tx);  
  
    List<PersonenBefoerderung> lpbl= lizenzenPruefen(lpb);  
    p(lpbl);  
}
```



Anwendung 2

Klasse *Anwendungenv2.java*

- Da Objekte der Klassen *Faehrschiff*, *LinienBus*, *Reisebus*, *Taxi*, *Privatfahrzeug* das Interface *Motorisiert* implementieren, können wir diese in eine Liste von *Motorisiert* Objekten eintragen und den Schadstoffausstoß berechnen.

```
private static void co2Ausstoss() {  
    List<Motorisiert> lm = new ArrayList<Motorisiert>();  
    Fährschiff f1 = new Fährschiff();  
    LinienBus b2 = new LinienBus();  
    Reisebus r1 = new Reisebus();  
    Privatfahrzeug pf = new Privatfahrzeug();  
    Taxi tx = new Taxi();  
    lm.add(f1);  
    lm.add(b2);  
    lm.add(r1);  
    lm.add(pf);  
    lm.add(tx);  
    double co2 = co2Ausstoss(lm);  
    p("CO2 " + co2);  
}
```



Anwendung 3

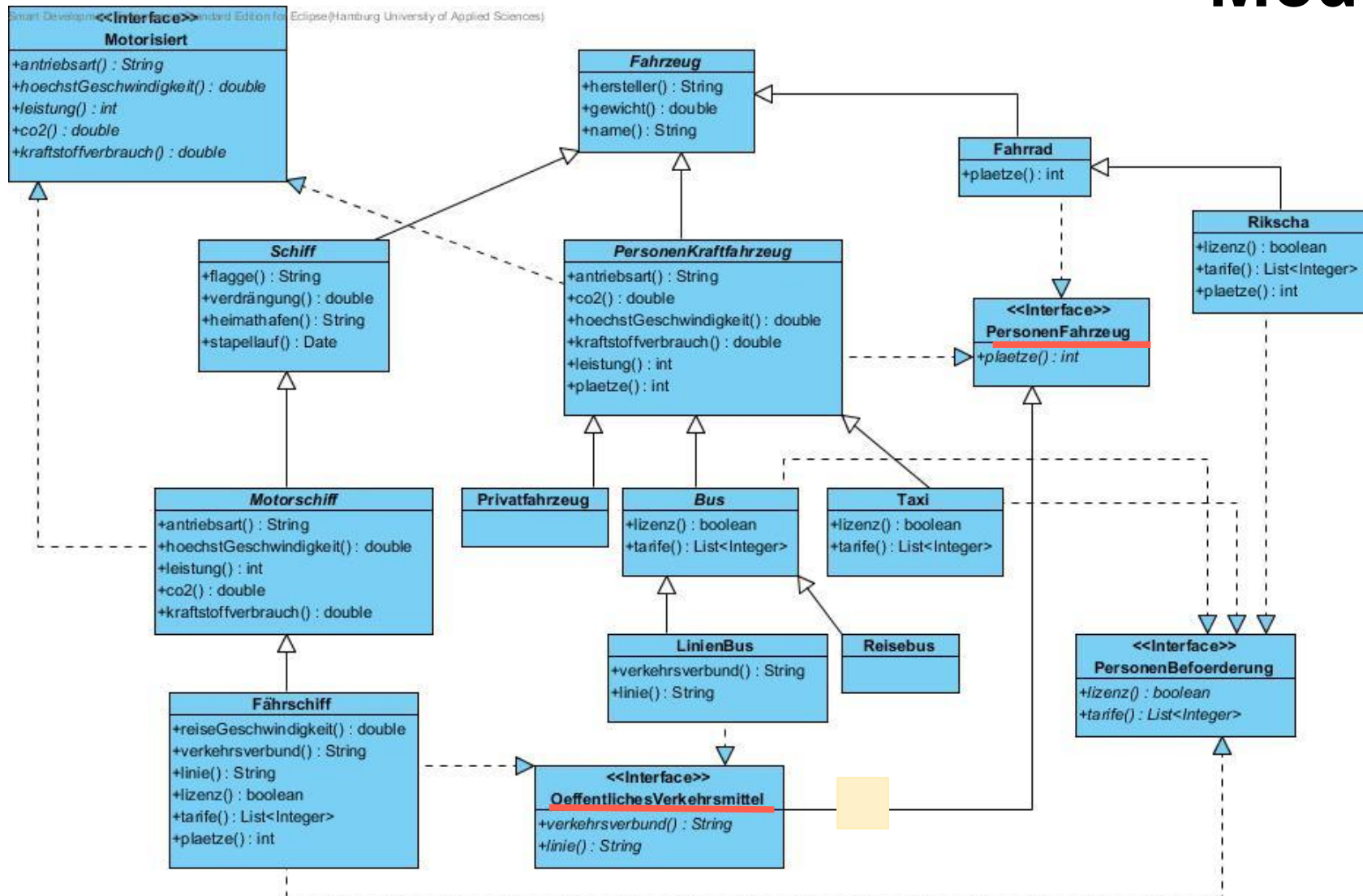
Klasse *Anwendungenv2.java*

- Schauen wir uns die Methode zur Berechnung der Kapazität für das neue Modell an, dann erhalten wir einen Fehler. Ursache: *OeffentlichesVerkehrsmittel* kennen die Methode *plaetze* nicht.
- Um Anforderung wie in Anwendung 3 realisieren zu können, müssen wir das Interface *OeffentlichesVerkehrsmittel* um das Interface *PersonenFahrzeug* erweitern, in dem wir vom Interface *PersonenFahrzeug* ableiten.
- Das führt zu der endgültigen Fassung des Modells für Fahrzeuge auf der nachfolgenden Seite.

```
99
100 private static boolean kapazitaetPruefen(
101     List<OeffentlichesVerkehrsmittel> loev, int i) {
102     int kap = 0;
103     for (OeffentlichesVerkehrsmittel oev : loev) {
104         kap += oev.plaetze();
105         if (kap < i) return false;
106     }
107     return true;
108 }
```




Modell





Interfacevererbung

```
public interface OeffentlichesVerkehrsmittel extends PersonenFahrzeug {  
    ...}  
public class Fährschiff extends Motorschiff  
    implements OeffentlichesVerkehrsmittel,  
        PersonenBefoerderung { ... }
```

- Interfaces können von anderen Interfaces ableiten.
- Das Protokoll für implementierende Klasse erweitert sich dann um die Methoden des Superinterfaces.
- Interfaces können auch von mehreren Interfaces ableiten, die dann komma-separiert nach dem *extends* aufgezählt werden.
- Mit dieser letzten Modifikation des Modells, einer Kombination von zwei Interfaces, können wir Anwendung 3 korrekt ausführen (Package *fahrzeugev3*, Klasse *Anwendungenv3.java*).

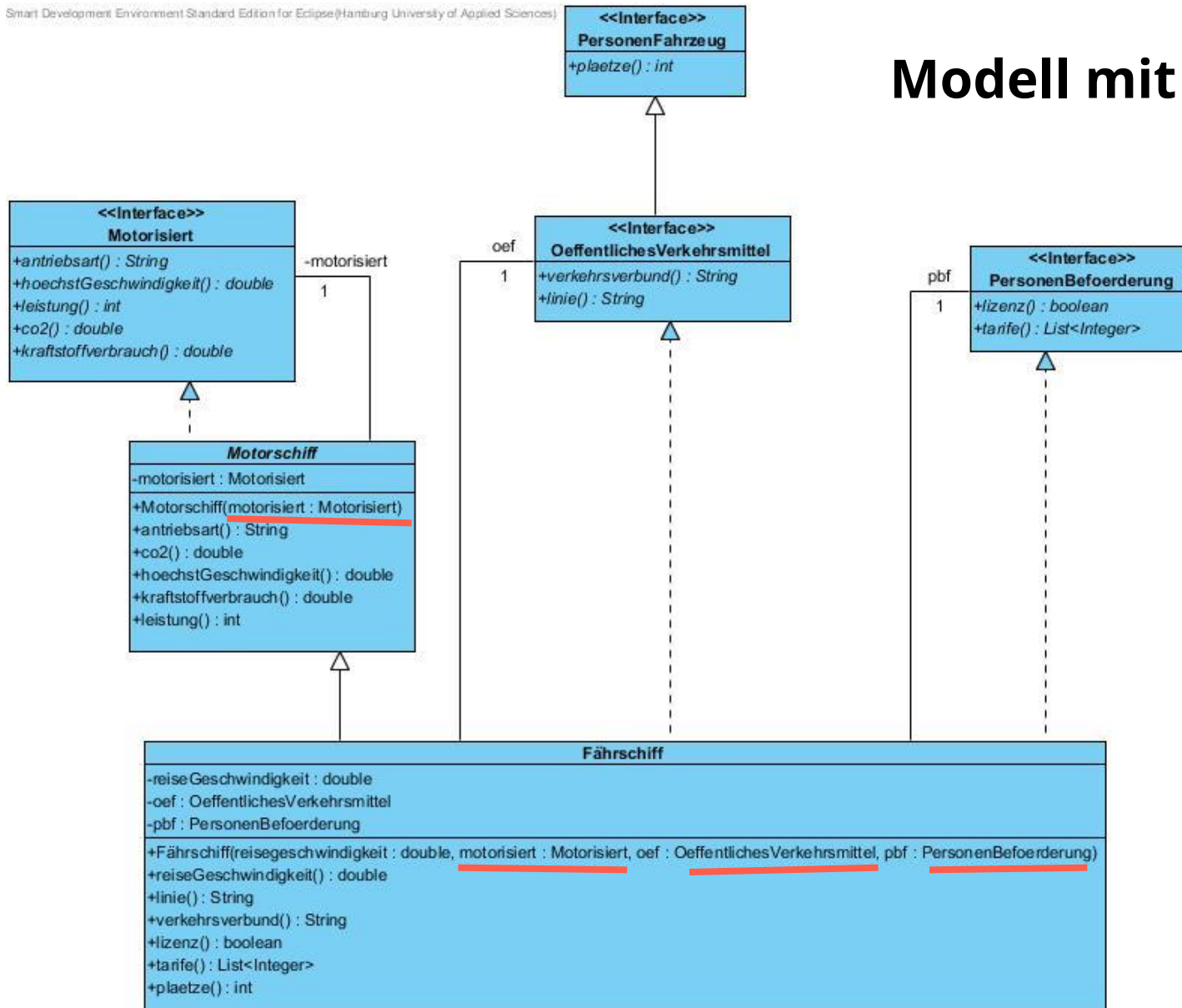


Delegates

- Im Fahrzeugbeispiel beschreiben die Interfaces verschieden Aspekte eines Objektes, die sich entweder als Komponente oder als Eigenschaftsbündel in separaten Klassen zusammenfassen lassen.
- Wir setzen die Fahrzeugobjekte aus Komponenten und Eigenschaftsbündeln zusammen.
- Bei der Konstruktion der Fahrzeugobjekte übergeben wir Objekte vom Typ *Motorisiert*, *PersonenBefoerderung* etc.
- Die Interfacemethoden implementieren wir, indem wir den Methodenaufruf auf die Klassen für Komponenten und Eigenschaftsbündel abbilden. Man spricht auch von delegieren der Methodenaufrufe.
- Die Objekte, deren Schnittstelle Teilaspekte der Schnittstelle einer Klasse bereitstellen heißen auch **Delegates**.



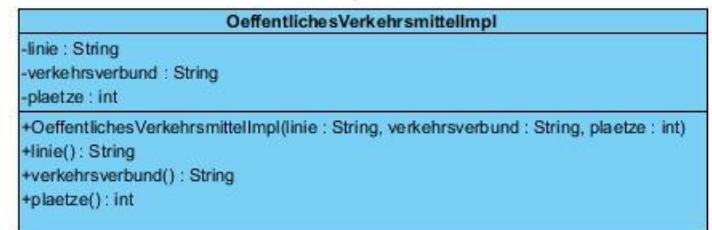
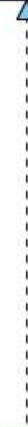
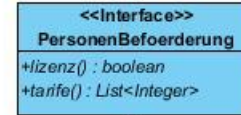
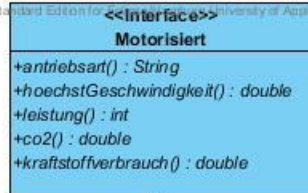
Modell mit Delegates





Implementierung der Delegates

Smart Development Environment Standard Edition for (University of Applied Sciences)





Implementierung der Delegates

Package *fahrzeugev4*, Klasse *Anwendungenv4.java*

- Die Klasse *PersonenBefoerderungImpl* eine Implementierung von *PersonenBefoerderung*, die Klasse *MotorisiertImpl* eine Implementierung von *Motorisiert*, etc.

```
public class PersonenBefoerderungImpl implements PersonenBefoerderung {  
  
    private boolean lizenz;  
    private List<Integer> tarife;  
  
    public PersonenBefoerderungImpl(boolean lizenz, List<Integer> tarife) {  
        this.lizenz = lizenz;  
        this.tarife = tarife;  
    }  
    @Override  
    public boolean lizenz() {return lizenz;}  
    @Override  
    public List<Integer> tarife() {return tarife;}  
}
```



Delegates verwenden

Package *fahrzeugev4*, Klasse *Anwendungenv4.java*

- Wenn wir Fahrzeugobjekte erzeugen, dann übergeben wir jetzt die Delegate-Objekte im Konstruktor der Fahrzeugobjekte.

```
private static void co2Ausstoss() {  
    List<Motorisiert> lm = new ArrayList<Motorisiert>();  
  
    Motorisiert motorisiert = new MotorisiertImpl("Diesel", 500, 80, 200,766);  
    OeffentlichesVerkehrsmittel oef = new OeffentlichesVerkehrsmittelImpl("N67", "HVV", 500);  
    PersonenBefoerderung pbf = new PersonenBefoerderungImpl(false,new ArrayList<Integer>());  
  
    Fährschiff f1 = new Fährschiff(65, motorisiert, oef, pbf);  
  
    lm.add(f1);  
    double co2 = co2Ausstoss(lm);  
    p("CO2 " + co2);  
}
```



Delegates verwenden

Package *fahrzeugev4*, Klasse *Anwendungenv4.java*

- Die Klassen merken sich die Komponenten in Instanz-Variablen.

```
public class Faehrschiff extends Motorschiff implements
    OeffentlichesVerkehrsmittel, PersonenBefoerderung {
    private double reiseGeschwindigkeit;
    private OeffentlichesVerkehrsmittel oef;
    private PersonenBefoerderung pbf;

    public Faehrschiff(double reisegeschwindigkeit,
        Motorisiert motorisiert,
        OeffentlichesVerkehrsmittel oef, PersonenBefoerderung pbf) {
        super(motorisiert);
        this.oef = oef;
        this.pbf = pbf;
        this.reiseGeschwindigkeit = reisegeschwindigkeit;
    }
}
```




Delegates verwenden

Package *fahrzeugev4*, Klasse *Anwendungenv4.java*

- Die Klassen implementieren die Interfacemethoden durch „Weiterleiten“ des Aufrufs an eine entsprechende Komponente, das **Delegate**.

```
public class Faehrschiff extends Motorschiff implements
    OeffentlichesVerkehrsmittel, PersonenBefoerderung {

    @Override
    public String linie() {return oef.linie();}

    @Override
    public String verkehrsverbund() {return oef.verkehrsverbund();}

    @Override
    public boolean lizenz() {return pbf.lizenz();}

    @Override
    public List<Integer> tarife() {return pbf.tarife();}

    @Override
    public int plaetze() {return oef.plaetze();}

}
```



Zusammenfassung

Interfaces als Konzept zur Flexibilisierung statischer Typisierung

- **Interfaces sind separate Typdeklarationen ohne Implementierung**
 - Sie beschreiben Typeigenschaften, indem sie das Protokoll (Spezifikation der Methoden) eines Typs vorgeben, ohne dieses zu implementieren: **Vererbung der Spezifikation**.
 - Klassen, die Interfaces implementieren, sichern die Einhaltung des Protokolls zu.
 - Der Compiler überprüft, ob die Klasse die Schnittstelle des Typs implementiert.
 - Eine Klasse kann mehrere Interfaces implementieren → Mehrfachvererbung der Spezifikation.
 - Über Interfaces erhalten Objekte einer Klasse mehr als einen Typ.
 - Seit Java 1.8 sind auch Implementierungen in Interfaces möglich (dazu später mehr)
- **Nachteil:**
 - Die Wiederverwendung der Implementierung ist über Interfaces nicht möglich.
 - **Aber:** Delegationsmustern mildern diesen Nachteil.

Zusammenfassung

Interfaces als Konzept zur Flexibilisierung statischer Typisierung



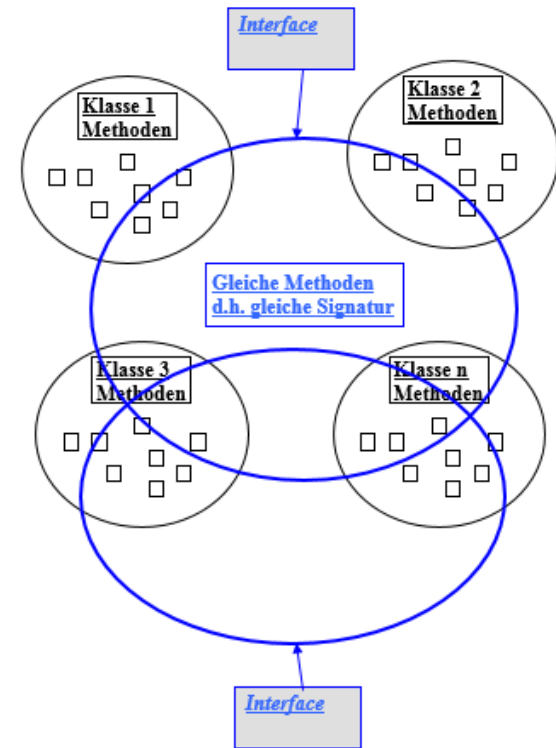
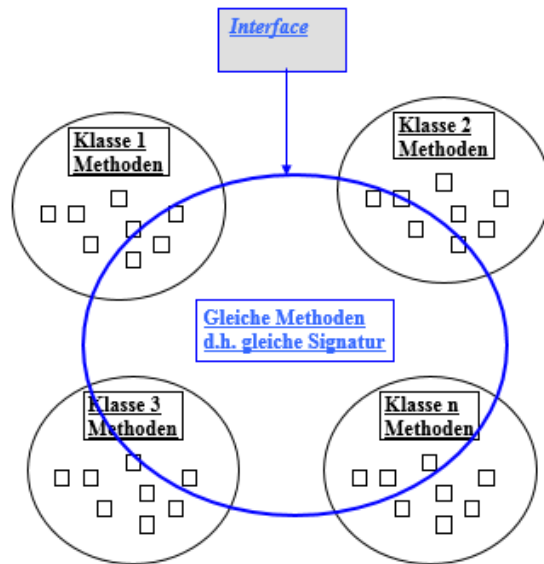
- **Klassen ermöglichen Implementierungsvererbung**
 - durch Subklassenbildung.
 - Subklassen können die Implementierung durch Überschreiben von Methoden ersetzen oder erweitern.
- **Nachteil:**
 - Nur einfache Vererbung. Der Typ einer Klasse kann nur auf die Subklassen vererbt werden.
 - Andere potentiell Typ-kompatible Klassen können zu einem solchen Typ nicht konform erklärt werden.
 - Klassenvererbung führt automatisch zu Typvererbung.



Zusammenfassung

Interfaces als Konzept zur Flexibilisierung statischer Typisierung

- Interfaces definieren unterschiedliche Aspekte oder Rollen eines Objektes:
- Klassen, die das gleiche Interface implementieren, besitzen einen gemeinsamen Teil-Typ.
- **1** Interface kann **n** Implementierungen haben
- **1** Klasse kann **n** Interfaces implementieren.





Multiple Interfaces am Beispiel Person

- Objekte der Klasse *Person* sollen **vergleichbar** sein, da wir aus einer Datenbank mit Personen eine sortierte Liste für ein Adressbuch erzeugen wollen.
- Gleichzeitig wollen wir tiefe **Kopien** von Personen-Objekten erzeugen können, um unerwünschte Änderungen an den Originalen zu verhindern.
- Technisch setzen wir die beiden Anforderungen um, indem wir Personen *Comparable* und *Cloneable* machen.
- *Cloneable* ist ein Interface in Java, das für die Kopierbarkeit eines Objektes steht. In *Object* ist die Methode *clone* bereits implementiert. Da Kopien eine teure Operation sind, sind nicht standardmäßig alle Objekte tief kopierbar. Daher ist die Methode *protected*. Bei Bedarf kann sie von Subklassen überschrieben und freigegeben werden (durch Erweitern der Sichtbarkeit auf *public*) werden kann.
- Damit haben Personen Objekte 4-Typen: *Person*, *Object*, *Comparable<Person>* und *Cloneable*.



Multiple Interfaces am Beispiel Person

- ***Object.clone*** erzeugt eine neue Instanz der Klasse des Objektes und kopiert alle Instanz-Variablen aber nicht die Objekte, die diese Variablen referenzieren.
- Wenn die Typen der Instanz-Variablen einer Klasse nur primitive Typen oder immutable Typen (z.B. ***String***) sind, dann erzeugt der Aufruf von ***super.clone*** eine tiefe Kopie eines Objektes.
- Wenn die Klasse das Interface ***Cloneable*** nicht implementiert, dann erzeugt Objekt clone eine ***CloneNotSupportedException***.



Multiple Interfaces am Beispiel Person

```
public class Person implements Comparable<Person>, Cloneable {

    private String vorname, name;
    public Person(String vor, String na){ ...}

    @Override
    public int compareTo(Person po) {
        int res = name.compareTo(po.name);
        return (res==0) ? vorname.compareTo(po.vorname): res;
    }

    @Override
    public Object clone() throws CloneNotSupportedException {
        Person p = (Person)super.clone();
        return p;
    }

    ... // getter und setter
    public String toString() { ...}
```



Multiple Interfaces am Beispiel Person

- Da **Person** nun **Comparable** und **Cloneable** ist, können wir sie in unterschiedlichen Kontexten verwenden.
- Als **Person**, als **Comparable** für das Sortieren von Personen Arrays, als **Cloneable** zum Kopieren einer Person.

```
Person p1,p2;  
p1 = new Person("Heide", "Simonis");  
p2 = new Person("Peter", "Lustig");
```

hier statt Comparable auch Person möglich

```
Comparable[] persAry = new Person[]{p1,p2};  
Arrays.sort(persAry);  
p(Arrays.toString(persAry));
```

hier statt Comparable auch Person möglich

```
Cloneable c = (Cloneable)p1.clone();  
p("p1 == p2:" + (p1 == c));  
p1.setName("Anders");  
p(p1);  
p(c);
```



```
[Person(Peter,Lustig), Person(Heide,Simonis)]  
p1 == p2:false  
Person(Heide,Anders)  
Person(Heide,Simonis)
```



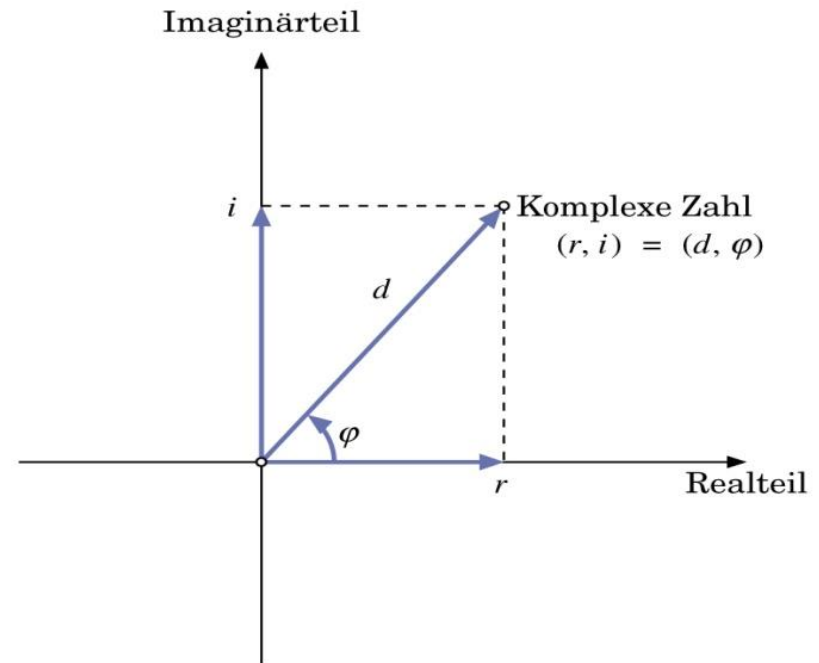

Interfaces als Abstraktion von Implementierungen

- Die zweite wichtige Funktion von Interfaces ist die **Abstraktion** von speziellen Implementierungsklassen.
- Wenn mehr als eine Klasse dasselbe Interface implementiert, dann erreichen wir in Programmen, die Objekte der Klassen nutzen (Client-Programme) und für Variablentypen den Interfacetyp verwenden Unabhängigkeit von konkreten Implementierungen.
- Dadurch lassen sich Implementierungen nachträglich austauschen, ohne dass das nutzende Programm an allen Stellen an denen Objekte des Interfacetyps verwendet werden, geändert werden muss.
- **Beispiel 3 Komplexe Zahlen:**
 - Komplexe Zahlen lassen sich in kartesischen und in Polarkoordinaten darstellen
 - Die Addition und Multiplikation soll für komplexe Zahlen unabhängig von deren Repräsentation formulieren werden



Beispiel 2: Interface für komplexe Zahlen

- **Komplexe Zahlen** = Punkte in einer Koordinatenebene mit einer „reellen“ Achse und einer „imaginären“ Achse.
- **Kartesische** Darstellung
 - Realteil r
 - Imaginärteil i
- Darstellung mit **Polarkoordinaten**
 - Abstand von Ursprung d
 - Drehwinkel φ („Phase“)
- Jede komplexe Zahl hat einen Realteil, Imaginärteil, Ursprungsabstand und eine Phase.
- Kartesische Darstellung und Polarkoordinaten sind äquivalente Darstellung einer komplexen Zahl.





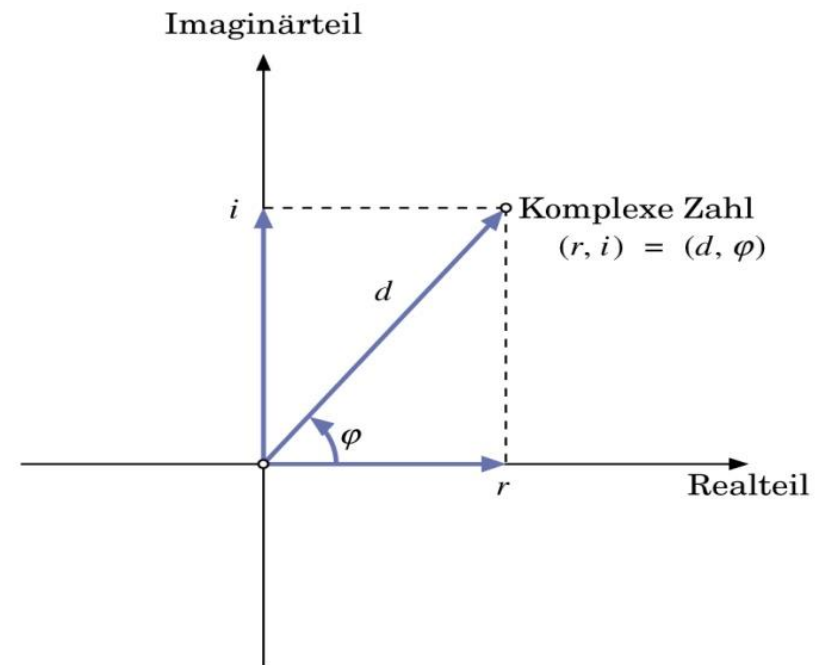
Beispiel 2: Interface für komplexe Zahlen

- **Aufgabe:** Schreiben Sie ein Programm mit dem Sie komplexe Zahl addieren und multiplizieren können.
- Die Formel für die Addition komplexer Zahlen ist besonders einfach für die kartesische Darstellung:

$$(r_1, i_1) + (r_2, i_2) = (r_1 + r_2, i_1 + i_2)$$

- Die Formel für die Multiplikation komplexer Zahlen ist besonders einfach für die Darstellung in Polarkoordinaten:

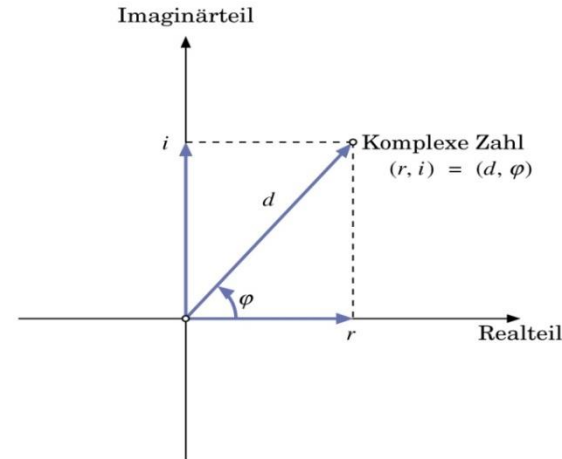
$$(d_1, \varphi_1) * (d_2, \varphi_2) = (d_1 * d_2, \varphi_1 + \varphi_2)$$





Beispiel 2: Interface für komplexe Zahlen

- Um Addition und Multiplikation in der jeweils dafür am besten geeigneten Darstellung durchführen zu können, fordern wir von allen komplexen Zahlen, dass sie Umwandlungsmethoden in die jeweils andere Darstellung anbieten:
- Jede komplexe Zahl muss die Methoden ***getReal***, ***getImag***, ***getDist*** und ***getPhase*** anbieten.
- Um diese Forderung Typ-sicher durchsetzen zu können, definieren wir ein Interface ***Complex***, das die mathematischen Operationen und Konvertierungsmethoden spezifiziert:

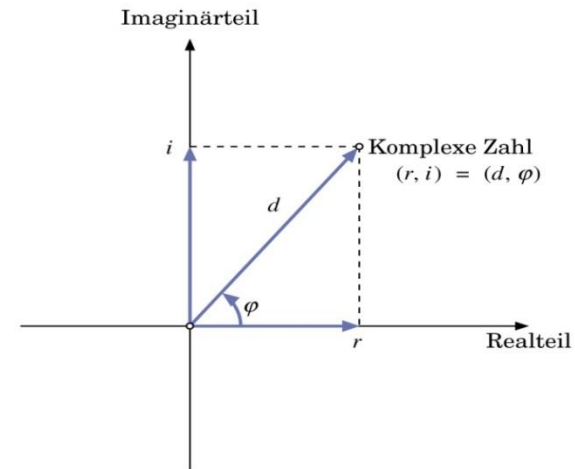


```
public interface Complex {  
    double getReal();  
    double getImag();  
    double getDistance();  
    double getPhase();  
    public Complex add(Complex c);  
    public Complex mult(Complex c);  
}
```



Klasse *Cartesian*

```
public class Cartesian implements Complex {  
  
    private final double real;  
    private final double imag;  
  
    ...  
    public double getImag() {  
        return imag;  
    }  
  
    public double getReal() {  
        return real;  
    }  
    public double getDist() {  
        return Math.hypot(real, imag);  
    }  
  
    public double getPhase() {  
        return Math.atan2(imag, real);  
    }  
}
```



Für die Umrechnung von kartesischen in Polar Koordinaten implementieren wir die Methoden *getDist*, *getPhase*, *getImag*, *getReal*

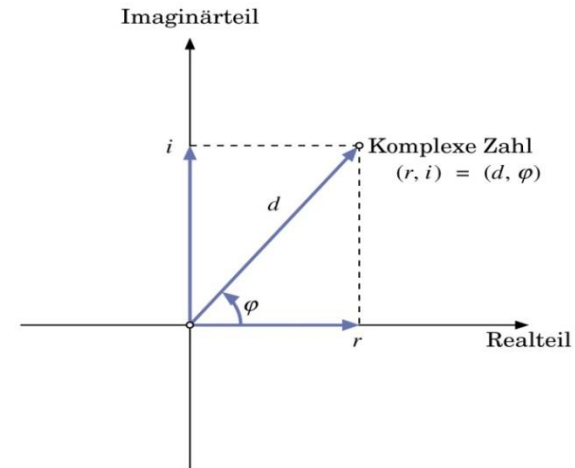
Math.hypot() → Satz des Pythagoras

Math.atan2() → berechnet den Winkel ϕ unter Berücksichtigung des Quadranten



Klasse *Polar*

```
public class Polar implements Complex {  
  
    private final double dist;  
    private final double phase;  
  
    ...  
    public double getDist() {  
        return dist;  
    }  
    public double getPhase() {  
        return phase;  
    }  
    public double getImag() {  
        return dist * Math.cos(phase);  
    }  
    public double getReal() {  
        return dist * Math.sin(phase);  
    }  
    ...  
}
```



Für die Umrechnung von Polar in kartesische Koordinaten implementieren wir die Methoden *getDist*, *getPhase*, *getImag*, *getReal*



Addition und Multiplikation für *Polar*

- Für die Addition werden Abstand und Phase in Real- und Imaginär-Darstellung konvertiert. Es werden ausschließlich die Konvertierungsmethoden verwendet (*get...*)
- In der Multiplikation können Abstand und Phase von *this* direkt verwendet werden, von der Zahl *c* sind nur die Konvertierungsmethoden sichtbar (*get...*).
- *mult* hat als einen **kovarianten** Ergebnistyp: *Polar* ist kompatibel zu *Complex*. *Complex* war als Ergebnistyp der Interfacemethode *mult* gefordert.

```
public class Polar implements Complex {  
    ...  
    public Complex add(Complex c) {  
        return new Cartesian(getReal()+c.getReal(),getImag()+c.getImag());  
    }  
    public Polar mult(Complex c) {  
        return new Polar(dist*c.getDist(),phase+c.getPhase());  
    }  
}
```



Addition und Multiplikation für *Cartesian*

- Für die Multiplikation werden Real- und Imaginärteil in Abstand und Phase konvertiert. Es werden ausschließlich die Konvertierungsmethoden verwendet (*get...*)
- In der Multiplikation können Real und Imaginärteil von *this* direkt verwendet werden, von der Zahl *c* sind nur die Konvertierungsmethoden sichtbar (*get...*).
- *add* hat als einen kovarianten Ergebnistyp: *Cartesian* ist kompatibel zu *Complex*. *Complex* war als Ergebnistyp der Interfacemethode *add* gefordert.

Ergebnistyp darf konkreter sein als
Interfacetyp

```
public class Cartesian implements Complex {  
    ...  
    public Cartesian add(Complex c) {  
        return new Cartesian(real+c.getReal(), imag+c.getImag());  
    }  
    public Complex mult(Complex c) {  
        return new Polar(getDist()*c.getDist(), getPhase()+c.getPhase());  
    }  
}
```




Verwendung der komplexen Zahlen

- Da beide Klassen vom Typ **Complex** sind, können wir einer Variablen vom Typ **Complex** Objekte vom Typ **Cartesian** oder **Polar** zuweisen.
- Da die Addition und Multiplikation auf **Complex** Objekten definiert ist, können wir Objekte vom Typ **Polar** und **Cartesian** miteinander addieren / multiplizieren.
- Zur Laufzeit entscheidet dann der dynamische Typ über die Auswahl der Methode.

```
Complex c1 = new Polar(2,2.34);  
Complex c2 = new Cartesian(1,2);
```

```
p(c1.add(c2));  
p(c1.add(c1));  
p(c1.mult(c2));  
p(c2.mult(c1));
```



```
Cartesian(2.43693,0.608873)  
Cartesian(2.87386,-2.78225)  
Polar(4.47214,2.59073)  
Polar(4.47214,2.59073)
```

Kopier-Konstruktoren für die Klassen *Polar* und *Cartesian*



- Kopier-Konstruktoren kopieren Inhalte aus Objekten desselben Typs.
- Wenden wir dies auf die Klasse *Polar* an, dann erhalten wir einen Konstruktor, der nur aus komplexen Polarobjekten kopieren kann. (**Variante 1**)
- Besser ist es in diesem Fall einen allgemeinen Kopier-Konstruktor für *Complex* Objekte zu definieren, dann kann zwischen *Polar* und *Cartesian* kopiert werden. (**Variante 2**)
- Dies gilt insbesondere hier, da es sich bei beiden Klassen um äquivalente Darstellungen einer komplexen Zahl handelt.

```
public class Polar implements Complex {  
    private final double dist;  
    private final double phase;  
    // Variante 1  
    public Polar(Polar p){  
        this.dist= p.dist;  
        this.phase= p.phase;  
    }  
    // Variante 2  
    public Polar(Complex c){  
        this.dist = c.getDist();  
        this.phase = c.getPhase();  
    }  
    ...  
}
```

Kopier-Konstruktoren für die Klassen *Polar* und *Cartesian*



- Der allgemeine Kopier-Konstruktor greift nicht direkt auf die Objektvariablen zu, sondern verwendet die Konvertierungsmethoden (*get...*).
- Der statische Typ *Complex* für den Parameter des Konstruktors reicht aus, um die Kopier-Konstruktoren sicher zu übersetzen.
- Welches Objekt zur Laufzeit übergeben wird, spielt für den Konstruktor keine Rolle, da zur Laufzeit die *get...* Methoden des dynamischen Typs die notwendigen Konvertierungen durchführen.

```
public class Polar implements Complex {  
    private final double dist;  
    private final double phase;  
    public Polar(Complex c){  
        this.dist = c.getDist();  
        this.phase = c.getPhase();  
    }  
    ...  
}  
  
Complex c1= new Polar(new  
    Cartesian(1,2));  
Complex c2= new Polar(new  
    Polar(1.5,0.8));  
Complex c3= new Cartesian(new  
    Cartesian(1,2));  
Complex c4= new Cartesian(new  
    Polar(1.5,0.8));
```



Regeln zu Interfaces

Implementierung von Interfaces

- Klassen, die ein Interface implementieren, zeigen dies durch *implements* an.
- Klassen sichern dem Compiler zu, dass sie die Methoden des Interfaces anbieten.
- Konkrete Klassen, die ein Interface implementieren, müssen alle Methoden des Interfaces **überschreiben (override)**.
- Die Ergebnistypen der Methoden dürfen **kovariant** sein.
- Abstrakte Klassen **können** Interface-Methoden überschreiben

Interfaces und Mehrfachvererbung

- Interfaces können von mehreren Interfaces ableiten.
interface I1 extends I2,I3 {}
- Klassen können mehrere Interfaces implementieren, aber nur von einer Klasse ableiten.
public class C2 extends C1 implements I1, I5 {}
- Java unterstützt nur Einfachvererbung der Implementierung, aber durch Interfaces Mehrfachvererbung der Spezifikation.



Interfaces haben nur *public* Methoden und "Klassenkonstanten"

public Methoden

- fehlt der *public* Modifikator für die Methoden eines Interfaces, ergänzt der Compiler diesen.

```
public interface Complex {  
    double getReal();  
    double getImag();  
}
```

↕ äquivalent

```
public interface Complex {  
    public double getReal();  
    public double getImag();  
}
```

"Klassenkonstanten"

- Der Compiler ergänzt für alle Attribute die Modifikatoren *public static final*.

```
public interface MathConstants {  
    double PI = 3.4;  
}
```

↕ äquivalent

```
public interface MathConstants {  
    public static final double PI = 3.4;  
}
```



ABSTRAKTE KLASSEN



Abstrakte Klassen

- Abstrakte Klassen liegen zwischen Interfaces und konkreten Klassen.
- **Interfaceaspekte:**
 - Sie erzwingen das Implementieren abstrakter Methoden in ableitenden Klassen.
 - Sie können **nicht** instanziiert werden.
- **Aspekte konkreter Klassen:**
 - Sie können Methoden implementieren.
 - Sie können Attribute und Konstruktoren enthalten.
 - Methoden können *protected*, *private* und package-private sein.



Abstrakte Klassen

- Wir unterscheiden
 - **rein abstrakte** Klassen: enthalten nur abstrakte Methode und keine Datenelemente. Rein abstrakte Klassen sind äquivalent zu Interfaces.
 - **abstrakte** Klassen, die auch Implementierungen enthalten.
- Abstrakte Klassen können von abstrakten Klassen ableiten.
- Subklassen müssen abstrakt sein, wenn sie nicht alle abstrakten Methoden implementieren.
- Klassen müssen abstrakt sein, wenn sie nicht alle Methoden eines Interfaces implementieren.

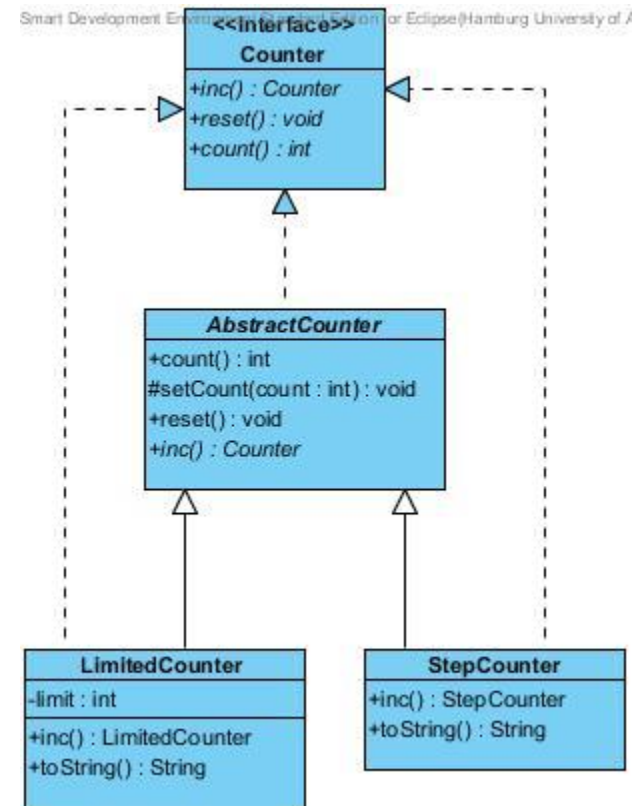


Beispiel 3: Zähler

- Wenn eine Klasse Implementierungen für einige Methoden enthält, aber nicht alle Methoden aller Subklassen implementieren kann, dann verwenden wir abstrakte Klassen.
- Beispiel: Zähler, mit dem Protokoll:
 - **inc()** erhöht den Counter
 - **reset()** setzt den Counter zurück
 - **count()** liefert den Zählerstand.

Diese Methoden fordern wir über ein gemeinsames Interface **Counter**.

- Unsere Zählerhierarchie hat einen **Step Counter**, der um eine vorgegebene Schrittweite hochzählt und einen **LimitedCounter**, der bis zu einem vorgegebenen Limit zählt und dann den Zähler zurücksetzt.





Beispiel 3: Zähler

```
package counter;

public abstract class AbstractCounter
    implements Counter {
    private int count;
    public AbstractCounter(){
        this.count = 0;
    }
    @Override
    public int count() {
        return count;
    }
    protected void setCount(int count){
        this.count = count;
    }
    @Override
    public void reset() {
        count = 0;
    }
    @Override
    public abstract Counter inc();
}
```

- Die Methoden *reset()* und *count()* lassen sich für alle *Counter* gleich implementieren.
- Die Methode *inc()* ist speziell für verschiedene *Counter* und ist daher eine abstrakte Methode in *AbstractCounter*.
- Da das Inkrementieren Zugriff auf den *count* benötigt, müssen wir die Methode *setCount* vorsehen. Damit nur ableitende Klassen *count* modifizieren können, setzen wir die Sichtbarkeit von *setCount* auf *protected*.



Beispiel 3: Zähler

- Konkrete Klassen, die von einer abstrakten Klasse ableiten, müssen die abstrakten Methoden implementieren.
- Die konkrete Klasse *StepCounter* muss die abstrakte Methode *inc()* implementieren.
- Obwohl der Ergebnistyp der abstrakten Methode *Counter* ist, dürfen überschreibende Methoden in Subklassen einen Ergebnistyp verwenden, der Subtyp von *Counter* ist. (**kovarianter Ergebnistyp**).

```
public class StepCounter extends
    AbstractCounter implements Counter{

    private int step;

    public StepCounter(int step) {
        this.step = step;
    }

    @Override
    public StepCounter inc() {
        setCount( count() + step);
        return this;
    }
}
```



Methodenkategorien abstrakter Klassen

- **abstrakte:** Methoden, die Subklassen implementieren müssen. In Java sind abstrakte Methoden nur die mit *abstract* markierten Methoden einer abstrakten Klassen.
- **konkrete:** Methoden, die vollständig in der abstrakten Klasse implementiert sind. Wenn diese Methoden nicht in den Subklassen überschrieben werden sollen, dann können wir in Java diese Methoden als *final* markieren.
- **Template:** Methoden, die abstrakte, oder konkrete Methoden kombinieren und ein gemeinsames Ablaufschema für die Subklassen definieren.
- **Hook:** abstrakte oder konkrete nicht finale Methoden, die in Template-Methoden einer abstrakten Klasse verwendet werden.



ÜBERSCHREIBEN (OVERRIDE)
ÜBERLADEN (OVERLOAD)
ÜBERSCHATTEN (SHADOW)

Überschreiben von Methoden / Überladen von Methoden



Überschreiben

- Eine Methode einer ableitenden Klasse ersetzt die Definition einer Methode **gleicher Signatur** in der Superklasse.
- Die Methode der Superklasse ist dann nicht mehr erreichbar, es sei denn, sie wird mit *super* explizit aufgerufen.
- **Kovarianter Ergebnistyp:** Der Typ des Rückgabewertes der überschreibenden Methode darf ein Subtyp des Typs des Rückgabewertes der Methode der Superklasse sein.

Überladen

- Eine Methode einer Klasse **gleichen Namens** aber **ungleicher Signatur** der Klasse oder einer der Superklassen definiert eine neue Methode.
- Alle Methoden gleichen Namens aber ungleicher Signatur in der Klasse oder deren Superklassen sind weiterhin erreichbar.



Methoden-Signatur und Methodensuche

- Die Signatur einer Methode ist definiert durch Typ und Reihenfolge der Parameter einer Methode.
- Der Ergebnistyp gehört **nicht** zur Signatur.
- **Beispiel:** Die Signatur der Methode *Float m(String arg1, List arg2, int arg3)* ist *m(String, List, int)*
- Der Compiler verwendet die Signatur um die Zulässigkeit eines Methodenaufrufs zu prüfen.
- Die Methodensignatur wird beim Übersetzen eines Programms hinterlegt.
- Die virtuelle Maschine entscheidet zur Laufzeit anhand des dynamischen Typs eines Objektes, in welcher Klasse die Suche nach der Methode mit der hinterlegten Signatur beginnt.
- Dabei sucht die virtuelle Maschine nach der Methode, die am besten zur hinterlegten Signatur passt.



Methoden-Signatur und Methodensuche

- **Beispiel:** Die Signatur der Methode
public boolean equals(Object o) ist *equals(Object)*

Die Signatur der Methode

public boolean equals(Person p) ist *equals(Person)*

equals(Person) überlädt *equals(Object)*

- Wir schauen uns an einigen Beispielen den Effekte mit überladenen Methoden an.



Überladen von *equals* und Inhaltsgleichheit

- Die Klasse *Person* enthält eine Implementierung der Methode *equals*, die das *equals* von *Object* überlädt.
- Sourcecode des Beispiels in *package overloadoverridesshadow;*

```
public class Person {  
    private String name;  
  
    public Person(String name) {  
        this.name = name;  
    }  
  
    public boolean equals(Person obj) {  
        if (this == obj)  
            return true;  
        if (obj == null)  
            return false;  
        if (name == null) {  
            if (obj.name != null)  
                return false;  
        } else if (!name.equals(obj.name))  
            return false;  
        return true;  
    }  
}
```



Überladen von *equals* und Auffinden von Objekten

- Wir erzeugen zwei Personen *p1* und *p2* mit gleichem Namen und prüfen, ob diese beiden Personen gleich sind.
- Da der statische Typ von *p2* *Person* ist, hinterlegt der Compiler die Methodensignatur *equals(Person)*.
- Die virtuelle Maschine sucht zur Laufzeit nach einer Methode, die am Besten zur Signatur *equals(Person)* passt.
- Da der dynamische Typ von *p1* *Person* ist, beginnt die Suche in der Klasse *Person*.
- Da die Methode *equals(Person)* der Klasse *Person* exakt zu der zur Compile-Zeit hinterlegten Signatur passt wird diese Methode ausgeführt.
- Das Ergebnis ist wie erwartet *true*.

```
Person p1 = new Person("Ada");  
Person p2 = new Person("Ada");  
p("p1.equals(p2)" + p1.equals(p2));
```



Überladen von *equals* und Auffinden von Objekten

- Dann erzeugen wir 2 Personen *p3* und *p4* und wählen als statischen Typ für *p4 Object*.
- Dann führen wird die Gleichheitsprüfung erneut durch. Jetzt ist das Ergebnis *false*. Warum?
- Da der statische Typ von *p4 Object* ist, hinterlegt der Compiler die Methodensignatur *equals(Object)*.
- Die virtuelle Maschine sucht jetzt zur Laufzeit in der Klasse *Person* nach einer Methode mit der Signatur.
- Diese Methode wird in Klasse *Person* nicht gefunden. Daher wird die *equals(Object)* Methode der Klasse *Object* verwendet.
- Die Methode in *Object* prüft auf Identität.
- Da *p3* und *p4* nicht identisch sind, ist das Ergebnis *false*.

```
Person p1 = new Person("Ada");
Person p2 = new Person("Ada");
p("p1.equals(p2)" + p1.equals(p2));
Person p3 = new Person("Ada");
Object p4 = new Person("Ada");
p("p3.equals(p4)" + p3.equals(p4));
```



Überladen von *equals* und Auffinden von Objekten

- Wenn wir *p1* in ein *Person*-Array eintragen und mit *p2* suchen, finden wir *p1*
- Wenn wir *p1* in ein *Object*-Array eintragen, finden wir *p1* nicht mehr über *p2*.
- Da statische Typ von *p1 Object* ist, verwendet der Compiler die Signatur *equals(Object)*. Diese Methode ist aber in Klasse *Person* nicht implementiert, daher wird die Implementierung in Klasse *Object* verwendet, die auf Identität prüft.

```
Person[] pa = {p1,p1};
int cntEqual = 0;
for (Person o : pa) {
    if( o!= null && o.equals(p2)) {
        cntEqual ++;
    }
}
p("found p2 in " + Arrays.deepToString(pa) +
  " " + cntEqual + " times");
```

```
Object[] oa = {p1,p1};
cntEqual = 0;
for (Object o : oa) {
    if( o!= null && o.equals(p2)) {
        cntEqual ++;
    }
}
p("found p2 in " + Arrays.deepToString(oa) +
  " " + cntEqual + " times");
```



Überladen von *equals* und Auffinden von Objekten

```
Person[] pa = {p1,p1};
int cntEqual = 0;
for (Person o : pa) {
    if( o!= null && o.equals(p2)) {
        cntEqual ++;
    }
}
p("found p2 in " + Arrays.deepToString(pa)
  + " " + cntEqual + " times");
```

found p2 in [Person [name=Ada], Person
[name=Ada]] 2 times

```
Object[] oa = {p1,p1};
cntEqual = 0;
for (Object o : oa) {
    if( o!= null && o.equals(p2)) {
        cntEqual ++;
    }
}
p("found p2 in " +
  Arrays.deepToString(oa) + " " +
  cntEqual + " times");
```



found p2 in [Person [name=Ada], Person
[name=Ada]] 0 times



Überladen von *equals* und Inhaltsgleichheit

- Jetzt schreiben wir *p1* in eine *List<Person>*. Die Methode *contains* zum Auffinden von Objekten hat die Signatur *contains(Object)* und verwendet intern die Methode mit der Signatur *equals(Object)*.
- Daher wird *p1* nicht gefunden, wenn wir mit *p2* anfragen.

```
List<Person> lp = new ArrayList<Person>();  
lp.add(p1);  
p(lp.contains(p2));
```



false

Überschatten (engl. Shadowing) in Subklassen



- Instanz-Variablen oder statisch gebundene Methoden mit gleicher Signatur wie Methoden der Superklasse, überschatten die Methoden der Superklasse.
- Im Gegensatz zum Überschreiben von Methoden, entscheidet bei Überschattungen mit statisch gebundenen Methoden, der statische Typ des Objektes, auf dem die Methode aufgerufen wird, in welcher Klasse die Methodensuche beginnt.
- D.h. Der Compiler hinterlegt bei der Übersetzung des Programms, welche Methodendefinition / welcher Attributwert verwendet wird.



Überschatten von Methoden (shadowing)

- Statisch gebundene Methoden gleicher Signatur überschatten sich.
- Um diesen Effekt zu vermeiden, sollten statische Methoden immer direkt auf der Klasse aufgerufen werden.
- Dies gilt auch für private Instanz-Methoden. Im Beispiel überschattet die Methode *inMe()* in der Klasse **A** die Methode der Klasse **B**.
- Wenn der Compiler den Aufruf einer privaten Methode entdeckt, dann bindet er die Methodendefinition der Klasse, in der die private Methode verwendet wird.

```
A b = new B();  
b.showMe();  
((B)b).showMe();  
b.thatsMe();  
((B)b).thatsMe();
```



```
A  
B  
inA  
inA
```

```
class A {  
    static void showMe() {p("A");}  
    private void inMe() {p("inA");}  
    void thatsMe() {inMe();}  
}  
  
class B extends A {  
    static void showMe() {p("B");}  
    private void inMe() {  
        p("inB");  
    }  
}
```




ZUSAMMENFASSUNG



Zusammenfassung Interfaces (Konzepte)

- Interfaces repräsentieren unterschiedliche Rollen von Objekten.
- Interfaces definieren **Teiltypen** von Objekten.
- Interfaces sind ein Mittel, um Spezifikationen zu vererben. (*Comparable* und *Complex* Beispiel)
- Klassen können mehrere Interfaces implementieren.
- Mehrfachvererbung ist in Java nur durch Interfacevererbung möglich.
- Delegation ermöglicht Wiederverwendung von Code bei Interfacevererbung.



Zusammenfassung Interfaces (Techniken)

- Interfaces können nicht instanziiert werden.
- Klassen zeigen die Implementierung eines Interfaces durch das Schlüsselwort *implements* an.
- Klassen, die ein Interface implementieren, müssen alle Methoden des Interfaces implementieren oder **abstrakte** Klassen sein.
- Alle **Methoden** eines Interfaces sind *public*.
- Interfaces enthalten nur *public static final* **Attribute**.



Zusammenfassung abstrakte Klassen

- Abstrakte Klassen sind eine Kombination von Interface und konkreten Klassen. Sie ermöglichen Vererbung der Spezifikation und Implementierungsvererbung.
- Abstrakte Methode werden mit *abstract* markiert und müssen in den konkreten Subklassen implementiert werden. Der Compiler prüft dies bei der Übersetzung des Programms.
- Konkrete Methoden abstrakter Klassen lassen sich anhand des Schlüsselwortes *final* identifizieren.
- Ein gutes Beispiel für die Kombination aus Interfacehierarchie und Hierarchie abstrakter Klassen finden wir in der Java Bibliothek für Objektsammlungen



Zusammenfassung statischer dynamischer Typ

- Der statische Typ ist der Typ einer Variablen / eines Objektes zur Compilezeit.
- Der statische Typ entscheidet in Java darüber, welche Methoden auf einer Variable / einem Objekt aufgerufen werden dürfen.
- Ein Cast ändert den statischen Typ einer Variablen / eines Objektes.
- Statische Typisierung unterstützt Typsicherheit zur Übersetzungszeit.
- Dennoch können Typfehler zur Laufzeit auftreten: Fehlerhafte Casts, Kovarianz von Arrays.



Zusammenfassung statischer dynamischer Typ

- Der dynamische Typ ist der Typ des Objektes, das zur Laufzeit an eine Variable gebunden ist.
- Der dynamische Typ entscheidet darüber, wo zur Laufzeit die Methodensuche beginnt und damit, welche Implementierung der Methode verwendet wird.
- Der Typ des Objektes bestimmt die Klasse, in der die Methodensuche beginnt.
- Die Methodenauswahl heißt auch Binden der Methode. Man spricht daher auch von **dynamischem Binden** einer Methode.

Zusammenfassung Überladen, Überschreiben, Überschatten



- Methoden gleichen Namens unterschiedlicher Signatur überladen sich gegenseitig.
- Eine Methode gleichen Namens und gleicher Signatur in einer Subklasse überschreibt die Methode der Superklasse.
- Der Compiler hinterlegt beim Übersetzen eines Programmes die Methodensignatur der Methodendefinitionen und der Methodenaufrufe. Die Signatur der Methodenaufrufe basiert auf den statischen Typen der Argumente.
- Zur Laufzeit wird auf der Basis des dynamischen Typs des Objektes, auf dem die Methode aufgerufen wird, nach der Methodendefinition gesucht, die am Besten zur hinterlegten Signatur des Aufrufs passt.
- Statische Methoden und private Methoden gleicher Signatur überschatten einander.